PROJECT

THOR

TITLE

Porting the GNU C Compiler to the Thor Microprocessor

	Name	Function	Date	Signature
Prepared	Harry Gunnarsson Thomas Lundqvist	Master Thesis Project		
Checked	Stefan Asserhäll	Thor Wizard		
Authorized	Bo Ernbert	Project Manager		

Distribution
Complete: CK
Summary: CA B1B, CP, DP, CV, CKPL, CLAC, CSAC, SA

From		Saab Ericsson Space AB		Telephone	Telefax	Visiting address
	Head office:	S-405 15 GÖTEBORG	SWEDEN	+46 31 35 00 00	+46 31 35 95 20	Delsjömotet
	Branch office:	S-581 88 LINKÖPING	SWEDEN	+46 13 28 64 00	+46 13 13 16 28	Bröderna Ugglas gata

Class: CUSTOMER Host System: FrameMaker 4.0
Contract No: Host File: B1_DATA/THOR/TNT/0028_01

SUMMARY

In this master thesis project an attempt has been made to port the freeware GNU C Compiler (version 2.7.0) to the Thor microprocessor. The resulting compiler compiles ANSI C programs and generates assembler code for Thor. The compiler is not validated, and to be useful it also needs an implementation of the standard C library.

The GNU C Compiler has a world wide reputation of being a very good compiler able to deliver fast and reliable code. Furthermore, the design of the compiler is made with the thought of making ports to several different machines possible. Naturally, that is the most important reason for choosing GNU CC for this project. Since the GNU C Compiler is a freeware program, even the source code is freely accessible, and without the source code the project would have been impossible.

The Thor microprocessor is designed for use in space borne embedded computer systems. The instruction set is stack oriented and this feature makes it unique compared to other architectures available today.

One problem that made the project very interesting was the fact that GNU CC is intended to work on traditional architectures with a set of registers. On Thor there are no register due to the stack oriented instruction set.

DOCUMENT CHANGE RECORD

Changes between issues are marked with a left-bar

Issue	Date	Paragraphs affected	Change Information
1	951204	all	New document

TABLE OF CONTENTS Page

1 Introduction 7
1.1 Background 7
1.2 Definition and goal of the GNU CC-Thor project 7
1.3 References to similar works 8

2 Description of the processor and the compiler 9
2.1 Description of the Thor microprocessor 9
2.1.1 Microprocessor overview 9
2.1.2 Architecture and instruction set 10
2.1.3 The pipeline 12
2.1.3.1 Instruction fetch stage 12
2.1.3.2 Address Generation Stage 13
2.1.3.3 Operand Fetch Stage 13
2.1.3.4 Execute Stage 13
2.1.4 Pipeline control 14
2.1.5 Data cache 14
2.1.5.1 The update memory buffer 15
2.1.5.2 Hazards 15
2.1.5.3 Snooping 15
2.1.6 Error detection 15
2.1.6.1 Comparator Function 15
2.1.6.2 EDAC 16
2.1.6.3 Program Flow Control 16
2.1.7 An example of a Thor program 16
2.2 Description of the GNU C compiler 17
2.2.1 Passes 18
2.2.1.1 Parsing 19
2.2.1.2 RTL generation 20
2.2.1.3 Jump optimization 20
2.2.1.4 Register scan 20
2.2.1.5 Jump threading 20
2.2.1.6 Common subexpression elimination (CSE) 20
2.2.1.7 Loop optimization 21
2.2.1.8 Stupid Register Allocation 21
2.2.1.9 Data flow analysis 21
2.2.1.10 Instruction combination 21
2.2.1.11 Instruction scheduling 21
2.2.1.12 Register class preferencing 22
2.2.1.13 Local register allocation 22
2.2.1.14 Global register allocation 22
2.2.1.15 Reloading 22

TABLE OF CONTENTS Page

2.2.1.16 Instruction Scheduling, second pass 22
2.2.1.17 Jump optimization, second pass 22
2.2.1.18 Delayed branch scheduling 22
2.2.1.19 Intel 80387 special pass....22
2.2.1.20 Final....22
2.2.1.21 Debugging information output 23
2.2.2 Running the compiler 23
2.2.2.1 Overall options 23
2.2.2.2 Debugging options 24
2.2.2.3 Optimizing options....25
2.2.3 Intermediate representation 26
2.2.4 Machine Description 27
2.2.4.1 The Machine Description file, ’.md’-file 27
2.2.4.2 Target Description Macros, ’.h’-file....33

3 GNU C Compiler for the Thor microprocessor 35
3.1 Problems 35
3.2 Strategy....36
3.2.1 The lack of registers in Thor 37
3.2.2 Frame pointer elimination 38
3.2.3 The 8-bit addressing problem....38
3.2.4 The representation of ’PSH’ and ’POP’ 40
3.3 Solution....42
3.3.1 Specification of system dependent definitions....43
3.3.1.1 The basic C data types....43
3.3.1.2 Function calling interface....43
3.3.2 Target control macros....48
3.3.2.1 Thor-specific macros....48
3.3.2.2 Link sections....48
3.3.2.3 Type and Storage layout....49
3.3.2.4 Registers and Register Classes....49
3.3.2.5 Stack and calling 51
3.3.2.6 Addressing modes 53
3.3.2.7 Condition codes....54
3.3.2.8 Assembler format 55
3.3.2.9 Miscellaneous....56
3.3.2.10 Compiler options that are forced to be activated....57
3.3.3 Machine description instruction patterns....57
3.3.3.1 An example of how a simple C program is handled 60
3.3.3.2 Data move instructions....63
3.3.3.3 Arithmetic operations....64

TABLE OF CONTENTS Page

3.3.3.4 Logic operations 66
3.3.3.5 Negating and one-complementing....67
3.3.3.6 Condition code setting instructions 67
3.3.3.7 Control transfer instructions....68
3.3.4 Attributes 71
3.3.5 Optimizing the code 71
3.3.5.1 Machine-specific peephole optimizations....71
3.3.5.2 Delay slot filling....73
3.3.5.3 Removal of unnecessary test/compare instructions....74
3.3.6 Changes in the source files of GNU CC....74
3.3.6.1 Bugs....74
3.3.6.2 Thor adaptations....76
3.3.7 Auxiliary files in the compiler environment 77
3.3.7.1 The ’crt0.asm’ file 78
3.3.7.2 The ’thor-libgcc1.asm’ file....78
3.3.7.3 The ’as’ and ’ld’ script files 79
3.3.7.4 The ’t-thor’ file....79
3.4 Outcome....79
3.4.1 GNU CC’s preference for registers 79
3.4.2 Comparison with the Oden Ada compiler....82
3.4.3 Interesting details....88
3.5 Remaining work....89
3.5.1 Standard C library functions....89
3.5.2 Validation 90
3.5.3 The compiler....90
3.5.3.1 Debug support 90
3.5.3.2 Simple improvements of the compiler 91

4 Conclusions 92
4.1 Conclusions drawn from our work 92
4.1.1 Advantages and drawbacks with trying to make a port....92
4.1.2 Making a port to the Thor microprocessor....93
4.2 Deficiencies 94
4.2.1 Deficiencies in the Thor Architecture 94
4.2.2 Deficiencies in GNU CC....95
4.3 Future improvements 95
4.3.1 Compiler enhancements 95
4.3.1.1 8-bit characters 95
4.3.1.2 Filling delay slots 96
4.3.1.3 Register allocation....96
4.3.1.4 A different strategy with ’PSH’ and ’POP’....97

TABLE OF CONTENTS Page

4.3.1.5 Move the 'clobber'-instruction....97
4.3.2 Supporting the GNU extended C....97
4.3.3 GNAT and G++....97
4.3.3.1 G++....97
4.3.3.2 GNAT....98
4.4 Did we accomplish our goal?....98

5 Definitions and abbreviations....99

6 Bibliography....100

APPENDIX A - Listing of machine dependent files.... 101
A.1 thor.md....102
A.2 thor.h....115
A.3 thor.c....128
A.4 crt0.asm....138
A.5 t-thor....139
A.6 thor-libgcc1.asm....140
A.7 as....142
A.8 ld....143

APPENDIX B - Diff files of the changes in GNU CC source.... 144
B.1 reload.c versus reload.c.org....144
B.2 reload1.c versus reload1.c.org....144
B.3 reorg.c versus reorg.c.org....145
B.4 stmt.c versus stmt.c.org....145
B.5 jump.c versus jump.c.org....146
B.6 function.c versus function.c.org....147
B.7 config.sub versus config.sub.org....148
B.8 configure versus configure.org....148

APPENDIX C - Instruction set for Thor.... 150

APPENDIX D - List of C validation suites.... 152

APPENDIX E - Installation of GNU CC for Thor.... 155

1 Introduction

Developing a compiler from scratch is unavoidably a project lasting over several years, so a method to reduce this development phase is very desirable. Hence, modifying the GNU CC is one direction that is fruitful to go, since it is freeware and therefore available to the public along with the source code. Moreover the GNU CC designers have been far-sighted enough to make the compiler fairly easy to port to many machines.

1.1 Background

Thor is a 32-bit microprocessor specially designed for space borne, embedded computer systems, to use when tight physical space requirements and fast execution of compiled Ada programs are required, since special hardware support for Ada tasking is added on-chip. Logic for error-detection and correction is also included directly on the chip. The basic idea to shrink the program size is to use a stack-based microprocessor, which requires less code size than comparable register-based processors. Hence, program size is also decreased because each instruction comes in two variants, one short (16-bit) and one long (32-bit). If one can utilize the short ones reasonably frequently, less code space is required.

The development tools existing today include an assembler and an Ada compiler, which will be formally validated during the latter part of 1995. It is likely that potential customers of the Thor concept are interested in a C compiler to use with existing programs while gradually switching to the Ada compiler when writing new programs.

To develop such a C compiler from scratch, or buy a commercial product with the need for many changes, would be unrealistic with the limited production volume of space-borne applications in mind.

A suitable compromise is therefore to make an attempt to port an existing compiler, the GNU CC, to produce code for Thor. GNU CC is so called freeware, which means that everyone can obtain and install the compiler without any fee at all, since even the source files are public. It has been developed during several years by computer idealists (believing all software should be free to copy), primarily within universities. One can make any changes one might wish, including porting it to new machines, with the condition that the new product is also made available to the general public.

GNU CC has a well-defined intermediate language, and is fairly well-documented. The compiler itself is designed to make it possible to do numerous ports with no, or minimal changes to the source code. Since the compiler uses this standardized intermediate language, various front ends have been developed to work with the back end. Languages besides C (GNU CC) supported in this way, include: Ada (GNAT), Modula-2 and C++ (G++). If one succeeds in porting GNU CC to Thor, it means that one gets at least three other compilers automatically.

If the reader is already familiar with the Thor microprocessor and the GNU C Compiler, it is not necessary to read section 2 *'Description of the processor and the compiler'*. This chapter just gives an introduction to the processor and the compiler.

1.2 Definition and goal of the GNU CC-Thor project

The goal of the project is to make the GNU C Compiler deliver trusted code for the Saab Ericsson

Space microprocessor Thor. Primarily, the compiler should work for the C language, and as a secondary goal it should work with an arbitrary front end developed for the compiler, including Ada and utilizing the exception facility and the built-in real-time hardware support, i.e. tasking instructions in the instruction set. Full documentation of the project also has very high priority, since further development of the project is likely to occur.

1.3 References to similar works

GNU CC itself has been ported to numerous machines of various kinds, including now obsolete machines as well as hyper-modern VLIW (Very Long Instruction Word) and superscalar RISC machines. Moreover, a common denominator among these is that they have some sort of register set in their architecture. Even a machine like the iX86 machine, which is in fact an accumulator processor, controls a set of registers, but some of them are indeed dedicated to certain tasks. These machines seem to suite GNU CC rather well, and it really shows that it is a very flexible compiler able to deliver high quality code to an impressive mix of processors.

As far as we know, the GNU CC has never been ported to a stack-based architecture, which means that this project is moving into unexplored territory. This is maybe not so very surprising, since most of the architectures evolved during recent years fall into the register-machine category, so the possibility to develop such a port has been limited.

2 Description of the processor and the compiler

In order to comprehend our implementation of the GNU CC port to the Thor microprocessor, the readers are required to have an insight into the target machine of the compiler as well as the compiler itself. Hence, the following sections will give a brief introduction to both.

2.1 Description of the Thor microprocessor

The Thor microprocessor is a general-purpose, single-chip 32-bit stack-oriented RISC-architecture. The microprocessor is intended for embedded computer systems with high performance requirements in real-time applications, combined with fast execution of programs written in Ada. The main features of the processor include:

- 32-bit RISC architecture with four stage pipeline, and a stack oriented instruction set.
- Ada support with fast rendezvous and interrupt handling.
- Fault tolerance support by concurrent error detection and correction (EDAC).
- Integer and IEEE standard 754 floating-point processing on-chip.

Each of the blocks of Thor will be described in detail in the following sections.

2.1.1 Microprocessor overview

The Thor microprocessor is designed for real-time embedded computers specially made for, and used in space borne systems, where high performance, low power consumption and maximum reliability are desired. The processor is a 32-bit RISC processor, with a stack-oriented architecture. Both integer and IEEE-754 floating-point arithmetic instructions are included in the instruction set.

Thor is developed to provide hardware support for the Ada programming language, in order to offer low interrupt latency and fast task rendezvous. The added hardware speeds up many of the runtime checks defined by the Ada language. Because the instruction set is stack oriented, the memory requirements of the programs are reduced. In the computer systems for which Thor is intended, a smaller memory will often result in significant savings in terms of mass, power consumption and reliability. Error detection facilities are included on-chip, offering fail-stop operation of the microprocessor itself, and the system memory is provided with an EDAC feature.

The design provides support for testing, using a Test Access Port (TAP) implemented according to IEEE-1149.1 standard. Access to the chip pins and the chip internal state is also allowed by the TAP, using a boundary-scan register. Non-intrusive real-time debugging, built-in In Circuit Emulation (ICE) and improved system level verification and testing support are benefits acquired due to the Test Access Port.

Thor uses a memory interface with separate address and data buses, and can access a 2 Gbyte memory. A memory-mapped I/O area occupies the top half of the memory. Each memory read cycle can be completed in one processor clock cycle. A 32-bit word will be transferred each memory access. No virtual memory is supported by the microprocessor, due to the fact that it is not possible to restart instructions in the pipeline.

2.1.2 Architecture and instruction set

The architecture offers a stack-oriented instruction set. The mnemonics, formats and clock cycle requirements are listed in *APPENDIX C - 'Instruction set for Thor'*. Each instruction is either two bytes (short) or four bytes (long). The first byte represents the instruction code, and the following byte (or bytes) is the parameter. The parameter can be either a two's complement signed integer or an unsigned integer. The most significant bit in the instruction code defines if an instruction is long or short, and the next bit indicates whether or not an instruction is signed or unsigned. The operation itself is defined by the remaining six bits. All 256 possible instruction codes are used.

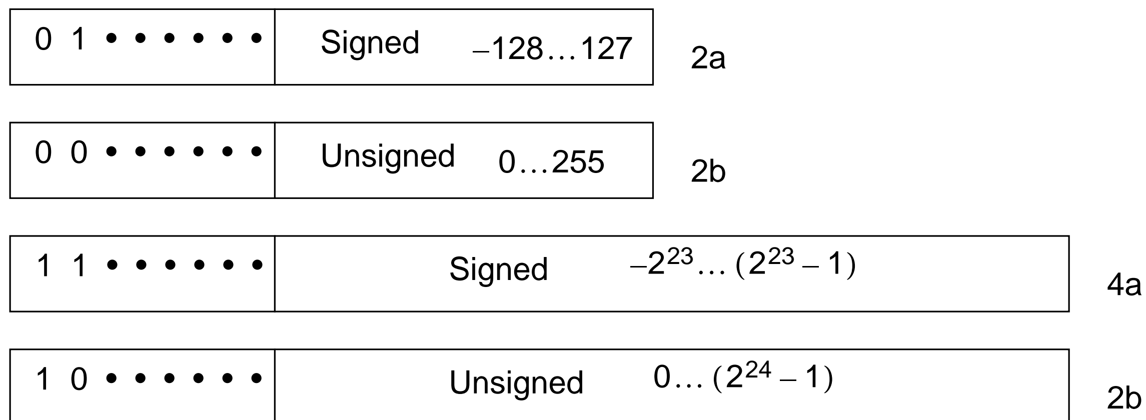


Figure 1 The different formats of Thor's instructions set.

The block diagram below (*see Figure 2, The Thor Chip Block Diagram*) of the Thor chip shows the basic parts of which the processor is built; the pipeline stage, the EDAC facility, the Ada support hardware and the buses.

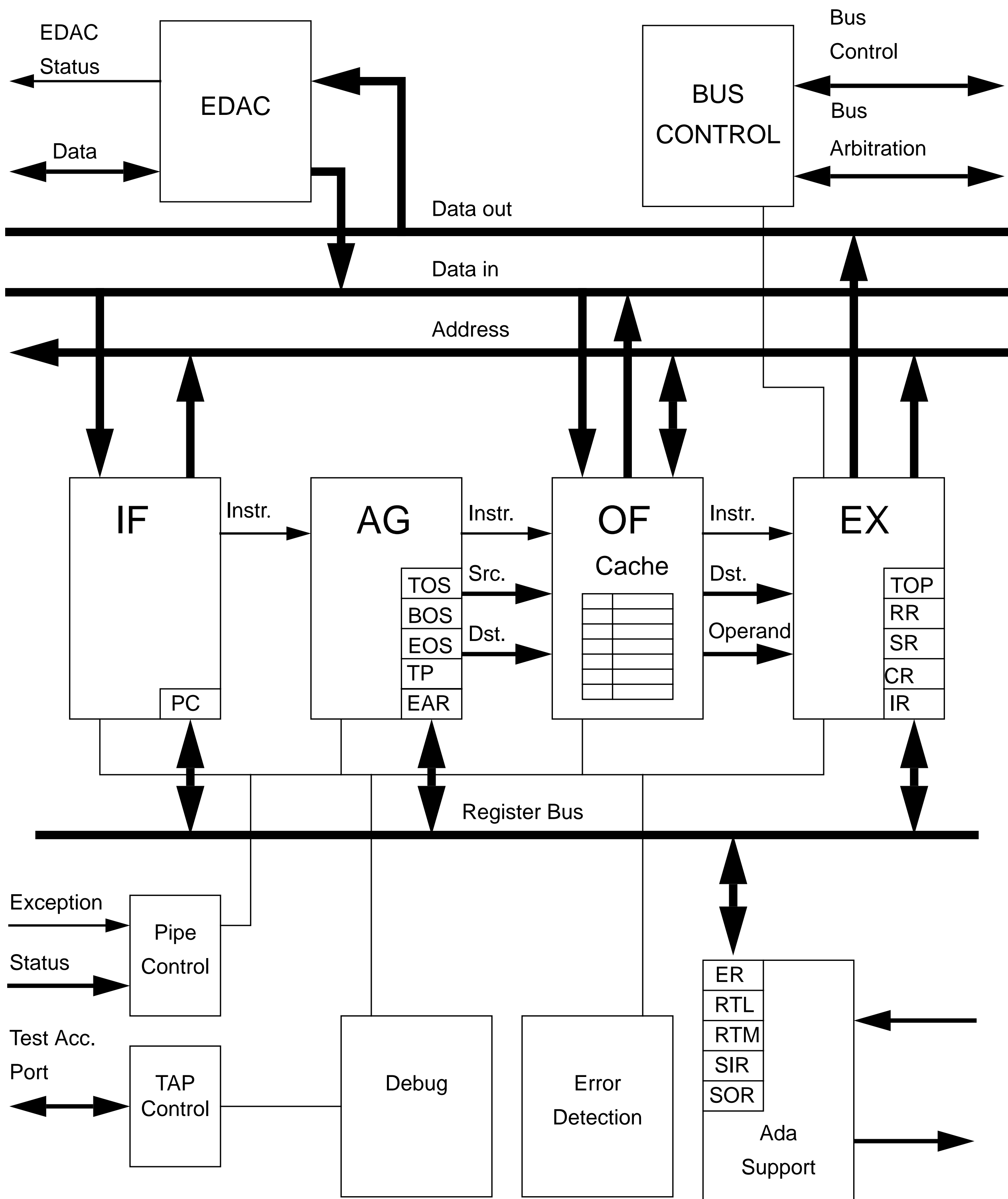


Figure 2 The Thor Chip Block Diagram

2.1.3 The pipeline

The chip's internal instruction pipeline consists of four separate stages:

- Instruction Fetch Stage (IF)
- Address Generation Stage (AG)
- Operand Fetch Stage (OP)
- Execute Stage (EX)

On each clock cycle, an instruction enters the IF stage and proceeds through the pipeline. The pipeline may be stopped, or stalled when some special situations occur. During a stall the information in the pipeline is maintained, until the reason of the stall is solved. A pipeline stall may occur when:

- The IF stage has no instruction ready for the AG stage
- The OF stage is waiting for a read operation of data from memory
- The EX stage is performing a multicycle instruction
- The EX stage is waiting for a I/O write operation of data to memory
- The EX stage is accessing ER, RTL or RTM, while they are updated
- The EX stage is performing a scheduling, when a dispatch is about to occur
- The UM buffer (data cache write-back buffer) is full
- when a HLT instruction is executed

Stalling can also occur during DMA and when assertion of some special instruction is performed.

2.1.3.1 Instruction fetch stage

The instruction fetch stage is equipped with four 32-bit instruction prefetch buffers, in order to prefetch instructions during free bus cycles. The Prefetch Program Counter (PPC) points out the next instruction to be fetched from the memory. The Program Counter (PC) is also included in the IF stage. PC is a halfword counter placed in a register, and the instruction pointed out by PC is located in the IF stage.

The IF stage continuously and cyclically reads instructions into the prefetch buffers, and each time a bus access has been completed the PPC is incremented. No access will be attempted when there is a control transfer instruction in the OF or EX stage and there are two words or more in the prefetch buffers. This behaviour is explained by the fact that there is a possibility that an order to fetch instructions from another place in memory could be issued by the control, and if so, the contents in the prefetch buffer would be obsolete.

The IF stage sends the next instruction and its parameter to the AG stage, as long as the pipeline is not stalled. PC is incremented either by one for short instructions or by two for long instructions. Whenever a control transfer instruction is executed the prefetch buffers are flushed, and both PC and PPC are loaded with the jump target address.

2.1.3.2 Address Generation Stage

The AG stage computes two addresses, the source and the destination address. Some instructions do not have a source address, for instance; immediate instructions. For these instructions the AG stage will pass the instruction parameter, which is the immediate value, as either the source or destination address.

The AG stage contains the following registers:

- Top Of Stack (TOS)
- Beginning Of Stack (BOS)
- End Of Stack (EOS)
- Task pointer

The destination and source addresses can be computed in several ways, depending on the instruction. When a TOS-relative address is needed an offset is added to TOS, or if a PC-relative address is desired an offset is added to the PC, a final variant is adding the TOP register (the value of TOS) with TOS (or some other parameter). Some other possibilities exists but they include computations made with the Task Pointer register (TP) and are therefore only present in Ada programs.

The address computations using the TOP register, will use what is termed indirect delayed addressing, i.e. the indirect address computation is performed in the EX stage, storing the indirect address in TOP. Therefore this computation is compelled to be completed two instructions prior to the instruction using the indirect address (in TOP) as a part of its source or destination address.

In order to check if the computed addresses are located in the legal address space, the AG stage will perform checks on the addresses, using BOS and EOS. The instructions accessing data in the stack will be compared with either BOS or EOS, and an exception will be raised if a violation is detected.

2.1.3.3 Operand Fetch Stage

The operand fetch stage will fetch the instruction operand pointed out by the source address provided by the AG stage (if required by the instruction). When the instruction operates on immediate operands, the immediate value is passed from the AG stage as the source or destination address. Now the operand is passed from the OF stage further on into the EX stage, along with the destination address (which is simply delivered through this stage). If possible, the operands will be fetched from the cache, and if they are not found there they are fetched from the memory. No memory access will take place when the instruction has an immediate operand.

2.1.3.4 Execute Stage

The EX stage executes the instructions, using the ALU[1], the multiplier and the Barrel Shifter, to compute logical and integer as well as floating point data. These units allow most instructions to execute in one clock cycle. The OF stage operand is redundant in cases where the instructions require no operand, and therefore ignored. For instructions needing operands, the EX will use the operand sent from the OF stage, and optionally the contents in TOP. The result produced is gener-

1. ALU stands for Arithmetic Logic Unit. In this part of the processor most computations are made.

ally passed to TOP, but is also written to the destination address. Depending on whether or not the address is found in the cache, it is written to the cache, or to the memory. The EX stage contains the following registers:

- Top Register (TOP)
- Result Register (RR)
- Status Register (SR)
- Configuration Register (CR)
- Identification Register (IR)

The TOP register will always hold the value at the top of the stack. All instructions which pop the stack will fetch the new value of the TOP register. The source address of the AG stage will be the new TOS, and this new value will then be fetched by the OF stage.

The RR register is only used by instructions delivering 64-bit results, to hold the most significant word.

The SR register holds the condition codes which are set by compare and arithmetic instructions, and this register in turn is used by the conditional jump instructions.

To allow setting of parameters to control overall chip behaviour, there is the CR register. Things that the register controls include: Clock frequency, Cache control, Bus timeout and more.

The ID register is provided to uniquely identify the chip, according to the IEEE-1149.1 standard, and includes chip version number, manufacturer, identity and part number.

2.1.4 Pipeline control

When executing normally, the instructions will proceed throughout the pipeline stages, and once in a while cause stalls, as discussed. When the EX stage is executing a control transfer instruction, the program counter in the IF stage will be changed. Now there are already two instructions in the pipe that will be executed before instructions fetched from the new locations arrive to the EX stage. The strategy to solve this problem is to use delayed control transfer. This way no performance loss will occur unless any of the two delay slots cannot be used for other instructions, and therefore must be filled by NOP instructions. There are a number of rare special occurrences where the pipeline must be otherwise controlled, in addition to control transfer.

2.1.5 Data cache

The data cache is designed as a direct mapped, write-back cache, with a size of 32 words. Each line in the cache consists of:

- The actual data word, which is a 32-bit word.
- A 23-bit tag.
- A valid flag, which is set when the data word is up to date.
- A corresponding dirty flag, which indicates when a data word has been modified and write-back has not been performed.

Only the lower half of the memory is cacheable, and therefore one bit in the tag is always set to

zero.

2.1.5.1 The update memory buffer

The UM buffer is a data cache write-back buffer, used when the line is needed for a word at a new address, at the same time as the dirty flag in the line is set. The new word may therefore be directly passed to the desired line, and the word currently occupying the line can be passed to the buffer if its dirty flag is set. The UM buffer will be emptied by writing out the data to the memory. Since the buffer can only hold one word there could be a problem if the buffer is needed and its contents has not been written to memory yet. The solution is to stall the pipe until the data word in the buffer has been flushed to memory.

2.1.5.2 Hazards

The cache works closely together with the pipeline and consequently structural hazards or conflicts may occur. The hazards and interlock situations that are possible includes:

- When data is written into the cache, to the line at which a simultaneous check for another data word is done.
- When the OF stage has read an operand, at the same address the EX stage writes data into the cache. The OF stage now contains an obsolete operand.
- When data that has been transferred to the UM buffer, and a read operation is issued for that data.

The cache controller detects and resolves all of these possible hazard situations by performing various checks using the data being written and bypassing wanted data directly from buffers and pipe stages to the functional unit desiring that data.

2.1.5.3 Snooping

In multi-processor systems and systems using DMA, consistency between data in memory and data in cache must be solved in some way, and Thor uses snooping for this purpose. When an external unit performs a memory read operation, and the cache has the correct version of the data available, the microprocessor will signal this, and then provide the data on the system memory bus. In the case of a write operation, the cache will check whether the data word is in the cache (check if the valid flag is set), and if that is the case the data word will be invalidated.

2.1.6 Error detection

Error detection in space-borne computer systems is of the highest concern, and Thor is equipped with various techniques dealing with this problem.

2.1.6.1 Comparator Function

A computer system with high fault tolerance demands, can be equipped with two Thor microprocessors connected in parallel. One chip is designed to be a slave, and one to be the master. The slave will never drive the system memory bus, but only the master. During the masters execution, the slave will compare its internally produced result with those from the other chip. A high error detection coverage of both chips can be obtained when using this dual configuration.

2.1.6.2 EDAC

The EDAC (Error Detection And Correction) facility works on the system memory data bus, and will detect all two-bit errors and correct all one-bit errors. Seven check bit signals are used in addition to the 32 data bus signals. The EDAC is implemented with a modified Hamming code. Depending on which error the EDAC detects it will assert certain signals, such as CDE (Correctable Data Error) and UDE (Uncorrectable Data Error). Even during DMA and when HLT is asserted, the EDAC will work.

2.1.6.3 Program Flow Control

Program flow control is performed by checksumming the instruction codes, until a NOP instruction is reached. Every NOP instruction will compare the calculated checksum with its parameter, and then eventually reset its checksum. If the parameter and the checksum are not identical, an error signal will be emitted. The program flow checksum is calculated as an arithmetic sum of the instruction codes modulo 2^{24}.

2.1.7 An example of a Thor program

Since the Thor microprocessor is based on a stack architecture, and thus does not contain any general registers, the way of writing assembler programs differs a lot from what most programmers are used to. Instead of loading registers with data and performing operations utilizing the register set, one works using the stack with the use of the data move instructions '`PSH`' and '`POP`'. When performing a '`PSH`' (often stack relative, i.e. with the parameter as an offset from the stack-top) one pushes a new value onto the stack and thus the top-of-stack pointer is decremented by one (the stack grows towards lower addresses). Also, the value at the top-of-stack resides in the '`TOP`'-register.

The following example should give the readers unfamiliar with writing assembler code for a stack machine a first insight to the art of Thor programming (*see APPENDIX C -, Instruction set for Thor*).

```
code    SECT 1,R,C  ;Assembler directive in which sec-
                    ;tion to place following insns.
LC0:                ;A label
    DATAF 6.66e+00  ;One word of constant float data
_main:              ;Label to a function
    MTOS -7         ;Reserve space for 7 words on stack
    PSHI 666        ;Push immediate value '666'
    POP 7           ;Pop it 7 steps back in the stack
    PSH LC0         ;Push float data via reference
    POP 6           ;Pop it 6 steps back in the stack
    PSH 5           ;Push the same value again
    INT             ;Make it an integer
    POP 2           ;Pop it 2 steps back in the stack
***CODE DELETED***
L5:                 ;A label in the code
    PSH 4           ;Push word from 4 steps back
```

```
    ADDI 1          ;Add '1' to pushed word
    MUL 9           ;Multiply with word 9 steps back
    POP 5           ;Pop it back to the stack
***CODE DELETED***
    RET 7           ;Return from the function.
    POP 8           ;'RET' has two delay slots, which
    MTOS 7          ;are filled with stack cleaning
                    ;instr. and popping the result to
                    ;the caller.
```

Example 1 A sample of a Thor program.

To run this program on the Thor processor one must first assemble this text to an object file following the IEEE standard, using the Thor assembler. This code is so called relocatable, i.e. containing no absolute addresses. If the text contains references to external variables one must use the linker in order to deal with unresolved references, and thus merge all necessary object files to one. Finally it is time to use the loader to fill the embedded computer system's memory with the now executable program. The loader converts the relocatable code's relative references to absolute addresses according to the system's address space.

2.2 Description of the GNU C compiler

The main goal of GNU CC, according to GNU's father Richard Stallman, is to make a good and fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary.

GNU CC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. This is a very clean way to describe the target. Unfortunately, the compiler sometimes needs information that is difficult to express in this fashion. The purpose of the portability is to reduce the total work needed on the compiler; it was not of interest for it's own sake.

GNU CC does not contain machine dependent code, but it does contain code that depends on machine parameters such as endianness (whether the most significant byte has the highest or lowest address of the bytes in a word) and the availability of autoincrement addressing. In the RTL-generation pass (Register Transfer Language, the intermediate language used in GNU CC), it is often necessary to have multiple strategies for generating code for a particular kind of syntax tree, strategies that are usable for different combinations of parameters. Often it is hard to address all possible cases, but a sound strategy is to put emphasis on the common ones and make them work as well as possible. As a result, a new target may require additional strategies. One will know if this happens because the compiler will call '`abort`'. Fortunately, the new strategies can be added in a machine-independent fashion, and will affect only the target machines that need them.

What one need to do when trying to make a port to the GNU CC is to write a few files that tells the system and the compiler what the architecture looks like. One file that needs to be written is a header C-file filled with macros that in detail explain the microprocessor's features, such as number of registers and how one is allowed to use them, i.e. if there are certain classes of registers

that one is only supposed to use for certain purposes. Addressing modes allowed on the processor and what kind of addresses accepted are things that are very important to explain in the right way, otherwise the compiler will produce faulty assembler code which uses addressing modes incorrectly. Naming of the macros must be done in the GNU standard way, i.e. you cannot invent a new name for a macro, since those are used in the source code of the compiler and if one macro is missing it is impossible to build the compiler (which is also written in C). Some macros can be omitted in the file, and if so, the compiler uses an appropriate default value. However, there is a well-defined set of macros the compiler cannot do without.

Another file that is necessary to write is the so called '`.md`'-file (machine description file), where, in principle, the machine's instruction set is defined. This file is written in a special syntax that is a textual interpretation of an RTL-object. Each object represents a single (or a sequence of) assembler instruction(s). The GNU CC has a predefined set of mandatory RTL-objects that one must support in the '`.md`'-file, otherwise the compiler will not build. The ideal port would have a one to one mapping between RTL-objects and assembler instructions, to give the compiler more opportunities in optimizing and rescheduling the code. In each RTL-object one defines which addressing modes and registers the object can use when outputting the code. If say, the memory reference, interfer with what the compiler had in mind, it tries to rearrange and transform the code to use legal referencing. RISC machines in particular have a limited way of using data in instructions, since they are load-store architectures and thus want to load data into registers before using them in arithmetic operations.

In the '`.md`'-file one also describes if the machine possesses special facilities such as extra functional units, if so the compiler can schedule the code in such a way that utilization of all functional units is maximized. One can also define which instructions require delay slots. Usually all control transfer instructions fall into this category, i.e. branch, jump and call instructions.

The compiler attempts to translate the whole parsed program into an RTL-representation made exclusively of the mandatory RTL-objects, and if this is not possible the compiler will crash. If the architecture does not contain an assembler instruction as powerful as its corresponding RTL-object, there is a possibility, in the '`.md`'-file, to split the RTL-expression into a sequence of several other, more suitable, objects. This is often the case in modern RISC machines. The most basic compiler optimization one can make is the so called peephole optimization, also defined in the '`.md`'-file.

Additional files are often written, but all they contain are help-functions to the files described above, so there are in fact only two machine dependent files to be written in order to make a port work. This makes it easier to port GNU CC to more machines than any other compiler, since usually one does not need to alter the source code of GNU CC.

2.2.1 Passes

The parsing pass is invoked only once, to parse the entire input. The RTL intermediate code for a function is generated as the function is parsed, one statement at a time. Each statement is read in as a syntax tree and then converted to RTL; then the storage for the tree of statements is reclaimed. Storage for types (and the expressions for their sizes), declarations, and a representation of the binding contours and how they nest, remain until the function is finished compiling; these are all needed to output the debugging information. An informative flow chart below shows

how a compiled program travels through all the different stages in the compiler (it is a bit simplified, for full coverage see the GNU CC documentation).

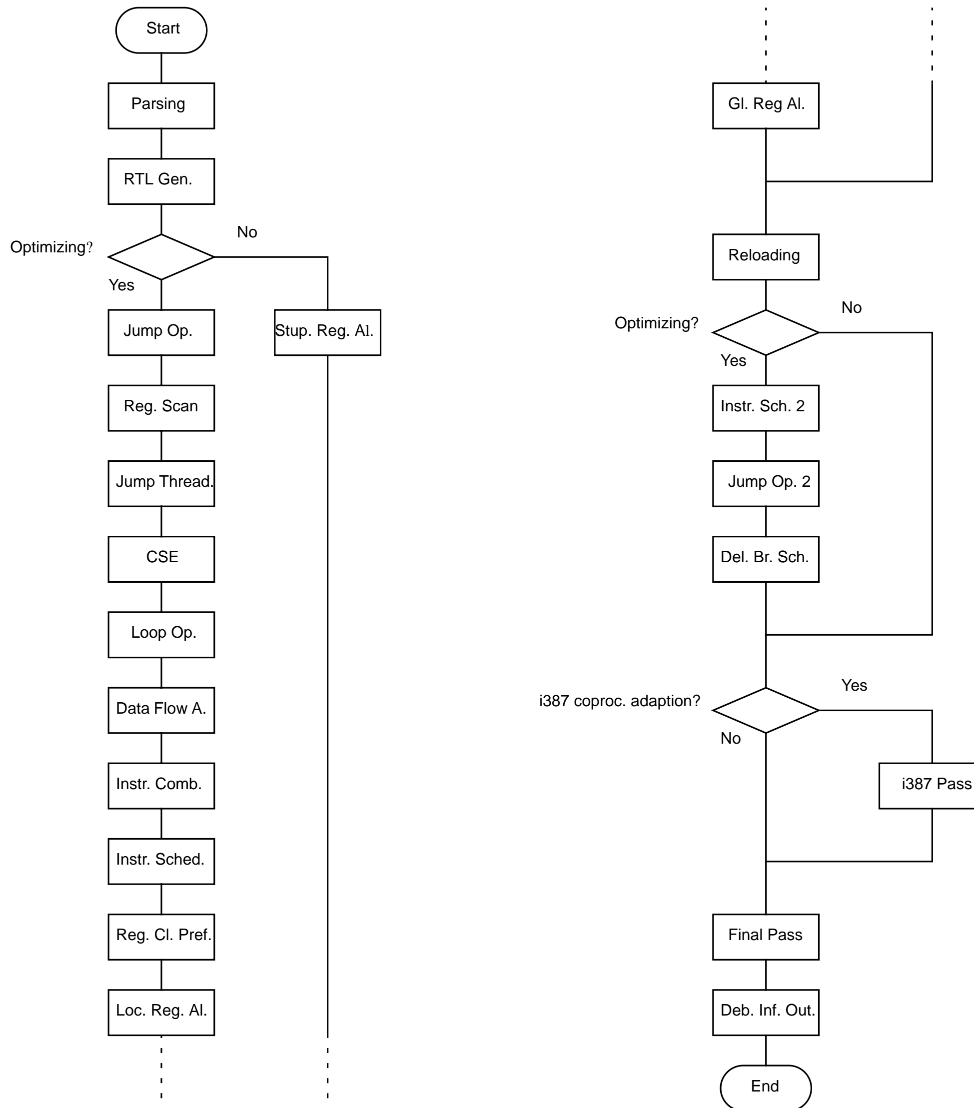


Figure 3 A simplified flow chart of the stages in the compiler.

Each of the following sections deals with one pass of the compiler.

2.2.1.1 Parsing

This pass reads the entire text of a function definition, constructing partial syntax trees. The tree representation does not entirely follow C syntax, because it is intended to support other languages as well. Language-specific data type analysis is also done in this pass, and every tree node that represents an expression has a data type attached. Variables are represented as declaration nodes. Constant folding[1] and some arithmetic simplifications are also done during this pass.

2.2.1.2 RTL generation

This is the conversion of the syntax tree into RTL code. It is actually done statement-by-statement during parsing, but for most purposes it can be thought of as a separate pass. This is where the bulk of target-parameter-dependent code is found, since often it is necessary for strategies to apply only when certain standard kinds of instructions are available. The purpose of named instruction patterns is to provide this information to the RTL generation pass. Optimization is done in this pass for '`if`'-conditions that are comparisons, boolean operations or conditional expressions. Tail recursion is detected at this time also. Decisions are made about how to best arrange loops and how to output 'switch' statements.

Also, the file '`insn-emit.c`', generated from the machine description by the program '`genemit`', is used in this pass. The header files '`insn-flags.h`' and '`insn-codes.h`', generated from the machine description by the programs '`genflags`' and '`gencodes`', tell this pass which standard names are available for use and which patterns correspond to them.

Aside from debugging information output, none of the following passes refers to the tree structure representation of the function (only part of which is saved). The decision of whether the function can and should be expanded inline in its subsequent callers is made at the end of RTL generation. The function must meet certain criteria, currently related to the size of the function and the types and number of parameters it has. Note that the function may contain loops, recursive calls to itself (tail-recursive functions can be inlined), gotos, in short, all constructs supported by GNU CC.

2.2.1.3 Jump optimization

This pass simplifies jumps to the following instruction, jumps across jumps, and jumps to jumps. It deletes unreferenced labels and unreachable code, except that unreachable code that contains a loop is not recognized as unreachable in this pass. (Such loops are deleted later in the basic block analysis.) It also converts some code originally written with jumps into sequences of instructions that directly set values from the results of comparisons, if the machine has such instructions. Jump optimization is performed two or three times. The first time is immediately following RTL generation. The second time is after CSE (*see 2.2.1.6, Common subexpression elimination (CSE)*), but only if CSE says repeated jump optimization is needed. The last time is right before the final pass. That time, cross-jumping and deletion of no-op move instructions are done together with the optimizations described above.

2.2.1.4 Register scan

This pass finds the first and last use of each register, as a guide for common subexpression elimination.

2.2.1.5 Jump threading

This pass detects a conditional jump that branches to an identical or inverse test. Such jumps can be 'threaded' through the second conditional test.

2.2.1.6 Common subexpression elimination (CSE)

CSE means that the compiler scans through the program in search of duplicate expressions, and if

1. Constant folding means that constants defined in the program will be inserted where they are used in the program.

the value of an expression is known earlier in the program it is thus not necessary to compute the expression once more. This pass also does constant propagation. If constant propagation causes conditional jumps to become unconditional or to become no-ops, jump optimization is run again when CSE is finished.

2.2.1.7 Loop optimization

This pass moves constant expressions out of loops, and optionally does strength reduction[1] and loop unrolling[2] as well.

If flag '`-frerun-cse-after-loop`' was enabled when compiling, a second common subexpression elimination pass is performed after the loop optimization pass. Jump threading is also done again at this time if it was specified.

2.2.1.8 Stupid Register Allocation

Stupid register allocation is performed at this point in a nonoptimizing compilation. It does a little data flow analysis as well. When stupid register allocation is in use, the next pass executed is the reloading pass; the others in between are skipped. This pass will execute faster than compilation in optimized mode.

2.2.1.9 Data flow analysis

This pass divides the program into basic blocks[3] (and in the process deletes unreachable code); then it computes which pseudo-registers are live at each point in the program, and makes the first instruction that uses a value point at the instruction that computed the value. This pass also deletes computations whose results are never used, and combines memory references with add or subtract instructions to make autoincrement or autodecrement addressing.

2.2.1.10 Instruction combination

This pass attempts to combine groups of two or three instructions that are related by data flow into single instructions. It combines the RTL expressions for the instructions by substitution, simplifies the result using algebra, and then attempts to match the result against the machine description.

2.2.1.11 Instruction scheduling

This pass looks for instructions whose output will not be available by the time that it is used in subsequent instructions. (Memory loads and floating point instructions often have this behaviour on RISC machines). It re-orders instructions within a basic block to try to separate the definition and use of items that otherwise would cause pipeline stalls. Instruction scheduling is performed twice. The first time is immediately after instruction combination and the second is immediately after reload.

---

1. Strength reduction means manipulation with the arithmetic operations while maintaining the correct semantics of the program. For example, converting a multiplication to an addition.
2. If a loop in a program is deterministic in terms of knowing in advance how many times a loop will execute, the loop can be unrolled, i.e. the jump instructions are omitted and all the instructions in the loop are repeated the same number of times the loop is to be executed.
3. A basic block is a part of a program where there exists no jump into or out of the sequence except the beginning and the end of the instruction sequence.

2.2.1.12 Register class preferencing

The RTL code is scanned to find out which register class is best for each pseudo register.

2.2.1.13 Local register allocation

This pass allocates hard registers to pseudo registers that are used only within one basic block. Because the basic block is linear, it can use fast and powerful techniques to do a very good job.

2.2.1.14 Global register allocation

This pass allocates hard registers for the remaining pseudo registers (those whose life spans are not contained in one basic block).

2.2.1.15 Reloading

This pass renumbers pseudo registers with the hardware registers numbers they were allocated. Pseudo registers that did not get hard registers are replaced with stack slots. Then it finds instructions that are invalid because a value has failed to end up in a register, or has ended up in a register of the wrong kind. It fixes up these instructions by generating code to reload the problematical values temporarily into registers. Additional instructions are generated to do the copying. The reload pass also optionally eliminates the frame pointer and inserts instructions to save and restore call-clobbered registers around calls.

2.2.1.16 Instruction Scheduling, second pass

Instruction scheduling is repeated here to try to avoid pipeline stalls due to memory loads generated for spilled pseudo registers.

2.2.1.17 Jump optimization, second pass

Jump optimization is repeated, this time including cross jumping[1] and deletion of no-operation move instructions.

2.2.1.18 Delayed branch scheduling

This pass attempts to find instructions that can go into the delay slots of other instructions, usually jumps and calls.

2.2.1.19 Intel 80387 special pass

Conversion from usage of some hard registers to usage of a register stack may be done at this point. Currently, this is supported only for the floating-point registers of the Intel 80387 coprocessor.

2.2.1.20 Final

This pass outputs the assembler code for the function. It is also responsible for identifying and removing unnecessary test and compare instructions. Machine-specific peephole optimizations are performed at the same time. The function entry and exit sequences are generated directly as assembler code in this pass; they never exist as RTL.

---

1. Cross jumping means detecting identical sequences of instructions followed by jumps to the same place, or followed by a label and a jump to that label, and replacing one with a jump to the other.

2.2.1.21 Debugging information output

This is run after final because it must output the stack slot offsets for pseudo registers that did not get hard registers.

2.2.2 Running the compiler

GNU CC normally does preprocessing, compilation, assembling and linking, when invoking it with command 'gcc'[1] followed by a filename. There are numerous compiler options with which one is able to control and steer each stage in the compilation process in detail. For example one often wants to stop the compilation after the assembler has run, in order to get a relocatable object file which one can, in a later stage, link together with other object files. Some options control the preprocessor and others the compiler itself. You can mix options and other arguments. For the most part, the order you use does not matter. Order does matter when you use several options of the same kind. The following subchapters will give a brief introduction to the most useful options when running the compiler.

There are lots of options not mentioned in the following sections, dealing with the assembler, linker, how rigorous the compiler should be about the syntax in the programs etc. Those few of the readers who need all these options should take a look in the GNU CC complete documentation.

2.2.2.1 Overall options

This section describes the options controlling the output, i.e. if one wants an executable, object, assembler or preprocessed source. For any given input file, the compiler assumes certain things about what kind of compilation to do depending on the suffix. Below follows a table with the GNU CC's way of interpreting the suffixes.

Table 1 Action depending on suffix

Suffix	Action to be taken
'.c'	C source file that must be preprocessed.
'.i'	C source file which should not be preprocessed.
'.h'	C header file (not to be preprocessed or compiled).
'.m'	Objective C code.
'.ii'	C++ source file which should not be preprocessed.
'.cc' '.C' '.cpp'	C++ source file that must be preprocessed. Suffix '.cc' is the preferred one to use.
'.s'	Assembler code.
'.S'	Assembler code which should be preprocessed.
OTHER	An object file to be fed directly into linking

1. The GNU CC cross compiler for Thor is invoked with the command 'thor-gcc'.

If one want to access only some stages of the compiler one can alternatively use flags to be inserted on the command line. Here follows a table with the most important options.

Table 2 Options to control the compilations

Option	Action to be taken
'-c'	Compile and assemble the source files, but do not link. An object file for each source file given is the ultimate output.
'-S'	This option inhibits the assembly part of the compilation. An assembler code file is the output for each non-assembler input file. The assembler file name are made, by replacing the files suffixes by '.s' (default behaviour).
'-E'	Stop after the preprocessing stage. The output is in the form of preprocessed source code, which is sent to the standard output.
'-o FILE'	Place output in file 'FILE'. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. Unless one are producing an executable file, it makes no sense applying this option on more than one file. The default name of the executable file when not using this option is 'a.out'.
'-v'	For each stage of the compilation, print on standard error output the commands executed. This includes various version numbers of the compiler, preprocessor etc.

2.2.2.2 Debugging options

GNU CC permits various special options that are used for debugging either your program or GNU CC itself. The following table lists the most important debugging features in the compiler.

The options starting with a '-d' followed by a single additional letter tells the compiler to make debugging dumps during compilation at times specified by the last letter in the option command. Primarily this is used when debugging the compiler. The debugging dump file's name is constructed by appending an extra suffix to the name of the file. Those options do not affect the compilation in any way, except that the compilation may run a little bit slower when outputting to the files.

Table 3 Debugging options to be incorporated when compiling

Debug option	Action to be taken
'-g'	Produce debugging information in the operating system's native format.
'-p'	Writing extra profile information suitable for the analysis program 'prof'.
'-pg'	Writing extra profile information suitable for the analysis program 'gprof'.
'-a'	Generate additional code to write profile information about basic blocks. Essentially it will count the number of times each basic block is entered.

Table 3 Debugging options to be incorporated when compiling

Debug option	Action to be taken
'-dM'	Dump all macro definitions, at the end of preprocessing, and write no output.
'-dN'	Dump all macro names, at the end of preprocessing.
'-dD'	Dump all macro definitions, at the end of preprocessing, in addition to normal output.
'-dy'	Dump debugging information during parsing, to standard error.
'-dr'	Dump after RTL generation, to 'FILE.rtl'.
'-dx'	Just generate RTL for a function instead of compiling it.
'-dj'	Dump after first jump optimization, to 'FILE.jump'.
'-ds'	Dump after CSE, to 'FILE.cse'.
'-dL'	Dump after loop optimization, to 'FILE.loop'.
'-dt'	Dump after the second CSE pass, to 'FILE.cse2'.
'-df'	Dump after flow analysis, to 'FILE.flow'.
'-dc'	Dump after instruction combination, to the file 'FILE.combine'.
'-dS'	Dump after the first instruction scheduling pass, to 'FILE.sched'.
'-dl'	Dump after local register allocation, to 'FILE.lreg'.
'-dg'	Dump after global register allocation, to 'FILE.greg'.
'-dR'	Dump after the second instruction scheduling pass, to 'FILE.sched2'.
'-dJ'	Dump after last jump optimization, to 'FILE.jump2'.
'-dd'	Dump after delayed branch scheduling, to 'FILE.dbr'.
'-dk'	Dump after conversion from registers to stack, to 'FILE.stack'.
'-da'	Produce all the dump files listed above.
'-dm'	Print statistics on memory usage, at the end of the run, to standard error.
'-dp'	Annotate the assembler output with a comment indicating which pattern and alternative was used.

Not all of the debugging options are listed in the table, but those remaining are very special and virtually never used in reality. Those interested in a full coverage should read the GNU CC documentation.

2.2.2.3 Optimizing options

The most common options when compiling are the optimizing options; '-OX' where 'X' stands for a number between 0 and 3. With the '-O0' option one explicitly says that *no* optimizing should occur (this is equivalent to giving no '-OX' option at all), and with all the other options

optimizations of various degrees are performed. Depending on the option given, the compiler performs rescheduling and transformations to make the code run faster (fast code is usually prioritized over small code size). The most apparent optimizations are rescheduling of jump instructions, inlining (only performed in '-O3' option), common subexpression elimination, jump threading and delayed branch scheduling. The machine-dependent peephole transformations are activated in all the options. What the '-OX' options do in reality are to activate certain flags in the compiler which specifically affects how the compilation proceeds. The user have direct access to all of those flags and can explicitly activate each and every one of them when starting a compilation, but usually it is enough for most users to use the '-OX' options. Only if a specific behaviour is desired to be encouraged or suppressed, the flags might come in handy. See the GNU CC documentation for a complete survey of the flags.

2.2.3 Intermediate representation

Most of the work of the compiler is done on an intermediate representation called RTL (Register Transfer Language). In this language, the instructions to output are described, pretty much one by one, in an algebraic form that describes what the instruction does. RTL is inspired by Lisp[1] lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form. Here follows an example from a '.rtl' dump file which hopefully will enlighten the readers. It shows the textual interpretation in a Lisp fashion, as described.

```
(insn 11 10 14
      (set (mem:QI (plus:QI (reg:QI 5)
                            (const_int -1)))
           (reg:QI 1 TMP)) -1 (nil)
    (nil))
```

Example 2 An extract of a RTL dump file when compiling for Thor

In *Example 2* we see a very simple example of what an RTL-object might look like. The numbers following the string 'insn' indicates which place in the intermediate language list this object occupies (11) and to which objects it is linked to (10 and 14). The next sequence of characters show the main syntax of the RTL-object. In this case it simply describes moving the contents of register 1 ('TMP') to a memory location. The address of the memory location equals the contents of register 5 minus 1. In other words one can say that this RTL-object represents a store operation from a register to a memory location.

The register numbers in *Example 2* are hard registers but normally one should not interpret these as hard registers but merely as pseudo registers, that in a later stage in the compiler will be mapped to the processor's hard registers. The characters '-1' before the 'nil'-lists tell that the compiler has not tried to match this RTL-object with an instruction defined in the machine description file. When matching, the compiler might need to transform the RTL-object in order to suite the defined instructions.

1. Lisp (List Processor) is a functional programming language.

RTL uses five kinds of objects: expressions, integers, wide integers, strings and vectors. Expressions are the most important ones. An RTL expression (“RTX”, for short) is a C structure, but it is usually referred to with a pointer. In the example above, the whole ’`set`’ sequence is an expression and therefore an RTX, and in turn it may contain other RTXs, like ’`const_int -1`’ in this case.

Every time a reference to memory or a register occurs in RTL code, one must specify in which machine mode the operation works. The most common machine modes are word, half-word and byte mode. In the example above one can see that each reference to a data location, is working in ’`QI`’-mode (indicated by the character sequence ’`:QI`’ after a ’`mem`’ or ’`reg`’ reference). ’`QI`’ stands for Quarter Integer, which means a quarter of a word (usually 8 bits, but this is not true for Thor, *see 3.3.2, Target control macros*). There are also an ’`SI`’- and an ’`HI`’-mode.

2.2.4 Machine Description

A machine description has two parts: one file containing instruction patterns (the ’`.md`’-file) and a C header file with macro definitions. The ’`.md`’-file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about). The header file defines numerous macros that convey information about the target machine that does not fit into the scheme of the ’`.md`’-file.

What really happens when building a compiler is that the ’`.md`’-file is interpreted by several programs. These programs generate C source files, which in turn are compiled and linked with the rest of the GNU CC source files. Further on, the ’`.h`’-macro file is included in the building, and the macro definitions are replaced in ordinary C preprocessor fashion. The help-functions in the ’`.c`’-file are compiled and linked as an ordinary module. This way of designing a compiler has several advantages:

- One can make a target independent compiler, which can be placed on several different platforms, compiling to an arbitrary target.
- It is not necessary to change any of the GNU CC source files, since all target dependent features are solved by the use of macros, and by the instruction patterns in the ’`.md`’-file.

2.2.4.1 The Machine Description file, ’`.md`’-file

This file mainly contains the instruction patterns, which defines the set of standard names the compiler is allowed to use, and also shows how the assembler instructions are output. An instruction pattern is either a ’`define_insn`’ or a ’`define_expand`’-pattern. In addition to the instruction patterns, the file also contains patterns for peephole optimization. To get an understanding of how things work, one can also take a look at *Figure 10 ’Example of RTL generation’*.

There is an important difference between ’`define_expand`’- and ’`define_insn`’-patterns. The ’`define_expand`’-expressions are only used in the RTL generations stage, used by the compiler when trying to convert the parse tree to the intermediate language, therefore one uses these constructs when a sequence of ’`define_insn`’-expressions are the only way of handling a certain task in an architecture’s instruction set. Their only task consists of showing the compiler how a specific operation is best represented in the machine. The compiler has access to a set of standard names which one can use in order to give a hint to the compiler that the machine can

solve a certain mission directly. These standard names can either be of the 'define_expand' or the 'define_insn' variant, where the 'define_expand' expression just translates the construct to several 'define_insn' expressions. In the '.md'-file so called nameless patters could be found, these are just like the standard-name expressions but the compiler does not have any prejudices about them, often one use them as collectors for the debris resulting from the ravages of the 'define_expand'-expressions.

Two very important fields in the templates when generating the RTL code in terms of deciding which pattern to select, are the predicate and the constraint field. Against those the generated RTL code is matched to decide which template to select. Moreover, if the predicate accepted the operand, the template is suitable. Next, the constraints are checked, and that field is a list of arbitrary length containing different cases of data references that should be separated from each other. Each constraint either directly corresponds to an assembler string, or a C function manipulating a more complex operand and in turn putting out the assembler code after modifying the memory reference in a suitable way. If no constraint is matched the compiler tries to reload the failing operand, i.e. a very complex operand that the patterns cannot handle is computed in steps by generating several simpler RTLs. The output template field can either contain as many strings as there are constraints, or a C block that in turn is able to modify and alter the operands before it outputs the assembler code. The block is able to call C help functions that one can define in the '.c'-file.

There exists a third field that can control if the template is the right one to choose. The field can contain an arbitrary C expression, that must evaluate to true or false. This is called the condition field and often is it left blank, since one can usually solve the RTL generation problem without it.

2.2.4.1.1 The 'define_insn' patterns

Each normal instruction pattern contains an incomplete RTL expression, with pieces to be filled in later, operand constraints that restrict how the pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a 'define_insn' expression. A 'define_insn' is an RTL expression containing four or five operands:

1) An optional name. The presence of a name indicate that this instruction pattern can perform a certain standard job for the RTL-generation pass of the compiler. This pass knows certain names and will use the instruction patterns with those names, if the names are defined in the machine description. The absence of a name is indicated by writing an empty string where the name should go. Such nameless instruction patterns are never used for generating RTL code, but they may permit several simpler instructions to be combined later on. Names that are not thus known and used in RTL-generation have no effect; they are equivalent to no name at all. Giving a nameless pattern a name could be a help when studying RTL dumps, since the name gives a hint what the pattern actually does. The standard convention, used by most ports, is to give the nameless patterns names starting with the character '*' followed by an appropriate name implying what kind of work the pattern does. A pattern named in this way will still be treated as a nameless one by the compiler.

2) The “RTL template”. This is a vector of incomplete RTL expressions

which shows what the instruction should look like. It is incomplete because it may contain 'match_operand', 'match_operator', and 'match_dup' expressions that stand for operands of the instruction. If the vector has only one element, that element is the template for the instruction pattern. If the vector has multiple elements, then the instruction pattern is a 'parallel'-expression containing the elements described.

3) A "condition". This is a string which contains a C expression that is the final test to decide whether an instruction body matches this pattern. For a named pattern, the condition (if present) may not depend on the data in the instruction being matched, but only on the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run.

4) The "output template": a string that says how to output matching instructions as assembler code. '%' in this string specifies where to substitute the value of an operand. When simple substitution is not general enough, a piece of C code can be specified to compute the output.

5) Optionally, a vector containing the values of attributes for instructions matching this pattern.

Here is an actual example of an instruction pattern for Thor:

```
(define_insn "*addqi"
  [(set (match_operand:QI 0
                    "tmp_register_operand" "=r,r,r")
    (plus:QI (match_operand:QI 1
                    "tmp_register_operand" "0,0,0")
         (match_operand:QI 2
                    "simple_operand" "I,m,r")))]
  ""
  "@
   ADDI %2
   ADD %z2
   ADD %z2"
[(set_attr "type" "arith,arith,arith")])
```

Example 3 An example of a Thor instruction pattern

This example shows what an add instruction in 'QI'-mode looks like. Observe the difference between an instruction pattern and an RTL-object. Here we only specify that some operand 0 should store the result from the addition between operand 1 and 2. Nothing is said about the register numbers and memory positions. The pattern shown is nameless (the name begins with '*') and used to match the RTL generated by a 'define_expand'-pattern.

The string 'tmp_register_operand' is a so called predicate, and in this case (for operand 0) it forces the operand to be a register (and only the 'TMP'-register defined according to Thor's machine description, *see 3.3.2, Target control macros*), and if not so, the compiler will not be able

to match this pattern with the generated RTL-object. If the compiler cannot find a suitable pattern it will crash, and therefore one must make sure that every possible case that can arise from the RTL generation must have an instruction template. The predicate may be an empty string; then it means that no test is to be done on the operand, so anything which occurs in this position is valid.The predicate is usually a pure C function taking two arguments, the machine mode and the operand, and returning true or false. One can have numerous self-defined predicates, apart from the standard ones defined by GNU CC, which one usually places in the '`.c`'-file.

The string following the predicate is called a constraint. Each '`match_operand`' in an instruction pattern can specify a constraint for the type of operands allowed. Constraints can say whether an operand may be in a register, and which kinds of registers; whether the operand can be a memory reference, and which kinds of addresses; whether the operand may be an immediate constant, and which possible values it may have. The constraints should be a subset of its corresponding predicate, and ought not to allow more possibilities than the predicate. Often the compiler succeeds in finding an allowed pattern but the RTL-object does not satisfy its constraints, in this case the compiler tries to transform the program in order to make possible for a certain operand to fulfil a specific constraint, and this without changing the semantics. This can often be arranged by inserting load instructions to and from registers together with the failing instruction, i.e. reloading. Constraints can also require two operands to match.

2.2.4.1.2 Expander Definitions

On some target machines, some standard pattern names for RTL generation cannot be handled with single instructions, but a sequence of RTL instructions can represent them. For these target machines, a '`define_expand`' pattern can be written to specify how to generate the sequence of RTL. A '`define_expand`' is an RTL expression that looks almost like a '`define_insn`'; but, unlike the latter, a '`define_expand`' is used only for RTL generation and it can produce more than one RTL instruction. A '`define_expand`' RTX has four operands:

1) The name. Each '`define_expand`' must have one of the standard names, since the only use for it is to refer to it by name.

2) The RTL template. This is just like the RTL template for a '`define_peephole`' in that it is a vector of RTL expressions each being one instruction.

3) The condition, a string containing a C expression. This expression is used to express how the availability of this pattern depends on subclasses of target machine, selected by command-line options when GNU CC is run. This is just like the condition of a '`define_insn`' that has a standard name. Therefore, the condition (if present) may not depend on the data in the instruction being matched, but only on the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run.

4) The preparation statements, a string containing zero or more C statements which are to be executed before RTL code is generated from the RTL template. Usually these statements prepare temporary registers for use as internal operands in the RTL template, but they can also generate RTL

instructions directly by calling routines such as 'emit_insn', etc. Any such instructions precede the ones that come from the RTL template.

Every RTL instruction emitted by a 'define_expand' must match some 'define_insn' in the machine description. Otherwise, the compiler will crash when trying to generate code for the instruction or trying to optimize it. The 'define_insn' that will in turn be matched against must not be a standard-name pattern, its (optional) name will not be checked at all. Here is a example how it is possible to use this facility:

```
(define_expand "movqi"
  [(set (match_operand:QI 0 "general_operand" "")
    (match_operand:QI 1 "general_operand" ""))]
  ""
  "
{
  if (emit_move_sequence (operands, QImode))
    DONE;
}")
```

Example 4 One way to use the 'define_expand'-feature (from 'thor.md')

The compiler will match this 'define_expand'-expression whenever it wants to move a quarterword ('QI'-mode means a full word in Thor's machine description, *see 3.3.2, Target control macros*) from an arbitrary location to another, since the predicate says 'general_operand' (register, memory etc.). The last field in the expand-expression contain a C-block, which essentially will execute a function 'emit_move_sequence' with two arguments, where 'operands' is a array containing all the operands in the RTX-field, and the second argument is a machine mode. The function will in turn emit a sequence of patterns that will be matched against 'define_insn'-expressions. A macro 'DONE' is inserted after the function call since we do not desire that the template itself should be emitted, just the sequence generated by the function.

2.2.4.1.3 Standard Names

The standard names are a set of predefined names that have a special meaning in the RTL-generation pass of the compiler. Giving one of these names to an instruction pattern tells the RTL generation pass that it can use the pattern to accomplish a certain task. Some names are essential, and must be found in the '.md'-file, otherwise the compiler will not build. Other names could be left out and the compiler will find another way to solve its mission. But if one finds a clever machine-dependent way of accomplishing the task, it would help the compiler to deliver the most optimized code possible.

2.2.4.1.4 Machine-Specific Peephole Optimizers

In addition to instruction patterns the '.md'-file may contain definitions of machine-specific peephole optimizations. The combiner does not notice certain peephole optimizations when the data flow in the program does not suggest that it should try them. For example, sometimes two consecutive instructions related in purpose can be combined even though the second one does not appear to use a register computed in the first one. A machine-specific peephole optimizer can detect such opportunities. A definition looks like this:

```
(define_peephole
    [INSN-PATTERN-1
     INSN-PATTERN-2
    ...]
    "CONDITION"
    "TEMPLATE"
"OPTIONAL INSN-ATTRIBUTES")
```

Example 5 A template of a peephole definition

In this skeleton, 'INSN-PATTERN-1' and so on are patterns to match consecutive instructions. The optimization applies to a sequence of instructions when 'INSN-PATTERN-1' matches the first one, 'INSN-PATTERN-2' matches the next, and so on. Each of the instructions matched by a peephole must also match a 'define_insn'.

Peepholes are checked only at the last stage just before code generation, and only optionally. Therefore, any instruction which would match a peephole but no 'define_insn' will cause a crash in code generation in an unoptimized compilation, or at various optimization stages. 'CONDITION' is a C expression which makes the final decision whether to perform the optimization (if the expression is nonzero). If 'CONDITION' is omitted (in other words, the string is empty) then the optimization is applied to every sequence of instructions that matches the patterns. Applying the optimization means replacing the sequence of instructions with one new instruction.

The 'TEMPLATE' controls ultimate output of assembler code for this combined instruction. It works exactly like the template of a 'define_insn'. Operand numbers in this template are the same ones used in matching the original sequence of instructions.

The result of a defined peephole optimizer does not need to match any of the instruction patterns in the machine description; it does not even have an opportunity to match them. The peephole optimizer definition itself serves as the instruction pattern to control how the instruction is output.

2.2.4.1.5 Instruction attributes

In addition to describing the instruction supported by the target machine, the '.md'-file also defines a group of "attributes" and a set of values for each. Every generated instruction is assigned a value for each attribute. The 'define_attr'-expression is used to define each attribute required by the target machine. It looks like:

```
(define_attr NAME LIST-OF-VALUES DEFAULT)
```

Example 6 How an attribute is defined

'NAME' is a string specifying the name of the attribute being defined. 'LIST-OF-VALUES' is either a string that specifies a comma-separated list of the values that can be assigned to this attribute, or a null string to indicate that the attribute takes numeric values. 'DEFAULT' is an attribute expression that gives the value of this attribute for instructions that match patterns whose definition does not include an explicit value for this attribute.

2.2.4.1.6 Delay slot attributes

The instruction attribute mechanism can be used to specify the requirements for delay slots, if

any, on a target machine. An instruction is said to require a delay slot if some instructions that are physically after the instruction are executed as if they were located before it. Classic examples are branch and call instructions, which often execute the following instruction before the branch or call is performed. On some machines, conditional branch instructions can optionally annul instructions in the delay slot. This means that the instruction will not be executed for certain branch outcomes. Delay slot scheduling differs from instruction scheduling in that determining whether or not an instruction needs a delay slot, is dependent only on the type of instruction being generated, not on data flow between the instructions.

The requirement of an instruction needing one or more delay slots is indicated via the 'define_delay'-expression. It has the following form:

```
(define_delay TEST
    [DELAY-1 ANNUL-TRUE-1 ANNUL-FALSE-1
     DELAY-2 ANNUL-TRUE-2 ANNUL-FALSE-2
        ...])
```

Example 7 Template of a 'define_delay'-attribute

'TEST' is an attribute test that indicates whether this 'define_delay' applies to a particular instruction. If so, the number of required delay slots is determined by the length of the vector specified as the second argument. An instruction placed in delay slot N must satisfy attribute test 'DELAY-N'. 'ANNUL-TRUE-N' is an attribute test that specifies which instructions may be annulled if the branch is true. Similarly, 'ANNUL-FALSE-N' specifies which instructions in the delay slot may be annulled if the branch is false. If annulling is not supported for that delay slot, '(nil)' should be coded.

2.2.4.2 Target Description Macros, '.h'-file

In addition to the '.md'-file, a machine description includes a C header file conventionally given the name '.h'-file. This header file defines numerous macros that convey the information about the target machine that does not fit into the scheme of the '.md'-file. Typical things are; various commands to control the compilation and assembling type and storage layout, i.e. how data should be aligned, what kind of registers the processor controls and how to use them, addressing modes, i.e. what sort of addresses are legal and in turn could be handled by the machine, what happens with the stack at various situations and how to master it.

Depending on which macro we are concerned about, the layout can be totally different, and often is it necessary to study the GNU CC documentation very carefully and sometimes even look in the source files in order to completely understand how the macro should be designed and used.

Since they are macros, any type control vanishes if the macro takes any arguments, and therefore extra concern should be taken when using the macros as functions (as always). And one ought not to forget that since macros means textual replacement, the precedence between operators and variables might be different compared to the C language, and extra care should be taken in this matter too.

GNU CC source files use the numerous macros defined in the '.h'-file. This way one is able to build several compilers for different machines, without changing the source files. One drawback

is that the source files may be a little hard to read, but there is maybe no other way to solve this problem. No example of any macros will be given here because, as explained, they would be too machine specific since they concern register sets, addressing modes etc. A example would therefore require immense amount of explanation about this particular machine. See documentation of the GNU CC, for full coverage of all macros and what they stand for, and the '`.h`'-file for Thor to see how we defined them.

3 GNU C Compiler for the Thor microprocessor

In this chapter we focus on our port of the GNU C compiler to the Thor microprocessor. It will mainly be a description of the resulting compiler, but we begin with our problems and our strategy. We will also try to estimate how well we succeeded in porting the GNU C compiler.

3.1 Problems

There are several things which must be done when porting the GNU CC:

- The Thor processor is not a known target for the designers of the GNU CC, and therefore we must make it possible to install and configure a compiler for yet another target.
- Since Thor does not have an operating system, and is strictly intended for embedded systems, one cannot install GNU CC on a computer system equipped with Thor. Therefore, one is forced to install the GNU CC as a cross-compiler working on a host (in our case a Sun Sparc 20 workstation), delivering code for Thor. Subtle problems can arise when making a cross-compiler if, for example, the host and the target do not have identical floating-point formats.
- There is no way to make GNU CC directly deliver hex-code for Thor, so we must use a separate assembler that assembles the textual output from the compiler. The output must be in the form of assembler instructions using the mnemonics listed in the appendix. Furthermore we must configure the compiler to utilize the Thor assembler instead of the usual unix assembler '`as`'. The code GNU CC delivers must then suit the assembler so that no illegal constructs are generated, which are not tolerated by the assembler. This also includes the hardware constraints that arise, for example when pushing a number too large to fit in an immediate instruction, the number must be put in a data word in memory and then pushed via a symbolic reference.

The problems above are quite ordinary ones and everyone who tries porting the GNU CC as a cross-compiler will meet them. But we soon encountered other problems, which are a bit more tricky:

- In the definition of the GNU CC's goals, one can read that the compiler is designed for 32-bit machines that have several general registers. Thor is indeed a 32-bit machine but is fails on the register claim, since Thor does not contain any registers except specialized registers dedicated to special tasks, such as the top-of-stack pointer and program counter.
- GNU CC is also designed for 8-bit addressable machines. Thor can only address words, not bytes. Therefore, to support byte addresses one would have to generate extra code that extracts (or inserts) bytes from a word. All this makes it difficult to support the natural semantics of every data type found in the C language, in the meaning that characters occupy 8 bits and static strings can be packed so one word contains 4 characters. It is a hard problem to make the compiler understand this, since it takes for granted that one can address bytes.

- Not included in the C language is the hardware Ada support found in Thor, which is implemented as special task instructions in the instruction set. Therefore it is very hard to make use of these instructions. An advanced solution is to design a set of library functions utilizing the Ada instructions.
- Usually when designing a compiler, one assigns certain registers to keep track of special things when executing the compiled program, such as a frame pointer, a top-of-stack pointer, a register to hold the return value when returning from functions etc. The most urgent problem seemed to be the frame pointer. Thor only has a top-of-stack pointer, no frame pointer. We must force the compiler to eliminate the frame pointer and to only use the top-of-stack pointer. However, according to the GNU CC documentation one can only tell the compiler to make a try to eliminate the frame pointer. There is no guarantee that it will succeed. If it fails, the assembler will get incorrect input and also fail, or deliver incorrect object files.
- The intermediate representation consists of RTL instructions where each instruction ideally matches one machine instruction. The RTL instructions are quite flexible, and one has many options when designing the intermediate language. Thor has the rather unique '`PSH`' and '`POP`' machine instructions, which need some kind of special representation in the intermediate language. None of the previous ports have had this problem and we did not know whether GNU CC could handle it or not.
- To fully support all different extensions of the C language supported by the GNU CC, like nested functions, and to support other front ends, like the C++ language, one would need to implement several tricky features. These features are normally implemented when one does a port of the GNU CC, but as our main goal is to support the ANSI C language without any special extensions we can skip them if they prove to be too difficult. If they are easy to implement we should include them. For example, one might need a more complex stack frame (*see 3.3.1.2, Function calling interface*) with a static and dynamic chain, and trampolines[1] to handle addresses to local functions.

3.2 Strategy

Of all the problems mentioned above, the problems with the lack of registers, the '`PSH`' and '`POP`' representation and the frame pointer were the most troublesome ones. The task instructions were not something we had to support, so we could forget them, and leave the matter for future work. The 8-bit pointer problem was also avoided in a sense, by letting the character type occupy a whole word of 32 bits. To simplify things, we also skipped the GNU CC's extensions to the C language. These extensions could be added later on and we could concentrate on designing a working ANSI C compiler.

We will now take a closer look at the main problems and their solution.

1. A piece of code created at run-time.

3.2.1 The lack of registers in Thor

When RTL is generated, the compiler uses pseudo registers whenever it needs some kind of storage. A pseudo register is never reused, so in big functions they are numerous. In later passes, the compiler tries to allocate the pseudo registers to the available hard registers (local and global allocation, *see 2.2.1.13 and 2.2.1.14*). Often, the hard registers will be too few and some pseudo registers will be left unallocated. These unallocated pseudo registers will be put in stack slots (the reload pass, *see 2.2.1.15*).

In our case we do not have any hard registers to allocate. But we still must find a way to please the compiler. Our first thought, in the very beginning, was to use the pass dealing with the Intel 387 numerical coprocessor in some way. The 387 pass seemed to do exactly what we were supposed to do, to convert registers into stack registers. After studying the source code we soon gave up. The 387 pass is too specialized, we think, to be easy to change for our needs. Thor's stack works in a different way compared to the 387 stack. The source code would need to be rewritten in many ways, and it would probably be better to write a completely new pass for the compiler.

One of our design principles has been to try to make things work without changing the GNU CC source code, and without introducing any new passes in the compiler. With this in mind we discarded the 387 solution.

The fundamental principle is that it is always possible to pretend that Thor has a lot of registers, and then later on, when the assembler instructions are output, change these registers into stack slots. For example, if we tell the compiler that we have 4 hard registers, we can allocate 4 extra stack slots in the *function prologue* (*see 3.3.2.8, Assembler format*) and then change all references to these 4 registers into stack slot references (stack relative addresses). The only question is, how many hard registers should we choose? We have studied three different cases; no registers, one register or many registers.

With *a lot of registers*, the compiler will always succeed. There will always be enough hard registers when allocating the pseudo registers. But still, how many is a lot? Ideally, we would tell the compiler to use just as many hard registers as it needs. But this is impossible since the number of registers must be set in advance when building the compiler and cannot be changed dynamically. The alternative is to choose a very large number of registers. But then we would waste stack slots and the compiler might choose not to do it's best when allocating small functions. A good compromise might be to examine some typical C functions (is there any typical C function?) and find out the maximum number of registers needed and then choose this number.

A more simple solution (and closer to the truth) is to pretend that Thor has *no hard registers* at all. The compiler would then have to put all the pseudo registers into stack slots and this would make the output of the assembler instructions simpler, because no register references would exist, only stack relative memory references. This solution is simple but it is also a bit stupid, because the compiler does not try any optimizations when reloading pseudo registers into stack slots. This means that if we have 100 pseudo registers they will be reloaded into 100 stack slots even if, for example, 20 hard registers would have sufficed when using a more clever allocation algorithm.

The solution with no hard registers is also a bit risky. In the reload pass, the compiler tries to match the operands with the constrains given in the machine description. If it fails, it will reload

the failing operand via a hard register. In these cases, there must be a hard register available, otherwise the compiler will crash. When writing the machine description we have tried to design the constrains so that they always match. Still, if a match fails and no hard registers are available, the compiler will crash.

The solution we have chosen, is to use *one hard register*. We tell the compiler that we have one hard register called the '`TOP`'-register and then change all references to this register into a stack relative memory reference. We also allocate one extra stack slot in the function prologue.

With one hard register we get a more secure solution than with no registers at all and we can handle one reload of an operand via a hard register. The compiler will also be able to use this register when allocating the pseudo registers and will tend to use this register quite frequently. The frequent use makes the peephole optimization (*see 3.3.5.1, Machine-specific peephole optimizations*) more successful in removing unnecessary push/pop sequences with temporary values.

Our first thought was that the '`TOP`'-register would in some way correspond to Thor's internal TOP register. This is true in the sense that the '`TOP`'-register sometimes resides at the top of the stack, but as our solution gradually evolved, it lost this tight connection with the internal TOP register, and is now almost like any other stack slot. The solution still has the advantages of being more secure and the code generated is easier to optimize.

3.2.2 Frame pointer elimination

A frame pointer is used to point out the location in the stack where one can find the parameters passed to a function and the allocated stack slots for local variables. It is a hard register (dedicated or general) which is initialized in the function prologue and used whenever access is needed to the frame. Some microprocessors can use this register as a general register, therefore it is desirable to eliminate the frame pointer whenever it is possible. If the top-of-stack pointer does not change inside an executing function, the offset between the frame pointer and the top-of-stack pointer will remain constant. In this case it is possible to change all references to the frame pointer into references to the top-of-stack pointer (with an added offset) and the frame pointer can be eliminated.

The Thor microprocessor does not have any frame pointer. So in our case it is not only desirable to eliminate the frame pointer, it is essential that it is successfully eliminated. Our strategy in this case is very simple. We tell the compiler to try to eliminate the frame pointer and we also make sure that it never fails by not using any kind of dynamic stack allocation. If the elimination fails it will crash.

Are there any disadvantages with this strategy? Well, dynamic allocation from the stack is forbidden. Therefore we can not permit the C function '`alloca`', which is common in UNIX systems. This function allocates memory from the stack (frame) of the calling function. The allocated memory is freed automatically when the called function ends. Moreover, we are not sure whether or not the compiler by itself can generate code to dynamically allocate memory from the stack. Our strategy has proven to be a working one but it might also be a bit risky.

3.2.3 The 8-bit addressing problem

The GNU C Compiler has been designed for machines with an 8-bit addressable memory and

with a word size of 32 bits, or at least, for machines that have instructions dealing with bytes (8 bits). This means that the normal data types in C, like the character type, can quite easily be defined to have their normal sizes. For example, the character type can be defined as 8 bits and the integer type as 32 bits, as we are used to, without any special consideration.

The Thor microprocessor can only address 32-bit units (the word size). This causes no problem when defining a suitable size of the integer type, but for the character type the size choice is not so obvious. If we choose a character size of 32 bits, we get a simple solution where no special instructions are needed to handle the characters and where characters will be fast to work with. On the other hand, if we choose 8-bit characters, we will save memory by packing 4 characters into every word. In order to work with these characters we must output special instructions to extract or insert 8-bit parts of a word. This would make the 8-bit solution slower. In addition, the compiler would not offer any help because it has no support for a 32-bit addressable memory.

In the C language it is not possible to have different pointer units. For example, we cannot make pointers to characters contain byte addresses, and pointers to integers contain word addresses. We must have pointers that contain either byte or word addresses, not both kinds. This is due to the fact that pointers can be assigned to each other, and we have no strict type checking in C. The issue with different pointers in C has been widely discussed: take a look, for example, in the usenet newsgroup *comp.compilers* or *gnu.gcc.help*.

To sum up, we find that if we want an 8-bit character type, we must have pointers containing byte addresses. This influences all types, and for all pointers to objects aligned to word boundaries, we must dereference the pointer, i.e. divide it by 4 to get a word address from the original byte address. In all cases of pointer usage we must add extra code to handle the conversion of byte to word addresses.

After considering all the advantages and disadvantages with the 8-bit pointer solution we judged that the 8-bit solution was too cumbersome. The solution with 32-bit pointers would be much easier to implement and would result in faster code. The memory wasted, when using large character arrays, was not of major concern and could be handled on the user level. With 32 bits as the smallest size, it was natural to define the size of all the different types to be 32 bits. In this way, the compiler's internal 'QI'-mode would represent one word, and the only patterns we would need to write in the machine description would be the 'QI'-mode patterns (plus the 'QF'-mode patterns to handle the float and double types).

To show how the memory waste could be overcome we present this small example. It is always possible to define a bit structure in C, as follows:

```
struct packed_char
{
     int char0 : 8;
     int char1 : 8;
     int char2 : 8;
     int char3 : 8;
};
```

Example 8 A bit structure

There will be no problem with declaring an array built up by these 'packed_char' structures, but to be able to work with strings of this kind, one needs new string handling functions. Anyway, all this shows that even if the compiler does not support 8-bit characters one can still solve the memory waste problem, when it is necessary. There is no doubt though, that it would be more user friendly to make the compiler support it directly.

Our decision to let pointers contain word addresses also has further implications. A special case is the PC register in Thor, which contains half-word addresses (16 bits). This means that all pointers to functions need special treatment. A pointer to a function contains a word address and has to be multiplied by 2 to get a half-word address (which can be used, for example, in an indirect call). This causes no problem, but consider the case when one needs the address of a function. The address has to be divided by 2 to fit in the word pointer and we will lose one bit of information. The result is that functions must never lie on an odd address, they must always be aligned to a 32-bit word boundary.

Apart from the PC register problem, there is also some internal design issues. The assembler interpret all offsets (and all addresses) as byte offsets. Therefore one must not forget to translate the compiler's word offsets into byte offsets (multiply with 4).

3.2.4 The representation of 'PSH' and 'POP'

When designing the machine description, one has to make many strategic decisions. One of them is which kind and how many machine patterns to define in the '.md'-file. The idea behind the patterns is to have named patterns ('define_insn' or 'define_expand') that emit one or several RTL instructions, where each instruction represents a machine instruction, and then have patterns (only 'define_insn') to translate these into assembler output. But this is only an idea. There are other possible designs.

The most extreme design (which deviates most from the basic idea) would be to let all the standard names correspond to one RTL instruction and in the end output a sequence of assembler instruction for each RTL instruction. There would be no need for 'define_expand'-expressions and the compiler would never fail in any way when working with the RTL instructions. This design would have fit quite well, but it has some drawbacks. When the RTL instructions correspond to several machine instructions, it is impossible to do any delay slot filling. It is also difficult to do peephole optimization because there would be too many combinations to cover.

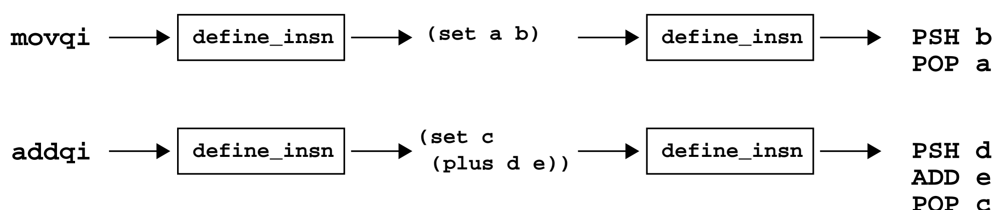


Figure 4 The extreme design

The next step towards the basic idea would be to begin precisely as in the previous design, and

then use 'define_split'-definitions[1] to split the RTL instructions. Each new RTL instruction would then represent one machine instruction. However, the compiler only tries to split RTL instruction when doing delay slot filling, and it only tries to split *some* instructions, not all of them. So, we would maybe get a working delay slot filling, but it would still be difficult to do peephole optimizations. There might be a way of solving this though, and make the compiler to try to split all instructions before doing any delay slot filling and peephole optimization. But we have not done enough research in this area, so we do not know if this might be a promising design.

Eventually, we came to the conclusion that it would be best to try to follow the basic idea. When the RTL was generated, one should try to make each RTL instruction correspond to one machine instruction. We found that some machine instructions were quite easy to express in the RTL expressions. For example, if the compiler wants to do a 'addqi', we emit three instructions: one to push the first operand on the stack, one to add the other operand and finally one to pop the result back into the destination operand. The 'add' instruction was straightforward to implement. The big problem, however, was how to express Thor's 'PSH' and 'POP' instructions in the RTL expressions.

The problem with the 'PSH' and 'POP' instructions was going to be a tricky one. The first thing we tried, was to follow the rules and the semantics of the RTL expressions, by the book. We found these mathematically correct representations for the 'PSH' and 'POP' instructions:

```
PSH  (set (mem:QI (pre_dec:QI (reg:QI 2 TOS)))
          source_operand)

POP  (set destination_operand
          (mem:QI (post_inc:QI (reg:QI 2 TOS))))
```

Figure 5 Our first try

This representation of the 'PSH' instruction is identical to the way the compiler represents the normal push instructions (those occurring before a call). 'pre_dec' means to decrement the top-of-stack pointer before storing the source operand. 'post_inc' means to increment the top-of-stack pointer after loading the destination operand. The representation shown above corresponds to the actual actions performed inside the Thor microprocessor.

With the help of the 'pre_dec' and the 'post_inc' one can describe the 'PSH' and 'POP' instructions in a neat way. GNU CC understands these instructions, but the sad thing is that it does not try any optimization on them. When it sees a 'pre_dec' or a 'post_inc', it treats the instruction as having a side effect and then abandons most attempts to optimize it. The 'pre_dec' and 'post_inc' directives tell the compiler that the instructions will be too complex to be worth bothering about. The reason for this is quite simple. No other port of the GNU CC have used 'pre_dec' and 'post_inc' in this way. The 'pre_dec' instructions (normal push instruction) never need any optimization, and the 'post_inc' directive is inserted by the

1. 'define_split'-definitions are another way of implementing the handling of the instructions, apart from 'define_expand'- and 'define_insn'-definitions.

combine phase, when a great deal of the optimization is already done.

The 'pre_dec'/'post_inc' solution was not a good one. We could just as well have used the extreme design, explained above, with a better result. We had to find some other way of representing the 'PSH' and 'POP' instructions, a way that let the compiler work with and optimize the RTL instructions.

The winning solution was to invent a new meaning for some RTL expressions. We introduced a new register called the 'TMP'-register. The 'PSH' instruction could now be represented by a store operation to the 'TMP'-register, and the 'POP' instruction by a load operation from the 'TMP'-register. The 'TMP'-register would act as a temporary top-of-stack value and it was only allowed to be used by our 'define_expand' patterns. The compiler is not supposed to use it when allocating pseudo registers.

```
PSH  (set (reg:QI 1 TMP) source_operand)
POP  (set destination_operand (reg:QI 1 TMP))
     (clobber (reg:QI 1 TMP))
```

Figure 6 The final design

One problem with this scheme was that the compiler, according to the semantics of the RTL, believed that the 'TMP'-register still contained a value after the 'POP' instruction. Therefore, we had to add a 'clobber'[1] instruction at the end, to indicate that the 'TMP'-register was destroyed in the process of doing a pop. At first, we tried to emit this 'clobber' within a parallel expression together with the 'set' instruction. This would have been the most correct way of expressing it, we thought, but it did not work. The compiler still used the 'TMP'-register after the clobber had destroyed it (a bug somewhere). So, we just emitted the 'clobber' instruction as a separate instruction at the end, and that seems to work fine.

There is an ever-threatening danger with our solution. We have done a small change in the semantics of the RTL and the compiler's optimization passes interpret the instructions using the original meaning. The compiler might do something we consider illegal according to our semantics. For example, the compiler might remove an unnecessary 'POP' instruction without understanding that the 'PSH' instruction must be removed as well. If both are not removed, we will get an erroneous program, which corrupts the stack when it executes.

To this day, we have manage to avoid all problems caused by the changed semantics, but it has made our work a little more restrictive and we have not been able to implement some things in the best possible way (*see 3.3.3.6, Condition code setting instructions*). But we feel that we now have a strategy that works.

3.3 Solution

We will now take a closer look on our solution and describe the different parts. The main parts are the '.h'-file and the '.md'-file, but some other auxiliary files may also need a little explanation.

1. A 'clobber' expression takes a register as a operand and by this it informs the compiler that the register's value is destroyed at this point.

The '.h'-file is mostly described in the section *3.3.2 'Target control macros'*. The '.md'-file in the sections *3.3.3 'Machine description instruction patterns'*, *3.3.4 'Attributes'*and *3.3.5 'Optimizing the code'*. The first section below presents some basic definitions, and can also be interesting for a user of the compiler.

3.3.1 Specification of system dependent definitions

In the following sections one finds basic definitions describing the data types and function calling interface. These definitions are fundamental for the way the compiler works and are also necessary to be aware of when using the compiler.

3.3.1.1 The basic C data types

The standard data types used by the GNU CC for Thor is not the ones that one might be used to, due to previously explained reasons. The table below lists the supported data types.

Table 4 The basic C data types known by the GNU CC for Thor.

Type	Size in bits	Comment
'char'	32	Signed by default.
'short'	32	
'int'	32	
'long'	32	
'long long int'	32	GCC extension.
'float'	32	
'double'	32	
'long double'	32	

As you can see, all data types have a size of 32 bits, Thor's word size. This is so because it made it easier to port GNU CC. To support 64 bit data types, one would need to write several help functions to cover all the different operators, and maybe some new patterns in the machine description. To make the size of the character type smaller, one would need to change the machine description quite a lot (*see 3.2.3, The 8-bit addressing problem*).

3.3.1.2 Function calling interface

The function calling interface has been designed to be as simple as possible. It must explain how the parameters of a function are passed and how a return value is returned.

The caller is responsible for pushing all the parameters on the stack before the call. It may also push an optional structure return address and finally the normal return address. The callee (the called function) is responsible for allocating stack space for it's local variables, for returning a return value (when appropriate), and finally for cleaning up the stack and return. When calling a function with a variable number of parameters, the caller will clean up the pushed parameters

instead of the callee. It is very important that every function is properly aligned to a 32-bit word boundary. Otherwise, it will not be possible to store the address of the function in a C pointer.

3.3.1.2.1 The normal case

The normal case is when we are not returning any structure value and not passing a variable number of parameters. *Figure 7* shows the stack in the normal case, just after executing the call instruction and before entering the called function.

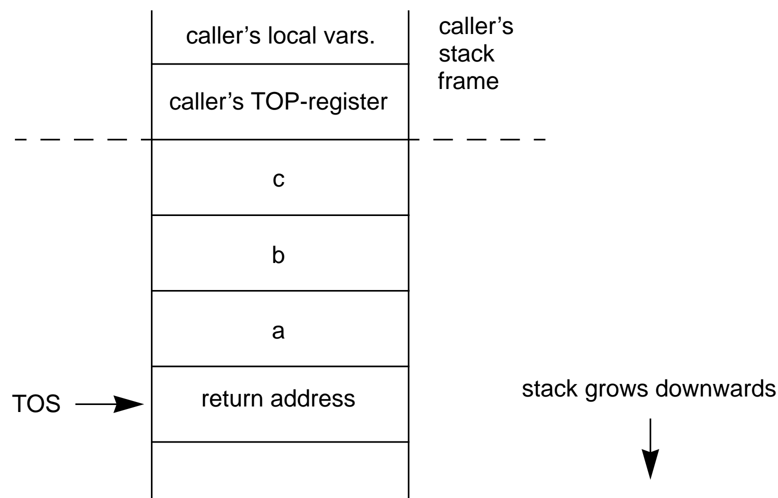


Figure 7 View of the stack in the normal case

The parameters are pushed onto the stack in backwards order, beginning with the last one and ending with the first one. The return value should be placed in the caller's 'TOP'-register. If it is not needed, one does not need to do anything.

The caller will not remove the pushed parameters afterwards. It is the callee's responsibility to clean the stack before returning. After the return, the top-of-stack pointer should point to the caller's 'TOP'-register, where the return value resides. A warning might be needed here, because it is possible to call functions with the wrong number of parameters, if not a proper prototype is defined. If such a call is made, the callee will clean up the wrong number of parameters and the stack will be corrupted (but anyway, it is an error to write such a program).

Example 9 shows a typical calling sequence and a typical called function when doing a normal call.

3.3.1.2.2 Structure value return

When returning a structure value (not a pointer to a structure) the size of the returned object can exceed 32 bits and not fit in one word. Therefore, we have chosen to let the compiler return all aggregate types in memory and not in the 'TOP'-register. *Figure 8* shows the stack when returning a structure value, just after executing the call instruction and before entering the called function.

<table>
<tr><th>Calling function</th><th>Called function</th></tr>
<tr><td>

```
  fun (999, 'A', 3.14);


       ...
       PSH  LC0
       PSHI 65
       PSHI 999
       CALL _fun
       NOP
       NOP
       ...
```

</td><td>

```
int fun (int a, char b, float c)
{
  int d,e;
  return 2;
}


       DATA 0     ;align function
_fun:
       MTOS -3    ;reserve space on stack
       PSHI 2
       POP  1     ;return value -> TOP
       RET  3     ;return address now at off. 3
       POP  7     ;copy TOP to caller's TOP reg
       MTOS 6     ;clean up local + param.
```

</td></tr>
</table>

Example 9 Calling sequence when doing a normal call

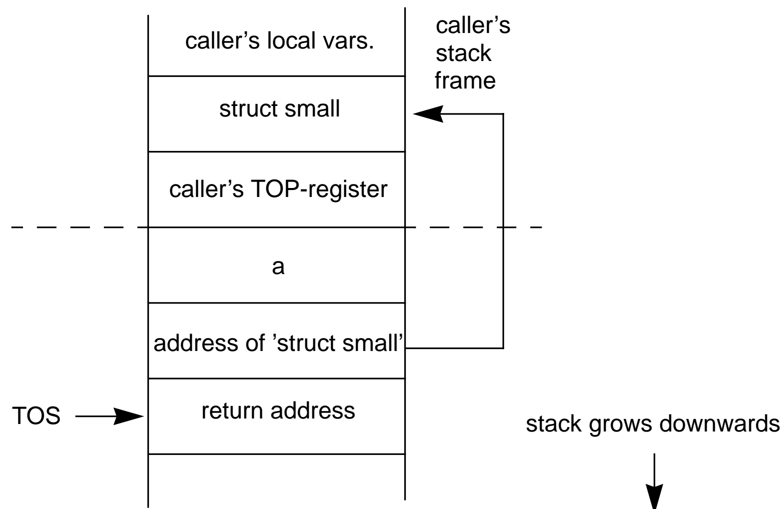


Figure 8 View of the stack in the structure value return case

When the return value is a structure, regardless of its size, the caller reserves space for a structure to hold the return value and passes a pointer to this structure to the called function. The pointer is pushed at the end, when all other parameters have been pushed, as a parameter that is invisible to the user. The callee must then copy the return value (the contents of the structure to return) to the appropriate place in memory before returning. In addition, the address of the structure (the first parameter) should be returned in the normal way by placing it in the caller's `'TOP'`-register.

Example 10 shows a calling sequence and a typical called function when returning a structure value (the called function is only an example, not a real generated function)

.

<table>
<tr><th>Calling function</th><th>Called function</th></tr>
<tr><td><pre>
struct small { int item; };

void main ()
{
  struct small s;

  s = fun2 (999);
}

       ...
       MTOS -2   ;s, TOP
       PSHR TOS   ;calc. address
       ADDI 1   ;of s
       POP 1
       PSHI 999   ;the parameter
       PSH 1   ;address of s
       CALL _fun2
       NOP
       NOP
       ...
</pre></td><td><pre>
struct small fun2 (int a)
{
  struct small t;

  t.item = 3;
  return t;
}

       DATA 0   ;align function
_fun2:
       MTOS -2   ;t, TOP
       PSHI 3
       POP 2   ;store 3 into t.item
       PSH 3   ;address of s
       MTOS 1
       PSH 1   ;t
       POPX 0   ;copy t to s
       PSH 3   ;address of s
       POP 1   ;TOP
       RET 2   ;return address now at off. 2
       POP 4   ;copy TOP to caller's TOP reg
       MTOS 3   ;clean up local + param.
</pre></td></tr>
</table>

Example 10 Calling sequence when returning a structure value

3.3.1.2.3 A variable number of parameters

This case resembles the normal one apart from the fact that the called function does not know the exact number of parameters that have been passed. To handle this situation the called function acts as if there were no parameters at all, and the calling function has to clean up the stack afterwards. *Figure 9* shows the stack when a variable number of parameters are passed, just after executing the call instruction and before entering the called function.

The caller first pushes all parameters in the normal way and then make the call. The called function runs just as an ordinary function would do, but when returning it does not copy the return value to the caller's 'TOP'-register. Instead, it puts it one stack slot above the return address. This is not so strange, because the called function acts as if there were no parameters at all, and quite correctly believes that the caller's 'TOP'-register resides adjacent to the return address. As a consequence, the called function only cleans up it's local variables and the return address and leaves the return value on the stack top.

After the call, the caller must restore the stack to its original state. This means that the return value at the top of the stack must be popped to the 'TOP'-register and the pushed parameters must be removed. *Example 11* shows a typical calling sequence and a typical called function when passing a variable number of arguments.

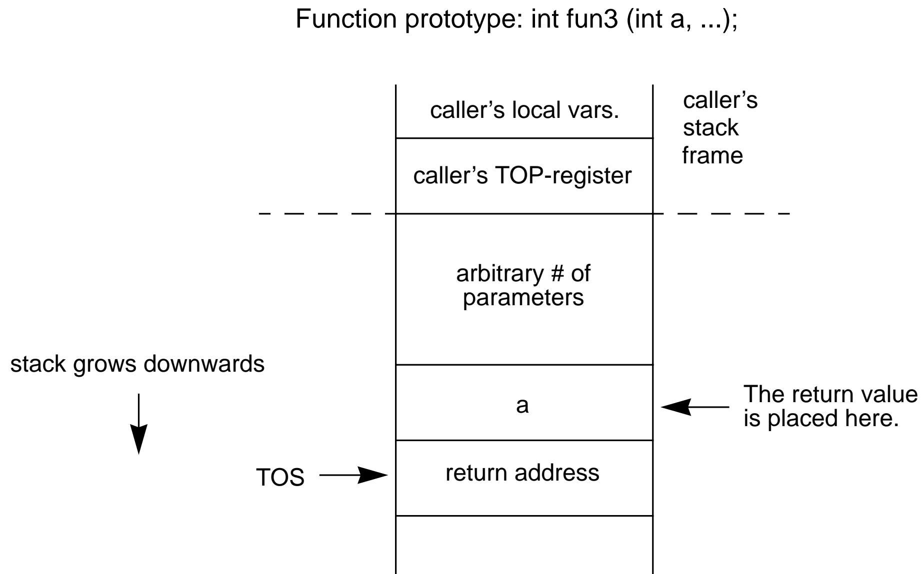


Figure 9 View of the stack when passing a variable number of parameters

<table>
<tr><th>Calling function</th><th>Called function</th></tr>
<tr><td><pre>void main ()
{
  int a;

  a = fun3 (2, 'A', 'B');
}

       ...
       PSHI 66   ;push parameters
       PSHI 65
       PSHI 2
       CALL _fun3
       NOP
       NOP
       POP 3   ;copy value at
                  ;top to TOP-reg
       MTOS 2   ;remove rest of
                  ;parameters
       ...</pre></td><td><pre>int fun3 (int n, ...)
{
  return 3;
}

       DATA 0   ;align function
_fun3:
       MTOS -1   ;TOP
       PSHI 3
       POP 1   ;store 3 into TOP
       RET 1   ;return address at off. 1
       POP 2   ;copy TOP onto n
       MTOS 1   ;clean up return address</pre></td></tr>
</table>

Example 11 Calling sequence when passing a variable number of parameters

In the C language, a variable number of parameters is indicated by the ellipsis '...'. It is necessary that both the caller and the callee are aware of the type of the function, since each part must behave differently. There must be a proper function prototype corresponding to the actual function definition, which includes the ellipsis '...'. Normally, when using a variable number of parameters, the first parameter contains some kind of information about the total count of param-

eters. All parameters are pushed backwards, so this first parameter will always have a fixed offset from the stack top. If the parameters had been pushed in the other direction, it would have been very difficult to handle a variable number of parameters.

3.3.1.2.4 Structure value return and a variable number of parameters

When returning a structure value and at the same time passing a variable number of parameters, we have a case that is simply a combination of the two last cases. The caller pushes one extra parameter on the stack, the address of the return structure. The called function fills this structure and then returns. The caller then removes all the parameters plus the extra stack slot for the address. There is usually no need to move the return value from the top of the stack to the 'TOP'-register, since the return value is simply the address pointing to the return structure and is of no use to the caller.

3.3.2 Target control macros

The following subchapters will give an description of how our '.h'-file layout finally turned out. While reading the sections we suggest that you keep a finger in the appendix where a complete listing of the '.h'-file is found (*see APPENDIX A -, Listing of machine dependent files*). In order to simplify searching among the macros we have grouped macros controlling similar things into sections separated by generous comment lines. In these sections we have tried to put emphasis on thoroughly explaining solutions vital for the functionality of the compiler. Macros that we believe to be self-explained, or not essential for understanding of the port, have been left out or described very briefly.

3.3.2.1 Thor-specific macros

Every time we have detected a bug in GNU CC or for other reasons have felt compelled to make a patch in the source files, we have invented a new macro to control the activation of the alteration. The standard convention we have used is that each of these macro names start with 'THOR_' concatenated with another string. For more information concerning the source alterations themselves, see section *3.3.6 'Changes in the source files of GNU CC'*.

3.3.2.2 Link sections

An object file is divided into sections containing different types of data. In the most common case, there are three sections: the *text section*, which holds instructions and read-only data; the *data section*, which holds initialized writable data; and the *bss section*, which holds uninitialized data. Some systems have other kinds of sections. The compiler must tell the assembler/linker when to switch sections. For this reason there are macros designed to control the section switching commands to the assembler.

In the Thor assembler/linker we only have, in principle, two different sections, one where code is placed and one where data is placed. The macros 'TEXT_SECTION_ASM_OP' and 'DATA_SECTION_ASM_OP' are the macros used to control the assembler output when switching segments. Those macros define the suitable strings to output to the assembler file when the compiler calls them.

An embedded computer system is equipped with various kinds of memory, such as ROM and RAM[1]. We must protect the initialized data from being destroyed when restarting the system. A

good compromise is therefore to make the system copy the entire data segment in ROM (containing both initialized and uninitialized data) to a new fresh data segment in RAM when starting the system. In this case one can restart the system without any loss of initialized data.

3.3.2.3 Type and Storage layout

Here we define macros that control endianness, addressable units, alignments of data and object files, roundings, various type sizes and the floating-point format. For Thor we have chosen to treat the sizes of all types as being equal to the least addressable unit which is a word (*see 3.2, Strategy*). Most macros encountered with these issues are very simple to understand, since they all look like a macro followed with a number telling the size of it, like '`#define POINTER_SIZE 32`', which obviously defines the pointer size.

3.3.2.4 Registers and Register Classes

This section explains how we describe what registers the target machine has, and how (in general) they can be used. The description of which registers a specific instruction can use is done with register classes. The numbering of the pseudo-registers, used in the intermediate language, begins with the register number following the last of the hardware registers, and must be defined by the macro '`FIRST_PSEUDO_REGISTER`'.

According to our strategy (*see 3.2, Strategy*) we planned to let GNU CC only have one single register to use when allocating (the '`TOP`'-register). However, that is not the whole truth, since we in fact allow yet another register, which is only used in '`define_expand`'-patterns (*see 2.2.4.1, The Machine Description file, '.md'-file*), when the generated patterns exchange data with each other. This register (called '`TMP`', which is a temporary top-of-stack) should not be freely accessed by GNU CC. A store into this register represents a '`PSH`'-instruction, a load from this register is a '`POP`'-instruction. The two remaining register are '`TOS`' (top-of-stack) and '`FP`' (frame pointer), both fixed. The macro '`FIXED_REGISTERS`' defines a set with '`1`' or '`0`', where '`1`' indicates that a register is fixed, i.e. not for arbitrary use. The position in the set tells which register the definition controls.

'`CALL_USED_REGISTERS`' is a macro which defines a set of values which is '`1`' for registers not available across function calls. These must include the '`FIXED_REGISTERS`' and also any registers that can be used without being saved.This macro therefore identifies the registers that are not available for general allocation of values that must live across function calls. Consequently, on Thor all four registers are set to '`1`'s.

'`HARD_REGNO_NREGS (REGNO, MODE)`' is a C expression for the number of consecutive hard registers, starting at register number '`REGNO`', required to hold a value of mode '`MODE`'. On Thor all registers hold the size of a word, and since all operations occur in 32 bits an adequate definition is equal to '`1`'.

'`HARD_REGNO_MODE_OK (REGNO, MODE)`' is a C expression that is nonzero if it is permissible to store a value of mode '`MODE`' in hard register number '`REGNO`' (or in several registers starting with that one). On Thor we can only say it is okay when the mode is either of '`QImode`' or '`QFmode`'. Thus no discrimination occurs between floating-point and integer data.

---

1. ROM stands for Read Only Memory and RAM stands for Random Access Memory

'MODES_TIEABLE_P (MODE1, MODE2)' is a C expression that is nonzero if it is desirable to choose register allocation to avoid move instructions between a value of mode 'MODE1' and a value of mode 'MODE2'. Value is '1' if it is a good idea to tie two pseudo registers when one has mode 'MODE1' and the other has mode 'MODE2'. Therefore this macro is tied directly to '1' on Thor.

On many machines, the numbered registers are not all equivalent. For example, certain registers may not be allowed for indexed addressing; certain registers may not be allowed in some instructions. These machine restrictions are described to the compiler using *register classes*. You define a number of register classes, giving each one a name and saying which of the registers that belong to it. You can then specify register classes that are allowed as operands to particular instruction patterns. This feature is not controlled by a macro but an C enumeration type called 'reg_class'. Several other macros in turn use this type when defining which registers are allowed as base pointers etc. Essentially we have three different classes, one for 'TOP', one for 'TMP' and one for the union between those two.

'REG_CLASS_FROM_LETTER (CHAR)' is a C expression which defines the machine-dependent operand constraint letters for register classes. If 'CHAR' is such a letter, the value should be the register class corresponding to it. Otherwise, the value should be 'NO_REGS'. In Thor we have defined one such letter, namely 't', which corresponds to the 'TOP'-register.

Normally the compiler avoids choosing registers that have been explicitly mentioned in the RTL as spill registers (these registers are normally those used to pass parameters and return values). However, some machines have so few registers of certain classes that there would not be enough registers to use as spill registers if this were done (for sure the case for Thor). Therefore we define 'SMALL_REGISTER_CLASSES'. When this is defined, the compiler allows registers explicitly used in the RTL to be used as spill registers but avoids extending the lifetime of these registers.

The macro 'CONST_OK_FOR_LETTER_P (VALUE, C)' represents a C expression that defines the machine-dependent operand constraint letters that specify particular ranges of integer values. If 'C' is one of those letters, the expression should check that 'VALUE', an integer, is in the appropriate range and return '1' if so, '0' otherwise. If 'C' is not one of those letters, the value should be '0' regardless of 'VALUE'. In Thor we have six letters defined from 'I' to 'N' which each defines an integer range (*see Table 5*). The ranges are based on Thor's different instruction formats. 'CONST_DOUBLE_OK_FOR_LETTER_P(VALUE, C)' is a similar C expression that defines the machine-dependent operand constraint letters that specify particular ranges of 'const_double' values. Since no double-float data is allowed in the current version this macro is hard-coded to '0'.

Table 5 Operand constraint letters defined by 'CONST_OK_FOR_LETTER_P'

Letter	Lower bound	Higher bound
'I'	-800000_{16}	$FFFFFF_{16}$
'J'	0_{16}	$FFFFFF_{16}$
'K'	-800000_{16}	$7FFFFF_{16}$
'L'	-80_{16}	$7F_{16}$

Table 5 Operand constraint letters defined by 'CONST_OK_FOR_LETTER_P'

Letter	Lower bound	Higher bound
'M'	0_{16}	FF_{16}
'N'	0_{16}	20_{16}

The macro 'EXTRA_CONSTRAINT (VALUE, C)' represents a C expression that defines the optional machine-dependent constraint letters that can be used to segregate specific types of operands, usually memory references, for the target machine. It should return '1' if 'VALUE' corresponds to the operand type represented by the constraint letter 'C'. If 'C' is not defined as an extra constraint, the value returned should be '0' regardless of 'VALUE'. For Thor we define the letter 'Q' as a constraint for PC- or stack-relative memory address.

'PREFERRED_RELOAD_CLASS (X, CLASS)' is a C expression that places additional restrictions on the register class to use when it is necessary to copy value 'X' into a register in class 'CLASS'. The value is a register class; perhaps 'CLASS', or perhaps another, smaller class. For Thor it is sufficient to return 'CLASS' when this macro is called.

'CLASS_MAX_NREGS (CLASS, MODE)' is a C expression for the maximum number of consecutive registers of class 'CLASS' needed to hold a value of mode 'MODE'. This is closely related to the macro 'HARD_REGNO_NREGS'. In fact, the value of the macro 'CLASS_MAX_NREGS (CLASS, MODE)' should be the maximum value of 'HARD_REGNO_NREGS (REGNO, MODE)' for all 'REGNO' values in the class 'CLASS'. In principle we could have defined this macro to '1' for Thor, but we use an arithmetic expression instead, built from macros defined earlier, that presently always will evaluate to '1'. We do this in order to be independent if changes are made to the earlier defined macros.

'REGNO_OK_FOR_BASE_P (NUM)' and 'REGNO_OK_FOR_INDEX_P (NUM)' is a C expression which is nonzero if register number 'NUM' is suitable for use as an index/base register in operand addresses. It may either be a suitable hard register or a pseudo register that has been allocated to such a hard register. The index macro returns true if 'NUM' is equal to the 'TOP'-register while the base macro returns true if 'NUM' is any of Thor's four registers except the 'TOS'-register.

3.3.2.5 Stack and calling

There is a macro controlling which direction the stack grows, called 'STACK_GROWS_DOWNWARD'. This is set for Thor, and the same is true about a corresponding macro 'FRAME_GROWS_DOWNWARD', which controls the frame layout.

'FRAME_POINTER_REQUIRED' is a macro which tells if the frame pointer must exist. Value should be nonzero if functions must have frame pointers. Zero means the frame pointer need not be set up (and parameters may be accessed via the stack pointer) in functions that seem suitable. For Thor we definitely want to eliminate the frame pointer since we do not have one.

'RETURN_POPS_ARGS(FUNDECL,FUNTYPE,SIZE)' represents a C expression that should indicate the number of bytes of its own arguments that a function pops on returning, or '0' if the function pops no arguments and the caller must therefore pop them all after the function returns.

'FUNDECL' is a C variable whose value is a tree node that describes the function in question. 'SIZE' is the number of bytes of arguments passed on the stack. On Thor we return 'SIZE' when we have a fixed number of arguments or when we have an *implicit library call*. If the function has a variable number of arguments we return '0'.

'FUNCTION_ARG (CUM, MODE, TYPE, NAMED)' is a C expression that controls whether a function argument is passed in a register, and which register. Value is zero to push the argument on the stack, or a hard register in which to store the argument. For Thor all arguments are passed on the stack, and therefore the value of the macro is always zero.

'CUMULATIVE_ARGS' represents a C type for declaring a variable that is used as the first argument of 'FUNCTION_ARG' and other related values. For some target machines, the type 'int' suffices and can hold the number of bytes of argument so far. There is no need to store anything in 'CUMULATIVE_ARGS' for target machines on which all arguments are passed on the stack, like Thor. However, the data structure must exist and should not be empty, so we use 'int'.

'INIT_CUMULATIVE_ARGS (CUM, FNTYPE, LIBNAME)' is a C statement for initializing the variable 'CUM' for the state at the beginning of the argument list. The variable has type 'CUMULATIVE_ARGS'. The value of 'FNTYPE' is the tree node for the data type of the function which will receive the args, or '0' if the args are to a compiler support library function. For Thor this macro always initialize the variable 'CUM' to zero.

'FUNCTION_ARG_ADVANCE (CUM, MODE, TYPE, NAMED)' is a C statement to update the summarizer variable 'CUM' to advance past an argument in the argument list. The values 'MODE', 'TYPE' and 'NAMED' describe that argument. This macro need not do anything if the argument in question was passed on the stack. The compiler knows how to track the amount of stack space used for arguments without any special help, so for Thor this macro need not do anything.

'FUNCTION_VALUE(VALTYPE, FUNC)' defines how to find the value returned by a function. The macro represents a C expression to create an RTX representing the place where a function returns a value of data type 'VALTYPE', which is the data type of the value (as a tree). If the precise function being called is known, 'FUNC' is its 'FUNCTION_DECL'; otherwise, 'FUNC' is '0'. On Thor, we always create a RTX with the 'TOP'-register where scalar returns always occur.

'LIBCALL_VALUE (MODE)' is a C expression to create an RTX representing the place where a library function returns a value of mode 'MODE'. It is possible to use a different value-returning convention for specific functions when all their calls are known, but for Thor we always return a RTX with the 'TOP'-register. Note that *library function* in this context means a compiler support routine, used to perform arithmetic, whose name is known specially by the compiler and was not mentioned in the C code being compiled. (*See 3.3.7.2, The 'thor-libgcc1.asm' file*)

'FUNCTION_VALUE_REGNO_P (REGNO)' represents a C predicate that is nonzero if 'REGNO' is the number of a hard register in which the values of called function may be returned. In Thor's case the macro is only true when 'REGNO' is equal to 'TOP'.

'DEFAULT_PCC_STRUCT_RETURN' is defined to '1' since all structure and union return values must be in memory.

'STRUCT_VALUE' could be defined as an expression returning an RTX for the place where the

address is passed. For Thor we return '0', i.e. the address is passed as an "invisible" first argument.

'FUNCTION_PROLOGUE (FILE, SIZE)' represents a C compound statement (usually a C block) that outputs the assembler code for entry to a function. The prologue is responsible for setting up the stack frame, initializing the frame pointer register (if there is one), saving registers that must be saved, and allocating 'SIZE' additional bytes of storage for the local variables. 'SIZE' is an integer. 'FILE' is a stdio stream to which the assembler code should be output. For Thor an additional word is allocated for the 'TOP'-register.

This macro is a good place to put code initializing the processing of a new function. Therefore, we allocate the array 'thor_top_offset_at', and set the variable 'thor_top_offset' to zero inside this macro. We use the GNU defined function 'xmalloc' when allocating memory, since it checks whether the desired amount of space is available and aborts if this is not the case. The array 'thor_top_offset_at' holds as many integers as there are labels in the compiled program. Its purpose is to keep track of stack offsets when jumping across basic blocks.

'EXIT_IGNORE_STACK' should be defined as a C expression that is nonzero if the return instruction or the function epilogue ignores the value of the stack pointer; in other words, if it is safe to delete an instruction to adjust the stack pointer before a return from the function. The value is tested only in functions that have frame pointers. On Thor we always need to keep track of the stack pointer, therefore the macro is tied to zero.

'FUNCTION_EPILOGUE (FILE, SIZE)' is replaced with a C compound statement (a block) that outputs the assembler code for exit from a function. The epilogue is responsible for restoring the saved registers and stack pointer to their values when the function was called, and returning control to the caller. This macro takes the same arguments as the macro 'FUNCTION_PROLOGUE'. On Thor a sequence of instructions 'RET', 'POP' and 'MTOS' is emitted in order to copy the callee's 'TOP'-reg to the caller's 'TOP'-reg, and then removing local variables. The 'RET' instruction's two delay slots come in use here and are filled with the two other instructions. We must also free the array 'thor_top_offset_at' and perform a check on the global variable 'thor_top_offset' to verify that its value is zero. If it is not equal to zero the stack has been corrupted somewhere in the function.

3.3.2.6 Addressing modes

'MAX_REGS_PER_ADDRESS' is a macro defining a number which is the maximum number of registers that can appear in a valid memory address.

'CONSTANT_ADDRESS_P (X)' is a C predicate that is '1' if the RTX 'X' is a constant which is a valid address. Sometimes one can define this macro directly to 'CONSTANT_P' which accepts integer-values expressions whose values are not explicitly known, such as 'symbol_ref', 'label_ref', and 'high' expressions and 'const' arithmetic expressions, in addition to 'const_int' and 'const_double' expressions. But for Thor we are more restrictive and allow just 'symbol_ref', 'label_ref' and 'const'.

'LEGITIMATE_CONSTANT_P (X)' is a C predicate that is nonzero if 'X' is a legitimate constant for an immediate operand on the target machine. On Thor this macro should be true whenever 'CONSTANT_ADDRESS_P (X)' is true plus the cases when 'X' is 'const_int' or

'const_double'.

We have two alternate definitions for each of the macros 'REG_OK_FOR..._P', which assume that the arg is a REG RTX and check its validity for a certain class. The usual definition accepts all pseudo regs; the other rejects them unless they have been allocated to suitable hard regs. The symbol 'REG_OK_STRICT' causes the latter definition to be used.

'REG_OK_FOR_BASE_P (X)' is a C expression that is nonzero if 'X' is valid for use as a base register. For hard registers, it should always accept those which the hardware permits and reject the others. Whether the macro accepts or rejects pseudo registers must be controlled by 'REG_OK_STRICT' as described above. This usually requires two variant definitions, of which 'REG_OK_STRICT' controls the one actually used. For Thor, in the strict case, we allow everything the macro 'REGNO_OK_FOR_BASE_P' accepts. In the unstrict case we also allow that 'X' is a pseudo register, apart from the cases the strict variant accepts.

'REG_OK_FOR_INDEX_P (X)' is a C expression similar to the previous, it returns nonzero if 'X' is valid for use as an index register. 'REG_OK_STRICT' selects the one actually used, as above. For Thor, in the strict case, we allow everything macro 'REGNO_OK_FOR_INDEX_P' accepts. In the unstrict case we also allow that 'X' is a pseudo register, apart from the cases the strict variant accepts.

'GO_IF_LEGITIMATE_ADDRESS (MODE, X, LABEL)' is a very important macro represented by a C compound statement with a conditional 'goto LABEL;' executed if 'X' (an RTX) is a legitimate memory address on the target machine for a memory operand of mode 'MODE'. We have split this macro into several simpler macros working as subroutines, to preserve readability. The names of those macros are: 'PC_OR_STACK_RELATIVE_P(X)', 'MEM_INDIRECT_RELATIVE_P(X)', 'STACK_RELATIVE_P(X)', 'MEM_INDIRECT_P(X)'. From 'GO_IF_LEGITIMATE_ADDRESS' the first two are called, and the other two are used from within those.

'PC_OR_STACK_RELATIVE_P(X)' checks if 'X' is a constant or satisfying 'STACK_RELATIVE_P(X)', which in turn verifies if 'X' is a legitimate stack relative address.

We must permit 'TOP' relative addresses ('TOP' as a base register). It is not possible to generate RTL with only one base register, so we must be able to handle stack indirect addresses as well. This is actually the same as the 'TOP' relative addressing, since our 'TOP'-register resides on the stack. All this means that we can permit all kinds of 'MEM' (PC- or stack relative) indirect addresses since 'symbol_ref'-expressions can be handled in the same way as stack references. A pseudo register is eventually transformed into the 'TOP'-register or a stack-position. So, like the 'TOP'-register, all pseudo registers are also equivalent to stack relative addresses.

3.3.2.7 Condition codes

'NOTICE_UPDATE_CC (EXP, INSN)' is a C compound statement to set the components of 'cc_status', which is a C structure containing information about flags in the machine, appropriately for an instruction 'INSN' whose body is 'EXP'. It is this macro's responsibility to recognize instructions that set the condition code as a by-product of other activity as well as those that explicitly set the codes. This C block makes use of the machine description facility called attributes (*see 2.2.4.2, Target Description Macros, '.h'-file*). If the condition codes are properly

updated for each instruction, the compiler may be able to remove certain instructions whose only task is to set the 'CC'-register (the 'SR'-register on Thor).

3.3.2.8 Assembler format

This section discusses various macros designed to control the assembler output, for example, the floating point format and register names. All the names of the macros dealing with these matters start with the character sequence 'ASM_'. Many of macros belonging to this section will not be described here because they are not very essential to our solution, since they deal with matters like the look of the comment character in the assembler etc. We consider macros not described here either belonging to this category or being self-explanatory.

There is a set of macros starting with 'ASM_OUTPUT_' which handles the output format of the various types in the C language. These macros are called when the compiler wants to swap data out to the memory and therefore the macros should also output certain assembler directives in advance of the data otherwise the assembler will not be able to interpret the data correctly. The directives given include; 'DATA' for various kinds of integer output both hex-style and ordinary, 'DATAF' for floating-point data and 'DATAS' for characters. There is another directive called 'REP' used for uninitialized data, it takes two "arguments" separated by a comma, the second for the number of words and the first for what to put in them (usually '0'). For example the directive 'L1 REP 0,5' means that 'L1' is a label to a uninitialized data area holding five words, each zeroed.

'ASM_GLOBALIZE_LABEL(FILE,NAME)' and 'ASM_OUTPUT_EXTERNAL(FILE,DECL,NAME)' are two macros called when the compiler wants to inform the assembler/linker that a label/variable is globally accessible respectively defined externally. There are two directives designed for this purpose; 'XDEF' and 'XREF'. For example, if one wants to access a global variable 'VAR', the compiler outputs 'VAR XREF' in the beginning of the assembler file. This way one can arbitrarily refer to 'VAR' in the assembler file without any complaints when assembling, these external references are up to the linker to resolve.

The macro 'PRINT_OPERAND (FILE, X, CODE)' handles all possible cases when operands to the assembler instructions are to be output. It calls in turn a C function with the same name but with lower case letters (for easier debugging) which is defined in file 'thor.c'. Here we disassemble and test the RTL-expression 'X' in several stages in order to find out what kind of operand it is. All addresses accepted as legal in macro 'GO_IF_LEGITIMATE_ADDRESS (MODE, X, LABEL)' should be taken into account here, and if an operand slips through the net of tests the compiler will ultimately crash. The readers can verify for themselves that all legal addresses by the means of 'GO_IF_LEGITIMATE_ADDRESS (MODE, X, LABEL)' are taken care of in this macro. When printing stack-relative addresses our own variable 'thor_top_offset' is of vital interest, since it reveals the additional offset one should add to the operand in order to get the correct stack offset. The parameter 'CODE' also helps the macro to output the correct stack offset in some cases. The contents in 'CODE' is the optional low-case letter found directly after the '%'-character in the *template output* field in some of the 'define_insn' patterns in the 'thor.md'-file. We have in our implementation used three letters for this purpose; 'z', 'y' and 'h'. 'PRINT_OPERAND_PUNCT_VALID_P (CODE)' is a C expression which evaluates to true if 'CODE' is a valid punctuation character for use in the 'PRINT_OPERAND' macro, in addi-

tion to the usual one '%'. We have defined one additional punctuation character '#' to use for instructions utilizing the 'define_delay' facility (currently used only in unconditional jumps, *see 3.3.5.2, Delay slot filling*), i.e. instructions that let the GNU CC fill their delay slots. If the 'CODE' parameter is tested as equal to character '#', it means that at most two 'NOP'-instructions are to be output. How many 'NOP'-instructions depends on the number of slots GNU CC succeeded to fill. The global function 'dbr_sequence_length ()' reveals this by returning the number of filled slots (0, 1 or 2 filled).

Table 6 Thor defined letters following the punctuation character '%'.

Letter	Meaning
'h'	Printing a 'CONST_INT' object in hexadecimal style (16#...).
'z'	Adding an offset of 1 to all stack-relative addresses.
'y'	Adding an offset of 2 to all stack-relative addresses.

'ASM_OUTPUT_ALIGN(FILE,LOG)' is a statement to output to the 'stdio' stream 'FILE' an assembler command to advance the location counter to a multiple of 2 within 'LOG' bytes. 'LOG' will be a C expression of type 'int'. Since the Thor assembler does not have any 'align' directive we have solved this by outputting a 'DATA 0' directive which implicitly will align the data.

'ASM_OUTPUT_SKIP (STREAM, NBYTES)' is a C statement to output to the 'stdio' stream 'STREAM' an assembler instruction to advance the location counter by 'NBYTES' bytes. Those bytes should be zero when loaded. 'NBYTES' will be a C expression of type 'int'. For Thor we output the 'REP' directive for 'NBYTES' words initializing them all to zero.

3.3.2.9 Miscellaneous

In this section we define various macros which do not fit in any other of the sections. Macros of little interest to the overall solution will not be discussed here.

With macro 'WORD_REGISTER_OPERATIONS' we force the compiler to work in words when different modes between operands in an operation occurs. (Must be defined, but it is not used on Thor since we only have word operations).

'Pmode' is an alias for the machine mode for pointers, which on Thor is equal to 'QImode'.

'FUNCTION_MODE' is an alias for the machine mode used for memory references to functions being called, in 'call' RTL expressions. On Thor this macro is defined to 'QImode'.

'NO_FUNCTION_CSE' should be defined if it is as good or better to call a constant function address than to call an address kept in a register. Since we do not want any calls via registers it is defined on Thor.

'NO_RECURSIVE_FUNCTION_CSE' should be defined if it is as good or better for a function to call itself with an explicit address than to call an address kept in a register. Analogous to the previous macro, it is defined for Thor.

3.3.2.10 Compiler options that are forced to be activated

'OPTIMIZATION_OPTIONS (LEVEL)' controls options that should always be active when compiling. Some machines may desire to change which optimizations are performed for various optimization levels. This macro, if defined, is executed once just after the optimization level is determined and before the remainder of the command options have been parsed. Values set in this macro are used as the default values for the other command line options.

For Thor we desire that the compiler omits the frame pointer and replaces it with the 'TOS'-pointer instead, this action is controlled by the option '-fomit_frame_pointer' which in turn sets the flag 'flag_omit_frame_pointer' internally in the compiler. Thus the flag is set for Thor. Furthermore we cannot allow the compiler to defer popping[1] the stack. This behaviour is beneficial on other machines when performing several calls to subroutines accompanied by preceding pushes of arguments, in this case one can delay the cleaning of the stack after the functions has returned with a humungous move of the stack-pointer when all calls are done. With Thor we cannot allow such operations and want to forbid it in all compilations. One enables this behaviour with the option '-fdefer_pop' which sets the flag 'flag_defer_pop'. This flag is always zeroed for Thor.

The macro 'OVERRIDE_OPTIONS' outputs warnings if an unhealthy mix of options are activated when compiling. Sometimes certain combinations of command options do not make sense on a particular target machine. One can define a macro 'OVERRIDE_OPTIONS' to take account of this. If defined, this macro is executed once just after all the command options have been parsed. If one wants to omit the frame pointer (which is always performed) a message is put on the 'stderr' stream informing the user that this is always done. The same thing is done if someone wants to defer popping (which is never allowed), then we point out that this cannot be done.

3.3.3 Machine description instruction patterns

To fully understand our 'thor.md'-file, it can be a good idea to first read *section 2.2.4.1 'The Machine Description file, '.md'-file'*. You can find a listing of the file in *APPENDIX A - 'Listing of machine dependent files'*.

As explained in the strategy section (*see 3.2, Strategy*), we use 'define_expand'-patterns to implement most of the standard names available to the RTX generating pass. The RTL-instructions thus created are then later recognized by 'define_insn'-patterns, which generate the actual assembler output. A few functions are directly implemented as 'define_insn'-patterns, handling both the RTL generation and the assembler output.

Each 'define_expand'-pattern that we have defined, normally implements a sort of three-address statement. The operation handled is given by the standard name of the pattern, and the RTL generating pass can send three general operands to the pattern. For example, the 'andqi3' standard name handles the “bitwise AND” operation. It is implemented as a 'define_expand'-pattern, which takes three general operands: the destination operand and the two arguments to the binary operator 'AND'. The pattern transforms this into three RTL-instructions. First a push instruction, then the instruction doing the operation, and finally a pop instruc-

1. With flag '-fdefer-pop' the compiler postpone several stack cleaning 'MTOS' instructions into a single instruction of that kind, with it's parameter equal to the sum of operands of all the postponed instructions.

tion. The example below illustrates the transformations done. It is representative for most patterns we have written.

The RTL generating pass wants to output the following three-address statement, by using the 'andqi3'-pattern:

```
op1 = op2 AND op3
```

This is transformed and emitted as RTL-instructions as:

```
tmp = op2                (PSH op2)
tmp = tmp AND op3        (AND op3)
op1 = tmp                (POP op1)
```

Where 'tmp' is our TMP-register.

Example 12 The job done by a typical 'define_expand'-pattern.

The 'define_expand'-patterns also does some more complicated decisions. If the operator is commutative, the operands may be exchanged. This is desired in some cases, because the operand used in the operator instruction (op3 in *Example 12*) must be simple enough for Thor to handle it. Thor's instructions only permits immediate and pc/stack relative addressing. Therefore, it is desirable that a complex operand (e.g. indirect address) ends up in the push operand (op2 in *Example 12*), because the 'define_insn'-pattern handling the push is built to cope with all kinds of operands. If both operands of the operators are complex, one of them must be reloaded via an additional pseudo register.

Almost everywhere in the 'define_expand'-patterns we use 'general_operand' as the predicate, since we must be able to handle all different cases. This is not true for the 'define_insn'-patterns, where we have introduced some other predicates. The two most commonly used predicates is 'tmp_register_operand' and 'simple_operand'. They are needed to catch the RTXs generated by the 'define_expand'-patterns. The 'tmp_register_operand' only accepts the 'TMP'-register. The 'simple_operand' accepts an operand simple enough to fit in a normal Thor assembler instruction.

The most commonly used constraints are listed in the table below. They usually correspond to the different addressing modes supported by Thor. Some of them are defined in the macros 'REG_CLASS_FROM_LETTER', 'CONST_OK_FOR_LETTER_P' and 'EXTRA_CONSTRAINT' (*see 3.3.2.4 'Registers and Register Classes'*), the others are standard constraints defined by GNU CC.

Table 7 Commonly used constraints in 'thor.md'.

Constraint	Meaning
'I'	An immediate operand, with a value that can be handled by an immediate assembler instruction. If the value lies outside the interval defined by 'I' it must be put in a 'DATA'-directive.

Table 7 Commonly used constraints in 'thor.md'.

Constraint	Meaning
'Q'	A pc or stack relative address. This also includes pseudo registers, since they will finally end up in a stack slot.
't'	The register class 'TOP_REG', i.e. the 'TOP'-register.
'Qt'	The 'Q' and 't' constraints are often used together, because the TOP-register resides on the stack and are therefore accessed with a stack relative address. The constraint 'Qt' is true whenever 'Q' or 't' is true.
'm'	A memory operand. Often this constraint comes after the 'Qt' constraint. In this way it catches the complex, indirect memory operands not caught by the 'Qt' constraint. When used with the predicate 'simple_operand' it catches every operand passed by the predicate except the 'TOP'-register. A peculiar thing is that when this constraint is used, it enables the reloading of illegal constants to a 'DATA'-directive. This reloading does not work if 'm' is combined, like the 'Qt', with another constraint.
'r'	Often used to represent the TOP-register where no other kind of register operand may occur. It would have been nice to be able to combine this with the 'm' constraint and use the combined constraint 'mr', but this would disable the reloading of illegal constants, as explained above.

Since Thor is a RISC processor, it does not support a large number of addressing modes but rather a restricted set, which seems to be the most natural one when working with a stack. Not all instructions support each addressing mode, but usually relative addressing is the most utilized one. The addressing modes available are:

1) *Relative addressing*. Which is usually split further into stack or PC -relative addressing. When addressing stack-relatively one refers to a word placed at a arbitrarily offset from top-of-stack, and performs the operation with the value at top of stack (the internal TOP register). If one uses the latter alternative one refers to data with an offset from the program counter.

2) *Indirect addressing*. The strategy here is to place the indirect address on the top of the stack and perform the operation via this address. This is the most advanced addressing mode one can use, and one must take care how the instructions are emitted since the indirect address used must be computed two instructions in advance (*see 2.1.3.2, Address Generation Stage*).

3) *Immediate addressing*. Here we use an immediate value directly and operate with it as usual together with the value at top of the stack (the internal TOP register).

4) *Register*. Some instructions, in particular the data move instructions, can work with the contents in Thor's registers directly, for example one can perform a jump by popping to the PC register.

3.3.3.1 An example of how a simple C program is handled

In order to enlighten the readers as to how the compiler works with the standard name patterns versus the nameless patterns, we here give an example of how a simple C program is compiled and what patterns are generated. When reading the rest of this section one should preferably keep an eye on the appendix where the listing of our '`.md`'-file is found (*see A.1, thor.md*).

```
void main(a,b,c)    /* 'main' takes args. otherwise the */
     int a,b,c;     /* whole prog. will be optimized    */
{                   /* away                             */
 loc:               /* A label                          */
  a = - b * c;      /* Some arithmetic operations       */
  goto loc;         /* Forcing the compiler to generate */
}                   /* an uncond.jump pattern           */
```

Example 13 Source code of the C program

The C program given above contains some simple arithmetic operations and an unconditional jump to a label. The program is compiled with the command line '`thor-gcc -S -da -O -nodelayed-branch example.c`'. The compiler options mean in turn; an assembler file is wanted (`-S`), dump files in every stage (`-da`), optimizations (`-O`) and finally that no delay slot filling is desired (`-fno-delayed-branch`). The reason for not wanting any delayed branch scheduling is that we prefer to have a very simple and clear example showing how the code is produced. The produced assembler file looks like this:

```
; Assembler file from Thor C Cross Compiler, version
1.0.0
; Generated by thor-gcc.
code    SECT 1,R,C
    DATA 0  ; align function
_main   XDEF
_main:
    MTOS -3 ; local vars(2) + TOP reg(1)
L2:
    PSH 5   ; S+5
    NOT -1
            ; PEEP - pop,push (dead)
    MUL 7   ; S+6
            ; PEEP - pop,push (dead)
    POP 5   ; S+4
    JR L2
    NOP
    NOP
    RET 3
    POP 7   ; copy TOP-reg to caller's TOP-reg
```

Statement:
pseudo1 = - b;
pseudo2 = pseudo1 * c;
a = pseudo2;
goto loc;

Standard name:
negqi2
mulqi3
movqi
jump

RTL-generation

```
define_expand "negqi2"
define_expand "mulqi3"
define_expand "movqi"
define_insn "jump"
```

RTX

```
(set
 (reg:QI 1 TMP)
 (mem:QI
  (plus:QI
   (reg:QI 2 TOS)
   (const_int 5))))

(set
 (reg:QI 1 TMP)
 (neg:QI
  (reg:QI 1 TMP)))

(set
 (mem:QI
  (plus:QI
   (reg:QI 2 TOS)
   (const_int 2)))
  (reg:QI 1 TMP))

(set
 (reg:QI 1 TMP)
 (mem:QI
  (plus:QI
   (reg:QI 2 TOS)
   (const_int 2))))

(set
 (reg:QI 1 TMP)
 (mult:QI
  (reg:QI 1 TMP)
  (mem:QI
   (plus:QI
    (reg:QI 2 TOS)
    (const_int 6)))))

(set
 (mem:QI
  (plus:QI
   (reg:QI 2 TOS)
   (const_int 1)))
 (reg:QI 1 TMP))

(set
 (reg:QI 1 TMP)
 (mem:QI
  (plus:QI
   (reg:QI 2 TOS)
   (const_int 1))))

(set
 (mem:QI
  (plus:QI
   (reg:QI 2 TOS)
   (const_int 4)))
 (reg:QI 1 TMP))

(set (pc)
 (label_ref 7))
```

Assembler output

```
define_insn "*movqi_psh_tmp"
define_insn "*negqi2"
define_peephole (pop-push)
define_insn "*mulqi3"
define_peephole (pop-push)
define_insn "*movqi_pop_tmp"
define_insn "jump"
```

Assembler

```
PSH 5
NOT -1
MUL 7
POP 5
JR L2
NOP
NOP
```

Figure 10 Example of RTL generation

```
MTOS 6  ; local vars(2) + 1 + pop_args(3)
```

Example 14 Assembler output of the discussed C program

As one can see the compiler has succeeded in removing two pairs of unnecessary data move instructions (*see 3.3.5.1, Machine-specific peephole optimizations*), both of the type that pop to the same temporary memory location as the next instruction pushes. The function return sequence as well as the function prologue are of the normal type. No RTL code are ever generated for the function return since they are handled in the macro called 'FUNCTION_EPILOGUE'. This is also partly true for the function entrances where allocation for local variables is taken care of in the macro called 'FUNCTION_PROLOGUE'. Therefore the readers should be aware of this fact when studying the RTL dumps from the example program, so no misunderstandings occurs.

An illustrative picture over the most important stages in the compiler are given in *Figure 10 'Example of RTL generation'*. This picture shows the work done on the two statements in the C block (*see Example 13, Source code of the C program*).

The first row in the picture (*statement*) shows how the compiler has disassembled the syntax tree into statements of low abstraction. We use a sort of C syntax to illustrate the statements. As one can see the multiplication and the negation action have been given a statement of their own, since the compiler has detected that the target machine must use two patterns to solve the first statement in the source code. At this stage the compiler uses so called pseudo registers when performing arithmetic and data moves. The third pseudo-statement is an unnecessary data move action, since the multiplication could write its result directly to the 'a' variable. These redundant data moves are very hard to avoid generating, but as one can see we have designed a special peephole pattern that will remove such moves.

The second row in the picture (*standard names*) shows which standard name the compiler has found to match each statement. As we see, in the '*RTL-generation*' row, the first three are statements designed as 'define_expand'-expressions and only the 'jump' standard name is a true 'define_insn'.

The *RTX* row shows all the RTL-expressions generated after the compiler has chosen the standard name. Observe that only the expression field in the RTL objects are listed in the picture. The *RTX* row's listings are taken from a '.greg' dump file, i.e. after the reload phase has been completed. Therefore the pseudo registers have been converted into machine registers and since we have given each of the registers a unique name one can see how the compiler utilize them in the listings in the *RTX* row. One should notice that the register 'TMP' is not a Thor register but merely a temporary 'TOP'-register used in the 'define_expand'-patterns (*see 3.2, Strategy*). In between *RTL-generation* and *RTX* various kinds of optimizations occur (if compiling with '-O').

The *assembler output* row shows which 'define_insn'-expressions in fact produce the assembler code. These can be of either the standard name category or the nameless category, where we have used the convention to use the character '*' in beginning of a nameless pattern's name. In our program there was only one RTL object that remained identical throughout the chain of passes, namely 'jump'. This is due to the fact that 'jump' is implemented directly as a 'define_insn'.

Before the actual assembler is produced the compiler scans through the program is search of

peephole optimization matches. In this case the compiler has found two suitable peephole transformations, both of the pop/push kind, where the 'pop'/'push' pairs simply could be removed without any change in the program's semantics. The last row in the picture (*assembler*) shows the real assembler code output from the RTL-expressions (*see Example 14, Assembler output of the discussed C program*).

3.3.3.2 Data move instructions

The standard names corresponding to data move patterns are 'movX', where 'X' stands for an arbitrary mode. The only standard names of this kind that we decided to support are 'movqi' and 'movqf', since quarter-word operations are the only ones needed. Note that here quarter-word means operating with 32 bits, not 8, since Thor itself is not byte addressable (*see 3.3, Solution*).

These patterns are not used solely in the RTL generation pass. Even the reload pass can generate move instructions to copy values from stack slots into temporary registers. When it does so, one of the operands is a hard register and the other is an operand that may need to be reloaded into a register. Therefore, when given such a pair of operands, the pattern must generate RTL which needs no reloading and needs no temporary registers. For example, if you support the pattern with a 'define_expand', then in such a case the 'define_expand' must neither call 'force_reg' nor any other such function which might generate new pseudo registers.

The only type of data moves produced by our compiler are different kinds of push- and pop-instructions, that move data to, or away from the top of the stack. There are four kinds of addressing modes to be concerned about when dealing with data moves. (*see 3.3.3, Machine description instruction patterns*). Observe that pop instructions cannot be used in immediate addressing mode. The idea when utilizing these instructions is to first push a word to 'TOP'/'TMP', then (optionally) perform an operation with its contents, and finally pop it back to somewhere in the stack. When dealing with pointers in the C language, which is often the case, one can see that usually some sort of indirect addressing is used. One should observe that the indirect address must be calculated two instructions before the push/pop-instruction, due to the pipeline design incorporated (*see 2.1.3.2, Address Generation Stage*).

Everyone that has studied the Thor instruction formats thoroughly will notice that there exists a limitation in terms of how big the offsets can be in relative addressing, but one can assume that the computer systems are built according to those limitations, and no concern need to be taken in these cases. Otherwise we would have been forced to use indirect addressing almost always. Exceeding these limitations will anyhow result in errors when loading the code into the target machine.

Finally, one should observe that these 'PSH'/'POP'-instructions discussed above are the ones corresponding to a register machine's load/store-instructions, and during RTL-generation other, "true" (normal), push/pop-patterns are generated. In a register machine these are the only ones operating on the stack, for example when pushing arguments before a call. These should also be handled when designing the nameless patterns.

3.3.3.2.1 Standard move patterns

The standard names 'movqi' and 'movqf' are implemented as 'define_expand'-patterns, which in turn generate a sequence of patterns which will be picked up by other patterns. Those

patterns are of the nameless kind, but they output the real assembler, and are thus more interesting. The function controlling this new generation of RTL objects is called 'emit_move_sequence' and is found in the file 'thor.c'. As predicate in those patterns 'general_operand' is used in order to allow every possible data reference. A macro 'DONE' is inserted after the function since we do not desire that the template itself should be emitted, just the sequence generated by the function. The macro inhibits this undesired behaviour by telling the compiler we have taken care of the RTL-generation that should be the effect of the 'define_expand' pattern. No constraints whatsoever are found in the expand expressions since we must be ready to receive every possible reference.

3.3.3.2.2 Nameless patterns handling data moves

Two groups among the nameless move patterns are segregated immediately; those which are true push/pop-templates, and those which are not (load/store-instructions interpreted as Thor's 'PSH'/'POP'-instructions). The true push/pop-patterns are recognized since one of their predicates is 'push_operand', which tests if the operand is of the desired type. There are two patterns of the push type, one for 'QF' and one for 'QI'-mode. In the rest of the patterns the predicate 'tmp_register_operand' is used, which tests if the 'TMP'-register generated from the expand patterns is utilized. 'general_operand' are used in both groups as their second predicate. Both groups have the same approach in terms of using the constraints in their second operand, one constraint for the immediate case, one for PC- or stack-relative addressing along with referencing via the 'TOP'-register and the last one addressing with a more general memory reference ('I'-, 'Qt'- and 'm'-constraint, *see APPENDIX A -, Listing of machine dependent files*).

Observe that when representing moves in RTL with floating-point data the immediate case is omitted, since there is no such instruction in the Thor architecture's instruction set.

In the "true" push case, the predicate in the pattern's first operand is a memory reference, and in the other patterns a register operand. This means that the first group picks up every RTL with a push operand as the first operand, and an arbitrary second operand, while the other group wants the 'TMP'-register as their first and likewise as their second operand. After the RTL code has chosen the right pattern, it is further examined and checked against the constraints, and depending on which addressing mode the RTL expression operates in, the correct assembler output sequence is chosen in the succeeding C block.

In these patterns we introduce a global variable, 'thor_top_offset', in order to keep track of the number of words pushed on the stack by normal, "true" pushes, i.e. the stack offset to the 'TOP'-register and the extra offset to add when addressing the local stack slots. It is incremented by one in the pattern dealing with the normal pushes, but also some other patterns update it.

The variable 'which_alternative' is a label handled by the compiler to indicate which constraint accepted the data reference. We use it to index an array of assembler strings, one for each constraint. There are several C help functions dedicated to assist in the assembler output and dealing with addresses so complex that no direct assembler string can be chosen right away. As one can see we are not using any conditions in our patterns.

3.3.3.3 Arithmetic operations

All the patterns handling arithmetic operations are designed in pretty much the same way as the

move patterns, with some minor differences between the operators, since, for example, subtraction is not commutative in the sense multiplication and addition are. This implies that the implementation of multiplication and addition are identical, but that is not entirely true either. Subtraction on the other side differ from division, since besides the usual “subtract”-instruction also “subtract reverse”-instructions are included in the instruction set of Thor. We found it most convenient to solve the arithmetic constructs in a way resembling the previous paragraph (*see 3.3.3.2, Data move instructions*), in terms of having a 'define_expand' accepting virtually everything, and in this stage splitting up and generating other RTL patterns that are picked up by the real 'define_insn'-templates.

The basic idea when performing an arithmetic operation, is to first push one operand on the stack, perform the operation and then ultimately pop the result to the desired position in the stack. Depending on the commutativity of the operation, there is a possibility to swap the operands if that makes it easier to generate code for it. This way of solving the problem makes the expand expressions responsible for generating code for the preceding push sequence and the succeeding pop sequence.

3.3.3.3.1 Standard arithmetic patterns

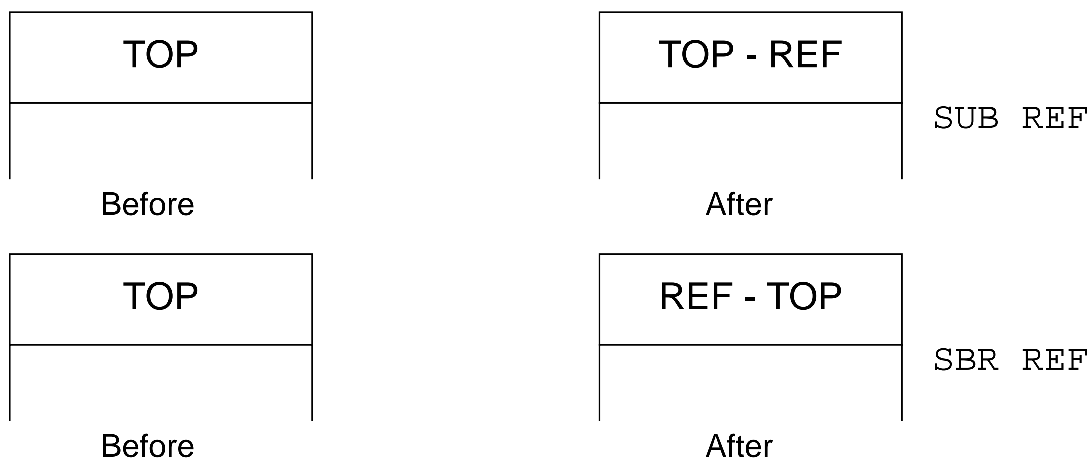
For each operation, there exist two modes as usual, 'QI'- and 'QF'-mode, and therefore it is sufficient with two 'define_expand'-expressions for each of the operators. According to our strategy, those should accept everything regardless of how complex the operands are. In the expand expression a call is made to a C function that handles the emitting of the loading from, and storing to the stack. A slight difference exists in the C functions called from every expand expression (apart from the different machine modes). For addition we must take care of a special case when a 'MTOS' (Move Top Of Stack) instruction is to be generated. Subtraction must decide which of the instructions (reversed or the usual) are the most beneficial in every generation. Multiplication is commutative and division is not.

3.3.3.3.2 Nameless arithmetic patterns

Since all of the nameless patterns are generated by the expand expression, and the 'TMP'-register is reserved for that reason, we have as predicate for one operand 'tmp_register_operand', which solely tests if the operand is the 'TMP'-register. As the second predicate we usually use 'simple_operand' which in turn checks that the operand is simple enough to be handled without difficulty by the output template. Immediate, memory and register are used as constraints for most patterns, since this is the most natural splitting of the possible cases if one keeps in mind the available addressing modes. For floating-point mode there are no immediate alternatives, as discussed earlier, and the compiler will take care of the necessary reloading for placing the immediate data in memory.

The output templates are pretty straight forward, with assembler strings for each of the possible constraints, and no modifying of the operands occurs as soon as the compiler has chosen the template. There are double sets of instructions for dealing with subtraction, one for subtracting the 'TOP'-register with a parameter and a corresponding one doing the reversed operation. The reason for having two sets of instructions is that a lot of swapping on the stack will probably occur if one set is omitted. The difference in the templates between normal and reversed is that both operands have been switched, and therefore also the corresponding predicates. The

'define_expand'-expression outputs the alternative which it thinks is most efficient. Unfortunately there is no instruction called 'SBRI', i.e. the immediate variant of reverse subtraction, and in this case reloading occurs as usual when no suitable constraint exists, i.e. the immediate operand is reloaded to memory via a 'DATA' assembler directive.



Example 15 Stack influences of the two versions of subtract (stack-relative)

Some special cases occur among the arithmetic constructs that must be dealt with exclusively. One case is when the compiler wants to perform an addition between the stack pointer and an immediate value and then storing the result in the 'TOP'-register. The output template then contains a hard-coded sequence which pushes the stack pointer and then add it with the immediate value.

Another special case is when the compiler desires to change the stack pointer value by adding an immediate value, i.e. performing a 'MTOS'-instruction. A new constraint controlling the immediate value is introduced here, since 'MTOS' is only found in 2a and 4a formats (*see 2.1.2, Architecture and instruction set*). Observe that we must decrement 'thor_top_offset' with a value equal to the operand to 'MTOS'-instruction. In both of these exceptions a predicate 'tos_register_operand' is utilized, which only allows the 'TOS'-register as an operand.

The pattern implementing the 'modqi' construct must be treated specially since the interpretation of the 'mod' operator seems to differ between GNU CC and the Thor processor. GNU CC's standard pattern 'modqi' means in fact the remainder of an integer division, and is thus not a true 'mod' in the way Thor defines it. The nameless pattern picking up the 'mod'-construct resembles the ordinary arithmetic operation in terms of using the constraints and predicates. but the output template is a hard-coded, five instruction sequence, doing the 'mod'-operation in the same way as GNU CC would do it.

3.3.3.4 Logic operations

Three logic instructions are included in Thor's instruction set namely: 'AND', 'OR' and 'XOR'. The two first instructions are found in two variants, one immediate and one for PC- and stack-rel-

ative addressing, but 'XOR' on the other hand has no immediate variant. The approach of implementing those instructions is very straight-forward since there are corresponding standard names that one can use. Previous paragraph (*see 3.3.3.3, Arithmetic operations*) should be enough as a guideline for the reader.

3.3.3.5 Negating and one-complementing

In Thor's instruction set there exists only one instruction we can use directly when implementing these instructions. The name is 'NOT' which performs a logical negation on each bit of the data on the stack top, and subtracts the sign extended immediate value in the parameter. When implementing the one-complement it turns out to be very easy, as everyone can understand, one just performs a 'NOT' with a parameter value of zero. The negating of a integer value reveals to be just as easy, all one have to do compared to one-complementing is to replace the parameter value with minus one for getting the effect of negating. The floating variant of negating is not supported by the instruction set, and the most straight forward way to solve this problem is performing a reversed subtraction with zero, in order to get the correct negated floating-point value according to the IEEE-754 standard.

The strategy to solve the operations, including moving to and from the stack, follows the paragraphs discussed earlier, i.e. first perform a push of an argument, then execute the operation and finally pop the result back into the stack. For all of the three standard patterns supported there is a nameless pattern to pick up the RTL operation generated by the expand expression, except for 'negqf2' where the operation is an ordinary arithmetic expression due to the lack of hardware support discussed earlier.

3.3.3.6 Condition code setting instructions

Instructions included in the instruction set, whose purpose are to test a data value for something and depending on the result set the condition code register, are the 'TEST'- and 'CMP'-instructions. In Thor two flags are usually affected, the zero flag 'Z' and the negative flag 'N'. An instruction changing the PC often follows after the condition code setting, which performs the change of the control flow depending on the outcome of the 'CMP'/'TEST'. The 'CMP' instructions are found in several variants, one for each signed/unsigned integer, one for floating-point compare and finally one for immediate integer. This instruction compares the data on the stack top with the data indicated by the effective address. The 'TEST' instruction is only found in one variant, which compares the data indicated by the effective address, treated as a two's complement integer, with zero. When trying to optimize as much as possible, there is often a possibility to omit the 'TEST' instruction, since several other instructions also sets the 'CC'-register, including for example 'PSH' instructions. In order to omit a 'TEST' the compiler must make sure that the 'CC'-register has its flags set correctly, if so the compiler can remove the 'TEST' (*see 3.3.5.3, Removal of unnecessary test/compare instructions*).

The design of the 'define_expand' expression for the 'test' standard name uses a slightly different approach than for example the arithmetic operations. Instead of generating brand new RTL templates from the expand, we only modify the operand a little bit, forcing it to be in a register using a function 'force_reg', and then let the pattern flow through to be picked up by a nameless pattern. This modifying occurs when the operand is not a stack relative reference. The predicate used, 'non_immediate_operand', tests if the operand is everything but a immedi-

ate reference (an immediate operand would be pretty obscure). The corresponding nameless pattern then uses 'stack_relative_operand' as a predicate, since we are now sure that stack relative addressing mode is the only possible one. Nothing special happens further on in this pattern, in the output template only the assembler string is found.

The implementation of the 'define_expand' expression for the 'cmpqi'/'cmpqf'-standard names resembles the 'define_expand' of 'TEST', in terms of using functions that forces operands to be laid in registers and then letting the template through, but in this case we also allow PC-relative addressing. Only the operating modes differs in the standard names 'cmpqi' and 'cmpqf', in every other sense they are identical.

As said earlier in this paragraph an unsigned variant exists for the compare instruction, and therefore special concern should be taken when deciding which variant to choose, the signed or the unsigned. To fully understand our strategy when generating compare and branch[1] sequences, one must study the instruction set and realize that there are no unsigned variants of the conditional branch instructions, but on the other hand in GNU CC there exist such standard names. The way to solve this problem for Thor is to generate the same branch instructions for both cases and let the compare instruction handle the choice between signed and unsigned. When generating the compare instruction one must look ahead to the following branch instruction to decide which compare variant to use. For that purpose we have invented a function called 'next_cc0_user_unsigned_jump_p' that returns true if the next RTL using the 'CC'-register is a unsigned branch RTL. Conclusively, there exist two nameless patterns designed to pick up the expand-expressions, one for the unsigned case and one for the signed. The condition field is used to separate the cases.

Since we know that the operand is forced out to a register by the expand expression if it is too complex, we can use the predicate 'simple_operand' in the nameless expressions. In the unsigned case there is no instruction supporting immediate values, and therefore a difference exists in the constraint field between the two cases, and reloading must occur when operating with immediate operands in the unsigned case.

In the output template we also handle things a bit differently from, e.g. the arithmetic approach. We must first push the operand as usual, then perform the compare, which in the unsigned case is done with the help of the 'CMPU'-instruction, and then clean up the stack. However, the stack cleaning can be postponed and put into one of the delay slots of the following conditional branch. The branch pattern must in some way be informed of this, and therefore we have invented a flag called 'thor_compare_need_stack_adjust' that will be set in these patterns. It is only the compare patterns that need this stack cleaning (an 'MTOS'-instruction), the test patterns, which are followed by the same kind of branch patterns, do not need any stack adjustments.

3.3.3.7 Control transfer instructions

In our implementation we support three different groups of standard names. The first group are the 'call' patterns which are, of course, constructed when the compiler detects a function call in the C source code and next are the 'jump'- and 'bXX'-patterns ('XX' stands for two letters) which

1. When we refer to the word 'branch' we mean the entire set of conditional branch instructions included in the instruction set. The same convention is used with the word 'compare'.

are matched in almost every other situation when a change of the program counter is needed. The last pattern is a very special standard name used for representing the 'switch' construct in C.

3.3.3.7.1 Calls

In our 'thor.md'-file we support four different standard names to handle different call situations. These are called: 'call', 'call_value', 'call_pop' and 'call_value_pop. The meaning of 'call' is a subroutine call instruction returning no value. Operand 0 is the function to call; operand 1 is the number of bytes of arguments pushed; operand 2 is the number of registers used as operands. 'call_value' is a subroutine call pattern returning a value. Operand 0 is the hard register in which the value is returned. There are three more operands, the same as the three operands of the 'call'-instruction (but with numbers increased by one). 'call_pop' and 'call_value_pop' are similar to 'call' and 'call_value', except used if defined and if 'RETURN_POPS_ARGS' is non-zero. They should emit a 'parallel'-expression that contains both the function call and a 'set' to indicate the adjustment made to the stack pointer. For machines where 'RETURN_POPS_ARGS' can be non-zero, the use of these patterns increases the number of functions for which the frame pointer can be eliminated.

All standard names have two constraint options, one for the usual call implemented by emitting a 'CALL' instruction followed by two 'NOP' (due to the delay slots), and one for indirect call where a C function, 'output_indirect_call', is called. The basic idea when performing an indirect call, is to pop the address lying on the top-of-stack to the program counter. Two things to keep in mind are first that the 'POP PC'-instruction, as well as other control transfer instructions, has two delay slots, and second that the program counter is a half-word pointer, since there are both short and full-word instructions. Therefore we perform a shift-left operation on the indirect address in 'TOP' in order to get a half-word address.

The 'call_value'-pattern demands a 'MTOS'-instruction after the call to clean up the stack, and due to this problem there is a special peephole pattern defined to match if the 'MTOS' instruction is succeeded by another 'MTOS' instructions (*see 3.3.5.1, Machine-specific peephole optimizations*).

3.3.3.7.2 Branches

In Thor, as well as most other machines, there exist two kinds of branching instructions: one for absolute PC-relative jumps and set of instructions handling branches when a condition has to be true to get permission to jump.

The set of conditional branches does not include any corresponding unsigned variants. Instead the architecture serve us with an unsigned compare instruction that sets the 'CC'-register, assuming the test occurred with unsigned integers (*see 3.3.3.6, Condition code setting instructions*). In GNU CC one must include the unsigned conditional branch standard names in the '.md'-file, otherwise the compiler will not build. But in our case we output the same branch instruction in both cases and delegate the responsibility to earlier patterns ('cmpqi'/'cmpqf') for generating the correct version of the compare instruction.

The unconditional jump instruction, 'JR' (Jump Relative), is pretty straight forward to implement, and we have chosen to have the standard name 'jump' as a 'define_insn' that directly outputs the assembler. The array 'thor_top_offset_at' is also updated here, by storing the

offset 'thor_top_offset' in the array indexed by the jump target's internal number in the compiler. In principle this number is the number of preceding basic blocks in the entire program flow.

The solution for the conditional branches is to have a 'define_expand'-expression for each and every standard name (including unsigned variants), and just letting the template flow through changing nothing whatsoever. Then we have a mutual nameless pattern appropriate for any existing conditional branch RTL. This pattern calls a C function which decides what kind of branch it has received and outputs a suitable assembler string. This function also updates the array 'thor_top_offset_at' in the same way as for the unconditional case. If the branch was preceded by a 'CMP'-instruction one must insert stack adjusting code in the branch's delay slot. Since we have invented a suitable flag in the compare patterns which is set when pushing occurs, we only check its value to determine if stack adjusting is needed, i.e. a 'MTOS 1' or 'MTOS 2' in one of the delay slots. If not so, two 'NOP'-instructions are output.

The unconditional 'JR' instruction of course has two delay slots too, but we let the GNU CC's delayed branch facilities handle the filling of those (*see 3.3.5.2, Delay slot filling*). If it finds independent instructions in association with the jump, the compiler rearranges the code and moves two of those into the jump's slots, and if this succeeds no loss in terms of program execution occurs due to the unnecessary instructions. In the conditional jump's case we found it very hard to make use of this facility due to incompatibility between the semantics of our 'TMP' register and GNU CC, and possible bugs in the GNU CC's source code.

Since we are always able to fill one of the delay slots (in the conditional jump case) we did not make a true effort to make the GNU CC's delay slot filling algorithms work, further on we do not think there are so many occasions where the second delay slot can in fact be filled.

3.3.3.7.3 The 'casesi' standard name

There is a standard name 'casesi' designed to construct a dispatch table, including bounds checks. If the programmer writes a 'switch'-statement in C the compiler can make use of this pattern if it makes a significant improvement (faster code).

If the 'switch'-statement is very sparse, i.e. the entries are located far away from each other, the cost of implementing the statement as a dispatch table can be quite large (a lot of unused entries taking code space), in this situation the compiler may choose to implement the 'switch' statement as a branch-tree instead.

This pattern takes five operands:

1) The index to dispatch on, which originally had mode 'SImode'. We found ourselves forced to make a patch in the source code (*see 3.3.6, Changes in the source files of GNU CC*) to change this hard-coded mode to 'QImode', since in our strategy the 'QI'-mode equals the word-size.

2) The lower bound for indices in the table, an integer constant.

3) The total range of indices in the table, the largest index minus the smallest one (both inclusive).

4) A label that precedes the table itself.

5) A label to jump to if the index has a value outside the bounds. (If the machine-description macro 'CASE_DROPS_THROUGH' is defined, then an out-of-bounds index drops through to the code following the jump table instead of jumping to this label. In that case, this label is not actually used by the 'casesi'-instruction, but it is always provided as an operand.)

The number of elements in the table is one plus the difference between the upper bound and the lower bound.

Most of the work in this pattern is made in a C help function, but some things in the pattern are worth noticing. The pattern has three constraints handling the cases when the index operand has different addressing modes: immediate, stack- or PC-relative including 'TOP'-references, and a complex operand. The function 'output_casesi' does the rest of the job including range checking, outputting of a compare instruction, and the indirect jump via the table. A total of 13 instructions is emitted from this pattern, not counting the dispatch table entries.

3.3.4 Attributes

We have chosen to have four attributes to help us dealing with additional features in the compiler. The names of the attributes are in turn: 'type', 'length', 'ok_for_delay_slot' and 'cc'.

The first attribute does not do anything useful by itself, since all it does is to put a label on what class of patterns a particular pattern belongs to. Say, if the pattern performs some sort of arithmetic operation the type-attribute 'arith' is set. The 'length'-attribute tells how many assembler-instructions are emitted from each pattern. The 'ok_for_delay_slot'-attribute tests if a pattern is suitable for inserting into a delay slot in terms of belonging to an allowed class and that the number of instructions the pattern represents does not exceed a limit. Finally, the 'cc'-attribute returns a string which represents certain actions to be taken in the macro 'NOTICE_UPDATE_CC'. This macro essentially decodes a pattern, and if necessary, updates the global structure 'cc_status' which contains information on how flags in the condition code register are set.

3.3.5 Optimizing the code

Most optimizations GNU CC performs are independent of the target, and thus remain unaltered among the various existing ports of GNU CC. Optimizations belonging to this category are all loop optimizations, constant propagation, instruction combination, strength reduction, deletion of unused code etc. Unfortunately, some common code enhancing operations must be expressed in a machine dependent fashion, since some of them depend on the hardware itself (*see 3.3.5.2, Delay slot filling*). One of the most used techniques to enhance code, *peepholes*, must also be handled target dependently. GNU CC has developed ways to describe these optimizations, usually in the '.md'-file. Each of the following sections will discuss implemented techniques for code enhancement in the Thor port.

3.3.5.1 Machine-specific peephole optimizations

Peephole optimizations are by far the most common code enhancing action existing in compilers today. This is due to the fact that the possible optimizations themselves are fairly easily detected and the gains obtained by them are usually quite large. The idea is to recognize special instruction

sequences being generated, and replace (alternatively remove) the sequence by another, more efficient, one. The possible peepholes can be expressed rather directly in the '`.md`'-file, using a syntax resembling '`define_insn`'-patterns (*see 2.2.4.1, The Machine Description file, '.md'-file*).

The most frequently used peephole in programs for Thor occurs when a sequence doing storing and loading (or loading and storing) from the same stack slot is generated. Such a sequence can be removed right away since the program's semantics remains the same. Other peepholes recognize far more complicated instruction sequences, and replace them with an equivalent and shorter sequence. Here follows a table which covers all instruction sequences we recognize and modify.

Table 8 Peepholes implemented on the Thor port

Sequence recognized	New Sequence	Comment
`PSH Loc` `POP Loc`		A '`PSH`' followed by a '`POP`' to the same location/register. Both instructions are removed.
`POP Loc` `PUSH Loc`		Same sequence as above but the order is switched. Both instructions are removed. The '`Loc`' reference must here be temporary, i.e. the pseudo register must die.

`PSHR TOS`
`ADDI Val1`
`ADDI 1`
`POP Loc`
`PUSH Loc`
`ADDI Val2`
	`PSHR TOS`
	`ADDI Sum`
		When the address of a specific position in the stack is needed, a frame pointer reference will be generated in the RTL. After elimination of the frame pointer the sequence will contain several instruction which can be combined. '`Sum`' in second column is equal to '`Val1+Val2+1`'. The '`POP`'/'`PUSH`' pair is removed.

`PSHR TOS`
`ADDI Val`
`ADDI 1`
	`PSHR TOS`
	`ADDI Sum`
		Another case not covered by the previous peephole. '`Sum`' in second column is equal to '`Val+1`'.

`PSH Ref1`
`PSH Ref2`
`MTOS 1`
`NOP`
`POPX Ref3`
	`PSH Ref4`
	`MTOS 1`
	`PSH Ref1`
	`POPX Ref3`
		If a simple '`PSH`' instruction is located previous to a '`POPX`' instruction, the '`PSH`' can be moved into the delay slot of the '`POPX`'. '`Ref4`' in second column is equal to '`Ref2`' incremented with one, if it is a stack reference.

`POP Loc`
`PSH Loc`
`MTOS 1`
`NOP`
`PSHX Ref`
	`MTOS 1`
	`NOP`
	`PSHX Ref`
		A '`POP`' followed by an '`PSHX`' (indirect push) can be simplified if the '`POP`' location is used as the address in the '`POPX`' instruction. The location must be temporary, i.e. the pseudo register must die.

Table 8 Peepholes implemented on the Thor port

Sequence recognized	New Sequence	Comment
POP Loc PSH Loc CMP Ref	CMP	A 'POP' instruction followed by a 'CMP' instruction, where the 'POP' refer to the same temporary location as the first 'PSH' of the 'CMP'. In this case the 'POP'/'PSH' pair can be removed.
POP Loc1 TEST Loc2 JRXX NOP NOP	TEST Loc2 JRXX POP Loc1 NOP	A simple 'POP' followed by a 'TEST' and a conditional jump can often be simplified. Sometimes the 'TEST' may even be omitted, if the condition code register is already set correctly.
CALL fun NOP NOP POP x MTOS -1 MTOS Size1	CALL fun NOP NOP POP x MTOS Size2	Peephole sequence due to 'call_value' pattern. 'Size2' in the second column is equal to 'Size1 - 1'.

If more direct information of the implementation is desired we recommend the reader to consult the last part of the 'thor.md'-file listing, found in the *APPENDIX A - 'Listing of machine dependent files'* part *A.1*, where all the 'define_peep'-patterns are found.

3.3.5.2 Delay slot filling

Every control transfer instruction in the instruction set has two delay slots to be filled with either no-operation instructions or independent ones taken from the destination or a place prior to this instruction. We fill some of the control transfer instructions manually since the same sequences of instructions always occur. This is the case concerning 'CALL', 'RET' and conditional jump-instructions (*see 3.3.3.7, Control transfer instructions*), so the only ones left to be handled are the unconditional jumps.

There is a tool in GNU CC called 'define_delay', which should assist when trying to fill the slots with something useful (*see 2.2.4.1, The Machine Description file, '.md'-file*). We use this tool when handling the delay slots of the 'JR'-instruction. The 'define_delay'-attribute uses the attribute 'ok_for_delay_slot' (*see 3.3.4, Attributes*) when verifying if an instruction is legal to insert in a delay slot. The compiler then tries to find preceding instructions or instructions from the jump target satisfying the predicate, and if it does not succeed 'NOP'-instructions will be inserted after the jump. The macro 'PRINT_OPERAND' is responsible for checking if the compiler has filled the slots, and if not emit 'NOP' instructions instead.

Unfortunately we have found some bugs in GNU CC when dealing with the 'define_delay'-attributes. It seems that when using this facility on two delay slots we are out on untested territory, none of the earlier ports have ever used this feature on more than one delay slot. We were forced to make a patch in the source code due to the bugs (*see 3.3.6, Changes in the source files of GNU CC*).

3.3.5.3 Removal of unnecessary test/compare instructions

When compiling with optimizing flags activated, the compiler scans the program to remove unnecessary 'TEST'-instructions. Since the compiler knows at every point in the program how the flags in the condition code register is set, some 'TEST'-instructions may be redundant if the tested word has already set the 'CC'-register earlier in the program, and no instructions have influenced it since. A simple example of this is when a word is pushed on the stack, popped back to the destination, and then immediately followed by a test-instruction (referring to the same location as the previous push or pop). In this case the test is redundant since pushing the stack also sets the condition code. But if the 'CC'-register is not set appropriately the compiler cannot remove the test.

How each pattern affect the 'CC'-register is determined by the 'cc'-attribute in association with the macro 'NOTICE_UPDATE_CC' (*see 3.3.4, Attributes and 3.3.2.7, Condition codes*).

3.3.6 Changes in the source files of GNU CC

When we first started with this project we had as a goal that we should avoid any changes in the source code as much as possible, because each alteration in the source code makes the compiler more incompatible with future releases of GNU CC. A port totally free of changes in the GNU CC source code would mean that one continuously could update the compiler to the latest GNU CC release and benefit from all improvements and bug fixes in these versions. This goal soon proved to be unreachable due to many causes. First, we discovered some pure bugs in the source code that would had been impossible to leave unfixed. Second, we encountered some difficulties due to GNU CC's preferences for registers which also required source alterations. Finally, our semantics of the 'TMP' register seems not to go along with GNU CC's way of working (*see 3.2, Strategy*).

We felt compelled to make a total of ten changes in GNU CC's source code, none of these considered superfluous. In order to make it easier for the interested reader to find the exact locations of these fixes we have produced diff[1] files between the changed files and the original ones. These diff files are listed in *APPENDIX B - 'Diff files of the changes in GNU CC source'*. The following two sections will thoroughly describe the changes and our intentions with them. In almost every alteration we have done, we have used the convention to introduce a new macro that is tested in the source with the preprocessor construct '#ifdef', and if the macro is defined an altered sequence of C statements are executed. The line references in the text refer to the modified source files. These new macros all start with the character sequence 'THOR_', in order to simplify detection of the changes. The macros are defined in the file 'thor.h'. In this way, one can use the altered source files to build a compiler for yet another target, since none of the macros are defined for the other possible target machines.

3.3.6.1 Bugs

This section convey, in our opinion, faulty programming from the people responsible for GNU CC.

1. All UNIX systems are equipped with a command 'diff', which produces a list of all inequalities between two text files.

1) In the file 'reload.c' (function 'find_reloads_address', line 4353): We have introduced a new macro 'THOR_RELOAD_CHECK_FOR_CONST'. This macro enables code to correct a problem with 'subst_reg_equivs'. This function replaces pseudo-registers with an equivalent constant expression. We found that the result in some special cases can be a '(PLUS (SYMBOL_REF, CONST_INT))'-expression without the 'CONST' in front of it. Function 'find_reloads_address' tries to correct invalid addresses in many ways, but fails to correct this case. The code added finds these cases and corrects them by adding a 'CONST' constructor in front of the RTL-expressions. Without this patch the compiler will crash once in a while.

2) In the file 'stmt.c' (function 'expand_end_case', line 4891): When using standard RTL-name 'casesi' we are tied to use 'SImode' as machine mode for an operand in the RTL-expression. Since we always use 'QImode' as a standard mode in Thor we are forced to make the compiler use 'QImode' when compiling for Thor. We have introduced the new macro 'THOR_CASESI_QIMODE' in order to select the right mode to use. We believe that this hard coded machine mode might come to irritate other people in their porting tries if they are working with machines differing form the intended ones (32 bit and byte addressable).

3) In the file 'reorg.c' (function 'update_block', line 2241): In Thor we have two delay slots for every flow-control instruction, and we use the GNU CC feature 'define_delay' in order to fill these delay slots when doing an unconditional jump (for conditional branches and calls we found it very hard to use 'define_delay' since GNU CC cannot keep track of how the stack-pointer changes when performing eager delay slot filling). Here we have found a possible bug in GNU CC when using two delay slots. Function 'update_block' could be called with a second argument 'where' that points at an instruction which has been deleted. Therefore the compiler will crash once in a while, when 'update_block' calls 'emit_insn_before' which in turn calls 'add_insn_before' which fails when given a pointer to a deleted instruction. We suppose that we are the first to suffer from this bug since it is very rare that a machine has two delay slots. In order to avoid this bug we have changed the direct call to 'emit_insn_before' in 'update_block' to call 'emit_insn_after' whenever the parameter 'where' contains a pointer to a deleted instruction. We do not think this patch will interfere with any of the other actions GNU CC makes. We have invented a new macro 'THOR_DELAY_SLOT_FIX' to control the change.

4) In the file 'function.c' (function 'fixup_var_refs_1', line 1937 and 1986): Here we have found a rather serious bug that will affect every port available, not only ours. In the source code an assignment to the variable 'x' occurs with the value given by the function 'single_set'.

Originally the function is called with the parameter 'PATTERN (insn)', where 'PATTERN' is a macro designed to work on variables of type 'rtx'. If you study the definition of function 'fixup_var_refs_1' you will be soon convinced that the call to the function should be with the variable 'insn' solely, not with the surrounding macro. We have defined a macro 'THOR_FIXUP_VAR_REFS_1' that will modify the two function calls on line 1937 and 1986. The bug is only apparent when optimizing, since the function 'fixup_var_refs_1' is used only in that case. C programs suffering from the bug look like the example given in *Example 16*.

```
main()
{
  int a;
  a = 3;
  fun(&a);
}
```

Example 16 Program activating the bug in file 'function.c'

If you have an auto variable that you use as a destination target in an assignment and later in the code you take the address of this variable, the incorrect call to function 'single_set' is issued. There might be other C code sequences that suffer from the bug, but this is how we first found the bug. The effect of the bug, in the example given, is that an extra reload via a pseudo register is generated. The function 'single_set' will always return false, and the compiler gets uncertain about what it dares to do with the variable. Therefore the compiler will unnecessarily copy the variable's address once more to a pseudo register, which in our case will be translated to a stack slot. We believe that the Thor port suffers more from the bug than most of the others due to the lack of registers.

3.3.6.2 Thor adaptations

This sections deals with the source code alterations made to enhance code quality for our port to Thor. One could build the compiler without some of them, but certain strange behaviour may then arise.

1) In the file 'stmt.c' (function 'expand_end_case', line 4784): A new macro 'THOR_CASE_VALUES_SPARSENESS' has been introduced. It is now used instead of the value 10 when making the choice between a jump-table and a branch-tree. The macro gives a limit for the sparseness in the jump table. For example, if we have 8 different cases and 'THOR_CASE_VALUES_SPARSENESS' equals 10, we will permit jump-tables with a maximum of 80 entries, before choosing a branch-tree. We

considered, after some calculations, that a value of 4 would be more suitable for Thor than the old hard coded value.

2) In the file 'reload1.c' (function 'init_reload', line 386): When the compiler wants to find out how many indirect spill levels that are allowed, it checks how many 'mem' levels that are allowed with an RTX containing a register number equal to 'LAST_VIRTUAL_REGISTER' + 1. For Thor this will not work, therefore we have introduced a new macro named 'THOR_HARD_FRAME_POINTER_INIT_RELOAD'. This macro controls, when compiling for Thor, that the register number 'HARD_FRAME_POINTER_REGNUM' is used instead.

3) In the file 'jump.c' (function 'jump_optimize', line 437): The jump2-pass is performing various reorganizations of jump/branch instructions and their labels when compiling with optimizations. While doing this it will remove death-note information about variables and this is desirable when compiling for most machines. For Thor this removal of death information will strongly reduce the number of peephole optimizations that can be done. Therefore we have introduced a new macro called 'THOR_PRESERVE_DEATH_INFO_REGNO_P' which is inspired by the now obsolete macro 'PRESERVE_DEATH_INFO_REGNO_P' used by Intel's 80x86 port. First we tried to awaken the latter macro but it was impossible to build the compiler since the code enabled by the macro did not work any more, so we introduced this new macro with the same functionality as the old one and replaced its occurrences in the file 'jump.c', but only here since it was only the effect of preserving death information marks we desired. We can now more efficiently make use of our peephole optimizations.

4) In the file 'function.c' (function 'assign_parms', line 3583): When entering a function there is a high possibility that GNU CC wants to do a copy of all parameters to registers. In Thor's case this will result in very inefficient code since there are no registers in the machine. All the copy instructions will therefore be redundant and undesired. We have inhibited this behaviour by introducing a new macro named 'THOR_NO_INITIAL_REG' directly tied to 1, and inserting it into the 'if'-statement on line 3583. The result of this is that no initial copy instructions will be inserted in the beginning of a function.

5) In the file 'configure' (line 2274) 'config.sub' (line 135): We have added a target 'thor' to be able to configure and build a cross-compiler for Thor. These files are used when the compiler is built and all the possible target machines are listed in them. Since our port is not included in the official GNU CC release, we were forced to make this patch.

3.3.7 Auxiliary files in the compiler environment

These following sections discuss files in the compiler environment necessary for the compiler to

build or work correctly. All files mentioned in these sections are listed in *APPENDIX A - 'Listing of machine dependent files'*.

3.3.7.1 The 'crt0.asm' file

This file is necessary when linking object files in order to get an executable file. The file contains start-up information concerning, for example, initial values of the top of stack pointer and the program counter. The script and make files used when building the compiler compiles the 'crt0.asm' file itself in the latter stages in the building process in order to get an object file. This object file is in turn used by the linker each time one links a program. The programmer does not need to bother about this file while developing software, since it is implicitly linked to the executable file.

3.3.7.2 The 'thor-libgcc1.asm' file

When programming in C it is pretty usual that one uses the unsigned variants of the available types, and since the Thor processor does not directly support, for example, division with unsigned integers, the compiler must solve this problem in some other way. GNU CC's approach to this problem is to allow the designers of the compiler to define implicit library functions handling these operations. With the word “implicit” we mean that the programmer cannot access these functions directly[1], only the compiler can generate calls to the functions if it detects that a certain operation is not supported by the architecture. GNU CC has defined numerous implicit library functions which covers the most frequently desired operations. If the compiler wants to perform an unsigned division it just calls the corresponding function which is supposed to perform the computation for the compiler. These functions are written directly in target machine assembler, and like the previous file it is compiled during the building phase of the compiler and the produced object file is then used by the linker when necessary.

Table 9 Implicit library functions

Function	Explanation

`___udivqi3`	An unsigned 32-bit division. Divides the numerator by the denominator and returns the unsigned result. A C function prototype would look like this:
	`unsigned int`
	`__udivqi3(unsigned int numerator, unsigned int denominator)`

`___umodqi3`	An unsigned 32-bit modulo. Calculates the modulo[a] from the numerator and the denominator with the help of '`___udivqi3`' and returns the result. A C function prototype would look like this:
	`unsigned int`
	`__umodqi3 (unsigned int nominator, unsigned int denominator)`

a. The modulo defined by: mod (a,b) = a - (a DIV b) * b

1. This is not entirely true since the programmer may call, for example, the externally defined function, '`__udivqi3`' (two underscores in the beginning of the name), and when the compiler outputs assembler code for the program it appends an extra '_' in the beginning of the function's name. In the produced assembler code there now exists a call to a function '`___udivqi3`' and since this is the exact name of one of the implicit library routines the linker is able to resolve this external reference. Observe that this is not the recommended way of programming.

For Thor we currently support two implicit library functions, as seen in *Table 9*.

3.3.7.3 The 'as' and 'ld' script files

Unfortunately one is not able to easily select the name of the assembler and the linker which one desires to use, since the use of 'as' and 'ld'[1] are hard-coded in GNU CC's source. Since we are developing a cross-compiler we naturally cannot use 'as' and 'ld'. The assembler we utilize is invoked by the command 'assemble' and the linker by 'link'.We have discovered that other ports not using the UNIX assembler/linker have solved this problem by overriding the command names and writing scripts with the same names as the UNIX assembler/linker, and this also turned out to be the strategy we decided to follow. One tricky part of solving the problem in a manner like this is to make all options to the compiler to work along with the assembler/linker, for example the UNIX assembler/linker does not use the same characters as options as our assembler/ linker. Moreover, the assembler file suffix, used in UNIX systems, '.s' is hard-coded in GNU CC's source code, and this raise trouble since the Thor assembler uses the suffix '.asm'. The script files are also responsible to place the object files in the directories where the assembler and linker take for granted to find them. All these problems, and more, are solved in these script files, and as script files in common these turned out to be quite hairy to read.

One should observe that there exists a possibility to pass options directly to the assembler when compiling, by using the option '-Wa,OPTION', where 'OPTION' stands for an arbitrary number of options separated by commas, so if the programmer wants to access a special feature in the assembler this is always possible, even if GNU CC does not have an appropriate corresponding option. There also exists a similar option for passing options to the linker, '-Wl, OPTION', which works exactly as the flag '-Wa'.

3.3.7.4 The 't-thor' file

This file contains extra rules and information to the makefile of GNU CC ('Makefile.in'). Additional paths and directory names used by the Thor assembler and linker are also included in this file.

3.4 Outcome

In these sections we critically discuss how far we reached in developing the compiler, and we investigate the quality of the code generated.

The main thing to observe is the fact that the compiler seems to work properly. When given a C source program, it outputs a correct assembler file. However, if one studies the assembler code a little closer and tries to estimate how well it compiles a program, one discovers some things that might have been done in a better way. The main reason for this non-optimal code is GNU CC's preference for registers, which will be dealt with in the first section below.

We will also make a comparison with an Ada compiler that is specially designed for Thor.

3.4.1 GNU CC's preference for registers

When compiling programs with our port of GNU CC one notices once in a while that the compiler delivers code which contains unnecessary copy instructions, especially when compiling with opti-

1. These are the names of the standard UNIX assembler and linker.

mizing options. This behaviour is due to the fact that GNU CC is designed for register machines, and thus not very suitable for stack machines like Thor.

The compiler seems to think that significant gains are made if data are kept in registers as much as possible. This behaviour is more obvious when optimizing. Even though there are ways to access data by indirect addressing, the compiler prefers to reload the data via registers. We find this discrimination of the other possible addressing modes a little bit peculiar, since even in a register machine there exists a limit when no gains are made by reloading data to register, for example if the data word is only used once.

On Thor the drawbacks of this uncontrolled preference for registers are obvious since there are no registers in the architecture. The register preferences cause unnecessary stack slots to be allocated in certain programs, where data swapping then occurs. Some of the data moves can be removed, thanks to the peephole optimizations, but most of them are impossible to influence. Since this behaviour is heavily encouraged in optimized mode, this can cause some unexpected results, making the unoptimized version of a program faster than the optimized one. These situations occur, for example when calling a function with reference parameters. The compiler can manage with almost no extra stack slots thanks to the use of indirect addressing, which on Thor is absolutely the smartest way of producing code. Unfortunately the compiler will allocate more stack slots when optimizing the code, in examples like that. The following example will show a typical C code sequence where this stupid behaviour is most apparent:

```
void fun(a,b,c)
     int *a,*b,*c;
{
  (*a) = (*b) * (*c);
}

main()
{
  int a,b,c;
  fun(&a,&b,&c);
}
```

Example 17 Troublesome C code when optimizing

The function 'fun' in the example above, uses reference parameters which are handled with indirect addressing pretty straight forward when compiling unoptimized. Here follows the assembler extract (unoptimized) of the function 'fun'.

```
_fun:
    MTOS -2      ; local vars(1) + TOP reg(1)
    PSH 5        ; S+5
    MTOS 1
    NOP
    PSHX 0
    POP 2        ; S+1
```

```
    PSH 4          ; S+4
    MTOS 1
    NOP
    PSHX 0
    MUL 2         ; S+1
    PSH 4         ; S+3
    MTOS 1
    NOP
    POPX 0
L1:
    RET 2
    POP 6         ; copy TOP-reg to callers TOP-reg
    MTOS 5        ; local vars(1) + 1 + pop_args(3)

    DATA 0        ; align function
```

Example 18 Unoptimized assembler extract

When compiling the same C program with optimizing the following code is produced.

```
_fun:
    MTOS -6       ; local vars(5) + TOP reg(1)
    PSH 7         ; S+7
    POP 6         ; S+5
    PSH 8         ; S+8
    POP 5         ; S+4
    PSH 9         ; S+9
                  ; PEEP - pop,push base address (dead)
    MTOS 1
    NOP
    PSHX 0
    POP 2         ; S+1
    PSH 4         ; S+4
    MTOS 1
    NOP
    PSHX 0
    MUL 2         ; S+1
                  ; PEEP - pop,push (dead)
    PSH 6         ; S+5
    MTOS 1
    NOP
    POPX 0
    RET 6
    POP 10        ; copy TOP-reg to callers TOP-reg
    MTOS 9        ; local vars(5) + 1 + pop_args(3)
```

```
        DATA 0      ; align function
```

Example 19 Optimized assembler extract

When comparing the two versions of the function 'fun', certain things are worth pointing out:

- The optimized version utilizes five stack slots compared to only one in the unoptimized case.
- A total of 23 instructions is produced in the optimized version compared to only 19 in the unoptimized case, despite the fact that the optimized program succeeded in removing two pairs of unnecessary push/pop pairs.
- The additional four instructions in the optimized version are redundant since all they do are to move data from one stack slot to another. Both versions use the same addressing mode, i.e. indirect addressing, so no difference exists between the two versions in terms of accessing data.

Now is seems that compiling with optimizing is totally useless, but this is not true. The optimizing stages do a lot of good work, but in this little example it is not apparent since the program is written to show the worst possible case. The example shown is the only case where both waste of instructions and program space can occur in such obvious way. Unfortunately we believe that the problem is hard to solve without severe changes in GNU CC's source.

The code may have been even worse in the optimized case without a small change we have done in the source files (*see 3.3.6, Changes in the source files of GNU CC*). This patch suppresses that the compiler copy *all* the arguments to registers when entering a function regardless of the way the parameters were passed. Without this patch all function entrances would have suffered the same problem discussed in this section, not only the reference parameters.

3.4.2 Comparison with the Oden[1] Ada compiler

To further study the code quality we will show, in this section, a comparison we have made between our compiler and the Oden Ada compiler. For both compilers we have compiled a code sequence, DAIS (Digital Avionics Instruction Set), which is well-known among people working in the space industry for giving a rather good sample of typical computations made in space borne computer programs. You may consider the DAIS program as a kind of benchmark showing typical occurrences of instruction sequences common in systems, for example, in satellites. The program essentially contains various kinds of arithmetic instructions, both integer and floating-point, embedded in 'if'-, 'for'-, and 'switch'-statements (C syntax).

In order to be able to compile the DAIS program on both compilers we had to translate a program given to us, written in Ada, to C. That procedure raised no trouble since the program do not contain any Ada specific constructs, just arithmetic. One slight difference between the Ada and the C version is that the latter one contains a function call, taking every defined variable in the program as arguments, as the very last thing in the program. We introduced this difference after studying a compiled version not containing the function call. When the GNU CC compiler performs the glo-

1. The Oden Ada compiler is specially designed for Thor, for example it supports the built-in hardware instructions in Thor.

bal flow analysis (only when optimizing) on the program it discovers that it can remove essential parts of the program without any change in the semantics, since none of the variables are used outside the program and no functions calls are included in the program. The Ada compiler on the other hand does not perform such flow analysis, and thus a comparison between the programs would have been meaningless. In order to obtain the most fair comparison possible, we therefore added a function call, which uses all defined variables in the program, in the very last part of the C program ('`do_not_delete_fun`'). Now, the flow analysis pass is not able to perform as much deletion of unused code as before, since all variables are used in the program.

Here we give both source files, side by side:

```
procedure DAIS is

   B1,B2,B3,B4:BOOLEAN;
   M1,M2,M3,M4,M5,M6,M7,M8:INTEGER;
   I,IR:INTEGER;
   R1,R2,R3,R4,R5,R6,R7,R8:FLOAT;
   K:FLOAT :=0.9999999;
begin
   IR := 14;
   if IR>75 then M1:=22; R1:=3.3; else M1:=33; R1:=2.2; end if;
   for I in 1 .. 32767  loop
      case M1 is
         when 77 => M1:=M1-60;
         when others => M1:=M1+1;
      end case;
      M2:=M1*17;
      M3:=M2/3+1;
      case M3 is
         when 284 => M4:=M3+7;
         when others => M4:=M3+9;
      end case;
      M5:=M4+13;
      M6:=M5+19;
      M7:=M6+M1+M2;
      case M6 is
         when 323 => M8:=M7*19;
         when others => M8:=M1*23;
      end case;
      R1:=R1*K+K;
      R2:=R1+0.91;
      if  K>0.999
      then R3:=R1*K;B2:=TRUE;
      else R3:=R2*K;B2:=FALSE;
      end if;
      if K>0.9999
      then R4:=R2;B1:=FALSE;
      else R4:=R3;B1:=TRUE;
      end if;
      R5:=R4+0.95;
      R6:=R1*R2*R3*K+R5;
      R7:=R1*K+0.96;
      if K>0.99
      then R8:=R5/K; K:=0.9999*K;B3:=B1 and B2; B4:=B1 or B2;
      else R8:=R7/K; K:=1.001*K; B3:=B1 or B2;  B4:=B1 and B2;
      end if;
   end loop;


end DAIS;
```

```
#define TRUE  1
#define FALSE 0
void do_not_delete_fun ();
void main()
{
  int B1,B2,B3,B4;
  int M1,M2,M3,M4,M5,M6,M7,M8;
  int I,IR;
  float R1,R2,R3,R4,R5,R6,R7,R8;
  float K = 0.9999999;

  IR = 14;
  if (IR > 75){M1 = 22; R1 = 3.3;} else {M1 = 33; R1 = 2.2;}
  for (I = 1; I <= 32767; I++) {
      switch (M1) {
        case 77: M1 = M1 - 60; break;
        default: M1 = M1 + 1;}

      M2 = M1 * 17;
      M3 = M2 / 3 + 1;
      switch (M3) {
        case 284: M4 = M3 + 7; break;
        default: M4 = M3 + 9;}

      M5 = M4 + 13;
      M6 = M5 + 19;
      M7 = M6 + M1 + M2;
      switch (M6) {
        case 323: M8 = M7 * 19; break;
        default: M8 = M1 * 23;}

      R1 = R1 * K + K;
      R2 = R1 + 0.91;
      if (K > 0.999)
      {R3 = R1 * K; B2 = TRUE; }
      else { R3 = R2 * K; B2 = FALSE;}

      if (K > 0.9999)
      {R4 = R2; B1 = FALSE;}
      else {R4 = R3; B1 = TRUE;}

      R5 = R4 + 0.95;
      R6 = R1 * R2 * R3 * K + R5;
      R7 = R1 * K + 0.96;
      if (K > 0.99)
      {R8 = R5 / K; K = 0.9999 * K;B3 = B1 & B2;B4 = B1 | B2;}
      else {R8 = R7 / K; K = 1.001 * K; B3 = B1 | B2;
            B4 = B1 & B2;}
    }
  do_not_delete_fun (B1,B2,B3,B4,M1,M2,M3,M4,M5,M6,M7,M8,I,IR,
                     R1,R2,R3,R4,R5,R6,R7,R8,K);
}
```

Example 20 Comparison between Ada and C versions of DAIS

Below we give listings of the assembler extract of both the Ada and the C version, with the source code given as comments. As per default, the Ada compiler performs various optimizations, and to turn on optimization on GNU CC we must explicitly give such options while compiling (*see 2.2.2.3, Optimizing options*). The command line given in both cases are: For Ada; '`compile -a dais.ada`' and for C; '`thor-gcc -S -O dais.c`'. The option '`-a`' tells the Ada compiler to deliver assembler code, and thus equivalent to option '`-S`' for GNU CC. Here comes both assembler files with source code comments inserted:

```
-- procedure DAIS is
--     B1,B2,B3,B4:BOOLEAN;
--     M1,M2,M3,M4,M5,M6,M7,M8:INTEGER;
--     I,IR:INTEGER;
--     R1,R2,R3,R4,R5,R6,R7,R8:FLOAT;
--     K:FLOAT :=0.9999999;
L3:     MTOS -22
        PSH   L1
-- begin
--     IR := 14;
        PSHI  14
        POP   10
--     if IR>75 then M1:=22; R1:=3.3; else M1:=33; R1:=2.2; end
if;
        PSH   9
        CMPI  75
        JRLE  L7
        MTOS  1
        NOP
        PSHI  22
        POP   19
        JR    L8
        PSH   L5
        POP   9
L7:     PSHI  33
        POP   19
        PSH   L9
        POP   9
--     -- Start measuring here
L8:
--     for I in 1 .. 32767  loop
        PSHI  1
L11:
--        case M1 is
        PSH   19
--           when 77 => M1:=M1-60;
        CMPI  77
        JRNE  L14
        NOP
        NOP
        PSH   20
        JR    L13
        ADDI -60
        POP   21
L14:
--           when others => M1:=M1+1;
        PSH   20
        JR    L13
        ADDI  1
        POP   21
L13:
--        end case;
--        M2:=M1*17;
        PSH   20
        MULI  17
        POP   20
--        M3:=M2/3+1;
        PSH   19
        DIV   L10
        ADDI  1
        POP   19
--        case M3 is
        PSH   18
--           when 284 => M4:=M3+7;
        CMPI  284
        JRNE  L17
        NOP
        NOP
        PSH   19
        JR    L16
        ADDI  7
        POP   19
L17:
--           when others => M4:=M3+9;
        PSH   19
        JR    L16
        ADDI  9
        POP   19
L16:
--        end case;
--        M5:=M4+13;
        PSH   18
        ADDI  13
        POP   18
--        M6:=M5+19;
        PSH   17
```

```
          ; Assembler file for Thor. Version 0.1.0
          ; Generated by thor-gcc.
; void main()
; {
          _do_not_delete_funXREF
          code    SECT 1,R,C
          LC0:
                  DATAF 9.9999988079e-01
          LC1:
                  DATAF 2.2000000477e+00
          LC2:
                  DATA 3
          LC3:
                  DATAF 9.1000000000e-01
          LC4:
                  DATAF 9.9900001287e-01
          LC5:
                  DATAF 9.9989998341e-01
          LC6:
                  DATAF 9.5000000000e-01
          LC7:
                  DATAF 9.6000000000e-01
          LC8:
                  DATAF 9.9000000954e-01
          LC9:
                  DATAF 1.0010000467e+00
          _main   XDEF
          _main:
;   int B1,B2,B3,B4;
;   int M1,M2,M3,M4,M5,M6,M7,M8;
;   int I,IR;
;   float R1,R2,R3,R4,R5,R6,R7,R8;
;   float K = 0.9999999;
                  MTOS -23         ; local vars(22) + TOP reg(1)
                  PSH LC0
                  POP 1            ; TOP
;   IR = 14;
;   if (IR > 75)
;     {
;       M1 = 22;
;       R1 = 3.3;
;     }
;   else
;     {
;       M1 = 33;
;       R1 = 2.2;
;     }
                  PSHI 33
                  POP 19           ; S+18
                  PSH LC1
                  POP 10           ; S+9
;   for (I = 1; I <= 32767; I++)
;     {
                  PSHI 1
                  POP 11           ; S+10
          L7:
;       switch (M1)
;         {
;         case 77:
;           M1 = M1 - 60;
;           break;
;         default:
;           M1 = M1 + 1;
;         }
                  PSH 18           ; S+18
                  CMPI 77
                  JRNE L10
                  MTOS 1
                  NOP
                  JR L27
                  NOP
                  PSHI 17
          L10:
                  PSH 18           ; S+18
                  ADDI 1
          L27:
                  POP 19           ; S+18
;       M2 = M1 * 17;
;       M3 = M2 / 3 + 1;
                  PSH 18           ; S+18
                  SL 4
                                   ; PEEP - pop,push (dead)
                  ADD 19           ; S+18
                  POP 18           ; S+17
                  PSH 17           ; S+17
```

```
       ADDI  19
       POP   17
--       M7:=M6+M1+M2;
       PSH   16
       ADD   22
       ADD   21
       POP   16
--       case M6 is
       PSH   16
--          when 323 => M8:=M7*19;
       CMPI  323
       JRNE  L19
       NOP
       NOP
       PSH   16
       JR    L18
       MULI  19
       POP   16
L19:
--          when others => M8:=M1*23;
       PSH   22
       JR    L18
       MULI  23
       POP   16
L18:
--       end case;
--       R1:=R1*K+K;
       PSH   12
       MULF  5
       ADDF  5
       POP   13
--       R2:=R1+0.91;
       PSH   12
       ADDF  L15
       POP   12
--       if  K>0.999
       PSH   4
       CMPF  L20
       JRLE  L21
       MTOS  1
       NOP
--       then R3:=R1*K;B2:=TRUE;
       PSH   12
       MULF  5
       POP   11
       JR    L22
       PSHI  1
       POP   26
L21:
--       else R3:=R2*K;B2:=FALSE;
       PSH   11
       MULF  5
       POP   11
       PSHI  0
       POP   26
--       end if;
L22:
--       if K>0.9999
       PSH   4
       CMPF  L23
       JRLE  L24
       MTOS  1
       NOP
--       then R4:=R2;B1:=FALSE;
       PSH   11
       POP   10
       JR    L25
       PSHI  0
       POP   27
L24:
--       else R4:=R3;B1:=TRUE;
       PSH   10
       POP   10
       PSHI  1
       POP   27
--       end if;
L25:
--       R5:=R4+0.95;
       PSH   9
       ADDF  L26
       POP   9
--       R6:=R1*R2*R3*K+R5;
       PSH   12
       MULF  12
       MULF  11
```

```
                   DIV LC2
                               ; PEEP - pop,push (dead)
                   ADDI 1
                   POP 17      ; S+16
;         switch (M3)
;           {
;           case 284:
;             M4 = M3 + 7;
;             break;
;           default:
;             M4 = M3 + 9;
;           }
                   PSH 16      ; S+16
                   CMPI 284
                   JRNE L14
                   MTOS 1
                   NOP
                   JR L28
                   NOP
                   PSHI 291
           L14:
                   PSH 16      ; S+16
                   ADDI 9
           L28:
                   POP 16      ; S+15
;         M5 = M4 + 13;
;         M6 = M5 + 19;
;         M7 = M6 + M1 + M2;
                   PSH 15      ; S+15
                   ADDI 13
                   POP 15      ; S+14
                   PSH 14      ; S+14
                   ADDI 19
                   POP 14      ; S+13
                   PSH 13      ; S+13
                   ADD 19      ; S+18
                               ; PEEP - pop,push (dead)
                   ADD 18      ; S+17
                   POP 13      ; S+12
;         switch (M6)
;           {
;           case 323:
;             M8 = M7 * 19;
;             break;
;           default:
;             M8 = M1 * 23;
;           }
                   PSH 13      ; S+13
                   CMPI 323
                   JRNE L18
                   MTOS 1
                   NOP
                   JR L29
                   PSH 12      ; S+12
                   MULI 19
           L18:
                   PSH 18      ; S+18
                   MULI 23
           L29:
                   POP 12      ; S+11
;         R1 = R1 * K + K;
;         R2 = R1 + 0.91;
                   PSH 9       ; S+9
                   MULF 1      ; TOP
                               ; PEEP - pop,push (dead)
                   ADDF 1      ; TOP
                   POP 10      ; S+9
                   PSH 9       ; S+9
                   ADDF LC3
                   POP 9       ; S+8
;         if (K > 0.999)
;           {
;             R3 = R1 * K;
;             B2 = TRUE;
;           }
;         else
;           {
;             R3 = R2 * K;
;             B2 = FALSE;
;           }
                   PSH 0       ; TOP
                   CMPF LC4
                   JRLE L20
                   MTOS 1
                   NOP
```

```
       MULF  5
       ADDF  9
       POP   8
--       R7:=R1*K+0.96;
       PSH   12
       MULF  5
       ADDF  L27
       POP   7
--       if K>0.99
       PSH   4
       CMPF  L28
       JRLE  L29
       MTOS  1
       NOP
--       then R8:=R5/K; K:=0.9999*K;B3:=B1 and B2; B4:=B1 or B2;
       PSH   8
       DIVF  5
       POP   6
       PSH   4
       MULF  L23
       POP   5
       PSH   25
       PSH   27
       AND   1
       POP   26
       PSH   26
       PSH   28
       OR    1
       JR    L30
       POP   26
       MTOS  2
L29:
--       else R8:=R7/K; K:=1.001*K; B3:=B1 or B2;  B4:=B1 and
B2;
       PSH   6
       DIVF  5
       POP   6
       PSH   4
       MULF  L31
       POP   5
       PSH   25
       PSH   27
       OR    1
       POP   26
       PSH   26
       PSH   28
       AND   1
       POP   26
       MTOS  2
--       end if;
L30:
--    end loop;
       PSH   3
       CMPI  32767
       JRLT  L11
       MTOS  4
       ADDI  1
L12:   MTOS  1
-- end DAIS;
L4:    RET   23
       MTOS  24
       NOP
L33:
L1:    DATAF     0.00000E-01
L5:    DATAF     3.30000E+00
L9:    DATAF     2.20000E+00
L10:   DATA   3
L15:   DATAF     9.10000E-01
L20:   DATAF     9.99000E-01
L23:   DATAF     9.99900E-01
L26:   DATAF     9.50000E-01
L27:   DATAF     9.60000E-01
L28:   DATAF     9.90000E-01
L31:   DATAF     1.00100E+00
L2:
L37:   PSHI -1
       PSH   L32
       PSHR  TOS
       ADDI  1
       POPR  ER
       PSH   L38
       POPR  TP
       PSHI -1
       POP    16 (TIB_SECTION)
       PSH   L39
       POP    17 (TIB_SECTION)
```

```
                PSH 9           ; S+9
                MULF 1          ; TOP
                JR L30
                POP 8           ; S+7
                PSHI 1
        L20:
                PSH 8           ; S+8
                MULF 1          ; TOP
                POP 8           ; S+7
                PSHI 0
        L30:
                POP 22          ; S+21
;       if (K > 0.9999)
;         {
;           R4 = R2;
;           B1 = FALSE;
;         }
;       else
;         {
;           R4 = R3;
;           B1 = TRUE;
;         }
                PSH 0           ; TOP
                CMPF LC5
                JRLE L22
                MTOS 1
                NOP
                PSH 8           ; S+8
                JR L31
                POP 7           ; S+6
                PSHI 0
        L22:
                PSH 7           ; S+7
                POP 7           ; S+6
                PSHI 1
        L31:
                POP 23          ; S+22
;       R5 = R4 + 0.95;
;       R6 = R1 * R2 * R3 * K + R5;
;       R7 = R1 * K + 0.96;
                PSH 6           ; S+6
                ADDF LC6
                POP 6           ; S+5
                PSH 9           ; S+9
                MULF 9          ; S+8
                                ; PEEP - pop,push (dead)
                MULF 8          ; S+7
                                ; PEEP - pop,push (dead)
                MULF 1          ; TOP
                                ; PEEP - pop,push (dead)
                ADDF 6          ; S+5
                POP 5           ; S+4
                PSH 9           ; S+9
                MULF 1          ; TOP
                                ; PEEP - pop,push (dead)
                ADDF LC7
                POP 4           ; S+3
;       if (K > 0.99)
;         {
;           R8 = R5 / K;
;           K = 0.9999 * K;
;           B3 = B1 & B2;
;           B4 = B1 | B2;
;         }
;       else
;         {
;           R8 = R7 / K;
;           K = 1.001 * K;
;           B3 = B1 | B2;
;           B4 = B1 & B2;
;         }
                PSH 0           ; TOP
                CMPF LC8
                JRLE L24
                MTOS 1
                NOP
                PSH 5           ; S+5
                DIVF 1          ; TOP
                POP 3           ; S+2
                PSH 0           ; TOP
                MULF LC5
                POP 1           ; TOP
                PSH 22          ; S+22
                AND 22          ; S+21
                POP 21          ; S+20
```

```
        PSHI  32
        POP    18 (TIB_SECTION)
        PSH   L39
        POP    19 (TIB_SECTION)
        PSHI  0
        POP    20 (TIB_SECTION)
        PSH   L40
        POP    21 (TIB_SECTION)
        PSH   L41
        POP    22 (TIB_SECTION)
        PSH   L42
        POP    23 (TIB_SECTION)
        PSHI  0
        PSHI  1
        TREG
        PSHI  1025
        POPR  TR
        TSCH
L36:    NOP
        NOP
        NOP
        JR    L36
        NOP
        NOP
L35:
        CALL  L3
        NOP
        NOP
        PSHI  0
L34:    CLRF  32
        NOP
        NOP
        HLT
        NOP
        NOP
        NOP
L32:    DATA  L34
L38:    DATA   0 (TIB_SECTION)
L39:    DATA   40 (DATA_SECTION)
L40:    DATA  L35
L41:    DATA   0 (DATA_SECTION)
L42:    DATA   42 (DATA_SECTION)
```

```
                        JR L32
                        PSH 22          ; S+22
                        OR 22           ; S+21
                L24:
                        PSH 3           ; S+3
                        DIVF 1          ; TOP
                        POP 3           ; S+2
                        PSH 0           ; TOP
                        MULF LC9
                        POP 1           ; TOP
                        PSH 22          ; S+22
                        OR 22           ; S+21
                        POP 21          ; S+20
                        PSH 22          ; S+22
                        AND 22          ; S+21
                L32:
                        POP 20          ; S+19
;       }
                        PSH 10          ; S+10
                        ADDI 1
                        POP 11          ; S+10
                        PSH 10          ; S+10
                        CMPI 32767
                        JRLE L7
                        MTOS 1
                        NOP
;   do_not_delete_fun (B1,B2,B3,B4,
;                      M1,M2,M3,M4,M5,M6,M7,M8,
;                      I,IR,
;                      R1,R2,R3,R4,R5,R6,R7,R8,
;                      K);
                        PSH 0           ; TOP
                        PSH 3           ; S+2
                        PSH 5           ; S+3
                        PSH 7           ; S+4
                        PSH 9           ; S+5
                        PSH 11          ; S+6
                        PSH 13          ; S+7
                        PSH 15          ; S+8
                        PSH 17          ; S+9
                        PSHI 14
                        PSH 20          ; S+10
                        PSH 22          ; S+11
                        PSH 24          ; S+12
                        PSH 26          ; S+13
                        PSH 28          ; S+14
                        PSH 30          ; S+15
                        PSH 32          ; S+16
                        PSH 34          ; S+17
                        PSH 36          ; S+18
                        PSH 38          ; S+19
                        PSH 40          ; S+20
                        PSH 42          ; S+21
                        PSH 44          ; S+22
                        CALL _do_not_delete_fun
                        NOP
                        NOP
; }
                        RET 23
                        POP 24  ; copy TOP-reg to callers TOP-reg
                        MTOS 23 ; local vars(22) + 1 + pop_args(0)
```

Example 21 Comparison of Ada and C compilers

Certain things must be noticed about the assembler files:

- Everything after label 'L2' in the Ada assembler file could be neglected, since that code takes care of the initialization of the Ada tasking features, supported by the hardware in the Thor processor.
- The call to 'do_not_delete_fun' in the end of the C assembler file including all the pushes of parameters, should be disregarded by reasons discussed above. Observe that the absolute last three instructions should not be neglected, since that is the return sequence of the 'main' function.
- In the C version we used the bitwise operators, '&' and '|', when we translated the Ada operators, 'and' and 'or'. If we had used operators '&&' and '||' in

the C program, the generated assembler file would not have contained the assembler instructions, 'AND' and 'OR', but rather testing instructions for lazy evaluation[1]. On the other side the Ada version utilizes these assembler instructions when generating code for 'and' and 'or'. Thus Ada and C do have a different interpretation for logic operators. Therefore we decided it would be more fair if the C version uses the bitwise operator so that both assembler files are more comparable to each other.

Both compilers seem to deliver good code at first sight, despite the differences in the assembler layout between the two files. We have not had the energy to perform an in-depth investigation finding out the superior compiler, but here comes some details we found interesting while studying and comparing the two versions:

- Both compilers utilizes the same amount of stack slots, 23, reserved for local variables in the program. Thus the Ada version reserves 22 slots with instruction 'MTOS', and then it pushes a floating-point constant just after that. That behaviour is very smart, since compared to our compiler it makes use of one instruction less. We reserves 23 slots right away, then pushes the constant which we in turn pop into one of the reserved slots.
- The first 'if' statement is a rather naive one, since the result of the test of variable 'IR' could be determined in advance. GNU CC detects this and thus does not perform any testing of variable 'IR' at all, it knows that the 'else' part will be chosen. The Ada compiler does not perform such an analysis and is not able to omit the testing of 'IR'. One can state that this is very stupid programming and optimizations like this would never occur when programming normally, but nevertheless the DAIS program is written is this fashion.
- The Ada compiler adds up all pushes done and keeps track of the stack-depth at every place in the program. This makes it possible to postpone the cleaning up of some of the temporary values pushed on the stack, until the end, where a final 'MTOS'-instruction takes care of all the cleaning. In the C compiler we always clean up temporary values as soon as possible, often in a delay slot of a following branch instruction, since we do not have a complete trace of the stack-depth at every place in the program. However, the comparison made shows that, in this case, it does not matter so much. The Ada compiler does not gain so much by using this approach, since we place the cleaning instruction in a delay slot, which is otherwise unused. But in the long run, the Ada compiler approach probably wins.

3.4.3 Interesting details

In this section we point out certain behaviours about our GNU CC port, good or bad things, that we have discovered while studying the produced code.

1. Lazy evaluation generally means that no value is computed until it is in fact needed. In this case it means that if the first expression in a 'or'-expression is true, the next expression is never evaluated, since it would not influence the outcome of the 'or'-expression.

- The delay-slot filling of the unconditional jumps finally seems to work quite well, despite all the trouble about our changed semantics and bugs in the GNU CC's source code we have encountered. Maybe some possible delay-slot fillings are now rejected, but we think those cases are quite rare. The delay-slot filling of the conditional jumps does not use any of the GNU CC's built-in algorithms for selecting appropriate instructions to insert, and as a result of this one delay-slot always remains empty. Thus it is a pity we did not find a suitable way to make use of these algorithms, we do not think that it is an easy task to make the algorithms work along with our strategy.
- When one compares assembler files compiled with the optimize option '`-O`', with versions compiled with the option '`-O2`' (same source code as with '`-O`'), the number of reserved stack slots seem to be a little higher in the latter version. We estimated the increase could be as high as 20% in the worst case. This behaviour is due to the GNU CC's preference for register discussed in section *3.4.1*, and the more the compiler is told to optimize the more anxious it gets to use registers. We do not think that this desire for registers can be easily suppressed without severe alterations in the source code. Even though the reserved number of stack slots may be a little higher in the '`-O`' version, one should not forget that it does many other more fruitful optimizations as well. Generally, we think that small functions suffer more from this problem than larger ones, since a large function gives the compiler more space for it's code transformations.
- The unoptimized version of a program sometimes contains a large number of unnecessary data moves and other peculiar things that at first sight appear to be redundant. This is partly due to our chosen strategy, which makes the compiler generate code with lot of '`PSH`' and '`POP`' instructions. On the other hand we manage to enhance the code quality quite much in the optimized case, thanks to all of our peephole optimizations. We have invented rather advanced patterns to be recognized by the peepholes, and the gains won by them should not be neglected.

3.5 Remaining work

In the following sections we discuss things that remain to be done before we consider the compiler being a useful tool for developing software. The ANSI C standard requires for example that a set of library functions are available for the programmer, and those are still to be implemented. Testing and validation of the compiler are also things left to be done. In the last section we discuss improvements that could be done to the compiler with a limited amount of work. We estimate that implementing all the standard libraries, providing debug support and running a validation suite would take almost as much time as we have spent until now.

3.5.1 Standard C library functions

If one wants to claim that a C compiler supports the ANSI standard, one must make certain library functions available, which the programmer can access when including header files such as; '`stdio.h`', '`math.h`' etc. In this project there was simply no time left to investigate how difficult it would be to construct and build these libraries. Most of the libraries are written in C code

and the idea is that one should compile these files with our Thor cross-compiler and directly gain access to all the standard libraries. Although this principle might be true with some libraries where no system dependent parts are included, it is not true with libraries such as '`stdio.h`' where some parts need to be specially written for Thor.

3.5.2 Validation

There are several commercial validation suites of C programs with which one can test an compiler and verify if it lives up to the claim of being an ANSI C compiler. Unfortunately the term “ANSI C” is not very strictly specified and there exist as many interpretations of this term as there are compilers. It seems that a competition among these validation suites is going on in trying to make the most obscure interpretation of the ANSI C standard. We received some samples from a company offering a test suite and we do not remember that GNU CC’s front end succeeded to parse even one of the sample programs, even though both the test programs and GNU CC claim to be ANSI C compatible.

A good validation suite should not only test the front end by trying to construct nearly impossible programs to parse but it must also test other parts of the compiler like code correctness etc. Programs testing correctness of the code are often so called self-checking, for example, a message may be printed on the '`stderr`' stream if the compiler fails to deliver correct code. Since we have not yet implemented any standard libraries, we have conclusively not been able to run any programs of this kind either, due to lack of time. See *APPENDIX D - 'List of C validation suites'* for a listing of companies offering such suites.

During our project, when we tried to test the compiler we usually wrote our own test programs which contained some special C constructs we wanted to validate, and when a construct seemed to work we tried to stress the compiler with yet another C construct. Often we tested how the compiler dealt with addressing modes. When our own test programs seemed to work satisfactorily and we could not figure out anything more to test, we tried to compile some source files of GNU CC itself and by studying the assembler code we discovered several hard-to-find bugs. When we had integrated the assembler into the compiler we also tried to run a few programs on the Thor chip itself, but any true validation of the compiler in the sense of running a whole validation suite on the chip has not been done.

3.5.3 The compiler

In the following sections we discuss problems concerning debug support and minor improvements that could be done to further enhance the code quality.

3.5.3.1 Debug support

In our present version of the compiler we have not provided any debug support at all, and we have not given much thought on how to solve the problem in the best way, even though we have some ideas.

In the '`.h`'-file one can define macros that activate one of the standard debug formats, such as DBX, DWARF, SBD, XCOFF or GNU’s own format GDB. If one chooses to use one of these formats, additional information is printed to the assembler file concerning matters like: file name, lines in the source code etc. The assembler must in turn be prepared to receive this information

and integrate it into the object file. Running the program in a debugging tool, like GNU's GDB, is a great help when trying to find bugs in a program, since one is able to identify on which line in the source code the program is executing, which frames are on the stack, contents of variables etc.

The debugging environment used when evaluating Ada programs do not use any of the debugging formats mentioned above, it uses the IEEE Standard 695, and this format should be the only one of interest to support because C programs are likely to be evaluated in the same environment. We have not investigated the difference between this format and any of those supported by GNU CC. It might be possible to make some of formats supported by GNU CC work accordingly to the IEEE format, probably with some changes in the source code of GNU CC.

The assembler does not support this IEEE standard today and must also be adjusted to make debugging support possible.

When studying how other GNU CC ports handles the debugging challenge we have found some pretty obscure attempts when trying to adapt debugging support to their systems and environments. One port we studied let a standard debug format output the debug information to the assembler file, as usual, but before the file was passed to the assembler a preprocessor was executed on the file removing all debug information and saving it in some other way. Changes were then made to the debug format. After the assembler had run and thus produced an object file, the debug information was inserted into the object file. We consider this solution to be a very ad-hoc way of dealing with debug support and we do not recommend this approach on this compiler. We think is should be far more easier to make IEEE adjustments directly in the source code even if severe alterations would be required.

3.5.3.2 Simple improvements of the compiler

Many small and big improvements can be done to the compiler. Here we present a few small ones:

- *Compare improvements*. Normally, when emitting the RTL, we expand an operation into three parts, first a push instruction, then the actual operation and in the end a pop instruction. However, when emitting the compare instruction we only expand it into one single part. The improvement would be to make it into two parts, first a push instruction, and then the actual compare. The reason for not doing this in the first place is that the jump-optimization pass in the compiler removed the compare instruction without removing the push instruction belonging to it. This might not be a problem any more, however, since we now have made the '`TMP`'-register RTX into a common, global object and in this way enhanced the workings of the jump-pass slightly, and it now seems to remove all superfluous push instructions. If one manage to expand the compare into two parts, one could probably remove some peepholes and maybe make the produced code better.
- *More peepholes*. The normal optimization passes in the compiler does not work in a satisfactory way in all situations. Many different combinations of instructions still remains to be improved, but one should always try to determine if it is worthwhile to write a peephole. Some combinations are rare and might be skipped. To find new peepholes to implement, study the resulting code!

4 Conclusions

In this chapter we give a survey of our thoughts of this project. We have made some conclusions about how difficult or easy it is to do a port of GNU CC. We have also found some deficiencies, both in Thor and in GNU CC. Finally, some future improvements will be mentioned.

4.1 Conclusions drawn from our work

Making a port of GNU CC has its advantages and drawbacks. We here try to present them, divided into two sections. One for general opinions, and one for Thor specific issues.

4.1.1 Advantages and drawbacks with trying to make a port

Here follows some opinions of ours concerning the porting of GNU CC. The opinions in this section are quite general and are not specific to Thor.

We begin with some advantages:

- Making a port of GNU CC certainly limits the total workload compared to developing a compiler from scratch which in many cases is a project lasting over several years. The total amount of work in our project does not exceed 730 hours per person, including writing this report.
- As mentioned before, GNU CC is a freeware program and there is no cost whatsoever obtaining a version of the compiler. Thus one gets thousands of hours of work for free.
- GNU CC has a reputation of being a reliable, fast and powerful compiler for most of it's target machines. Often, GNU CC is used as a reference when a measurement is desired of how good code a compiler is able to deliver. Statements like: "... able to deliver code as good as GNU CC" are seen once in a while.
- In principle, one gets not only a C compiler when porting GNU CC, one gets several compilers for different programming languages, one for each front end available.
- We believe that it is pretty easy to configure and install a compiler (*see APPENDIX E -, Installation of GNU CC for Thor*). All one has to do is to inform the configure script and the makefile of what host and target the compiler is intended to work on.
- The overall goal of GNU CC is to deliver reliable code for an arbitrary, byte addressable, 32-bit register machine. When one tries to port it to a word addressable stack machine (Thor), one must be prepared that everything might not work according to the plan, but writing a port to a new register machine, with a generated code of acceptable quality, should not be a particularly hard mission, since you are not forced to deal with the kind of problems we have encountered during this project.

There are also some disadvantages:

- The intention of GNU CC is, we believe, that all one has to do when porting the

compiler is to write a few files, independent of the bulk of source code, without needing to read any of the source files. Due to, in certain parts, incomplete documentation we soon found this goal unrealistic. In order to obtain full understanding and control on how GNU CC works, we were compelled to both study the source code, sometimes very carefully, and unfortunately make a few changes in the source code, due to various reasons.

- When we first started the project we worked with version 2.6.3 of GNU CC and later on we switched to the brand new version 2.7.0. Unfortunately, some of the standard macros had changed in their definitions and semantics. If one wants to make an unofficial port of GNU CC, one must be prepared for incompatibilities that may occur if a large update is issued.
- As said, the back end of the compiler may work together with several front ends, and due to differences between the supported languages, the GNU designers have not been able to construct a parser completely according to the ANSI C standard. The differences are small, so one will almost never notice them.

4.1.2 Making a port to the Thor microprocessor

When doing the port of GNU CC to Thor we have made the following conclusions:

- The GNU C Compiler can be ported to a machine that uses stack oriented instructions instead of registers. There are several possible strategies to chose between and we have chosen one which have one simulated register, the TOP register (*see 3.2.1, The lack of registers in Thor*). GNU CC has a rudimentary support of a stack-based machine because it implicitly reloads unallocated registers into stack slots.
- Even if GNU CC is designed for 8-bit addressable machines, one can use it for Thor, which is 32-bit addressable, but it is difficult to make the size of the basic C data types less than 32 bits (the character type). This may be a big problem if one is writing a word-processor with a lot of character manipulations, but we do not believe that it poses any serious problem for space-oriented, real-time applications. We have chosen to let all data types have a size of 32 bits, and this simplified things a great deal. (*See 3.2.3, The 8-bit addressing problem.*)
- The generated assembler code has fairly good quality. The register preference of GNU CC sometimes causes code to be produced that calculates expressions in the wrong order, resulting in many stores and loads from stack slots. It also causes a lot of copying between different temporary stack slots. For smaller expression one gets a better result due to the peephole optimization. A comparison with the Oden Ada compiler shows that the produced code are quite often as good as that of the Ada compiler (*see 3.4.2, Comparison with the Oden Ada compiler*).
- The GNU C Compiler has not been officially ported to any machine with 2 delay slots and the pass handling delay-slot filling is not completely free from errors. This fact and the fact that we have changed the semantics to be able to represent

push and pop instructions, make delay-slot filling hard to use. We have only managed to make delay-slot filling work for unconditional jumps.

4.2 Deficiencies

In the process of porting GNU CC, we have discovered some deficiencies, both in the Thor architecture and in GNU CC. Some of the things mentioned below might not be considered a deficiency by everyone, but we have include them anyway, until we find a more appropriate place for them.

4.2.1 Deficiencies in the Thor Architecture

During our project we sometimes faced things in the processor we wished were solved in another way and some general improvements possible to do.

- The set of *subtract reverse* instructions lacks, in our opinion, the instruction 'SBRI', i.e. subtract reverse immediate. The standard name handling the reversed subtraction would have been easier to write if that instruction had existed. In our solution we have to occupy extra words of memory, since the constant in the operation is placed in a 'DATA' directive and in turn referred to via a label reference.
- There are two additional data move instructions, apart from the push- and pop instructions, called 'LDX' and 'STX' (*see APPENDIX C -, Instruction set for Thor*) which we do not utilize at all. We have not figured out any situation when it would be beneficial to use them. The 'LDX' instruction moves data placed in the stack to another location in the stack, where this location is indirectly pointed out by the 'TOP'-register two instructions prior the 'LDX'. The most interesting effect of this instruction is that the stack remains the same during the execution, i.e. nothing is either pushed or popped to the stack. The 'STX' instruction has the reversed effect compared to the 'LDX'. One could add that the Oden Ada compiler does not utilize these instructions either.
- There is an instruction called 'MOD' in the instruction set that we do not utilize in our port. The reason for not making use of it depends on the interpretation of modulus division (%) in the language C which differs from the one used in Ada. Since the processor is originally designed for Ada it may be unfair to place this point in this section.
- Most C programs, as well as C++ programs, use a lot of pointers. This causes a lot of indirect addressing. We handle these addresses on Thor with the 'PSHX' and 'POPX' instructions, and thus need 3 extra instructions (sometimes less) compared to a stack relative address. This seems to be a little bit inefficient and it also wastes memory. An improvement would be to add more registers to Thor, handling indirect addressing. A minor change might be to change the 'LDX' and 'STX' instructions, and make use of the RR register. This register is not so commonly used and could be used as a sort of frame pointer, with the 'LDX' and 'STX' as a push and pop instruction indirect via this register.

4.2.2 Deficiencies in GNU CC

In the following list we point out a few things that we think made our work more troublesome, apart from the bugs we have found (*see 3.3.6, Changes in the source files of GNU CC*) and GNU CC's overall register thinking (*see 3.4.1, GNU CC's preference for registers*).

- The overall documentation of GNU CC (GNU CC's emacs[1] info pages) could have been more complete and easier to understand. One is more or less forced to read the source files in order to fully understand the purpose of certain macros.
- Some features in the compiler, usually controlled by macros, are not documented at all. Therefore you can say it is pure luck if you happens to spot one in the source files. In our '`thor.h`'-file we have a section in the beginning of the file where we have placed each undocumented macro we use in our port.
- The documentation should have included, in our opinion, a scheme explaining what one ought to keep in mind when making a new port, for example: the minimal set of macros that ought to be defined. We were forced to apply some sort of trial and error approach when testing certain things in the compiler.
- While the compiler is designed to be relatively easy to port to different machines, some things are astonishingly inflexible. For example one is not able to easily select the assembler and linker to use in the compiler (*see 3.3.7.3, The 'as' and 'ld' script files*). There are several other things one cannot affect due to the fact that the feature is hard-coded in the source files (see implementation of '`casesi`' standard name in section *3.3.3.7*).
- GNU C C is designed for machines supporting byte addressing. For architecture like Thor which is only able to address whole words this deficiency was pretty frustrating (*see 3.2.3, The 8-bit addressing problem*).
- GNU CC's approach of using macros in the source files in order to steer how the compiler works certainly have the disadvantage that the source code gets very hairy to read, since the source is covered with '`#ifdef`' preprocessor commands.

4.3 Future improvements

In this section we discuss further improvements that could be considered after the work described in paragraph *3.5 'Remaining work'* has been done. Thus we consider these improvements to be less urgent to be implemented than those in the referred paragraph above.

4.3.1 Compiler enhancements

The compiler itself can be enhanced in many ways. In this section we discuss things internal to the compiler.

4.3.1.1 8-bit characters

As we have explained in section *3.2.3 'The 8-bit addressing problem'* we have chosen to have 32-

1. A text editor made by the GNU association.

bit characters, due to GNU CC's inability to handle word addressable machines. Of course it would be an improvement if we somehow managed to represent characters in bytes. We think there are two possible ways to choose from in order to accomplish this task:

1) You could redefine our pointer size to point to objects of byte size. Since Thor cannot address bytes, one have to recalculate each address in the program, so the data accesses are word aligned. This implies that the whole port has to be rewritten and the part of the strategy could be thrown into the wastepaper basket. We believe that solving the problem in this way would require a lot more work than the next suggestion.

2) You could design a abstract data structure with functions working on it. Then, when the programmer define a variable of this type, he can manipulate the variable with the functions. A C++ class[1] would have been perfect for this purpose, in order to hide the representation of the data structure from the programmer. Suggestion for the structure and strategy are discussed briefly in *Example 8 'A bit structure', page 39*. The advantages of selecting this way of solving the problem are obvious. One can keep our basic port unaltered, which seems to work quite well, and therefore one does not need to perform the manipulations of addresses necessary in the previous suggestion.

There exists yet another way to accomplish the task, but we think it would require that significant parts of GNU CC be rewritten.

3) Essentially, when defining the type and storage layout, you would like it to be possible to define a pointer size of a byte, a character size of byte and the least addressable unit as a word (*see 3.3.2.3, Type and Storage layout*) and then let the compiler generate all necessary code needed for character manipulation. If this where possible all our problems concerning 8-bit pointers would never have existed. Unfortunately, as mentioned, the strategy of the GNU CC itself must be redefined to make GNU CC support this layout.

We are convinced that the preferred strategy is alternative 2, and that the first and last one would be to complicated.

4.3.1.2 Filling delay slots

One could make a true effort to make the GNU CC's delay slot filling algorithms work along with the representation of our '`TMP`' register (*see 3.3.3.7, Control transfer instructions*). In this way one might be able to fill the delay slots of conditional jumps.

4.3.1.3 Register allocation

The approach used today concerning the number of registers given to the compiler to play with (*see 3.2.1, The lack of registers in Thor*), could maybe be modified to reduce the size of local stack allocations. In the section referred to, one can find a discussion why we chose our approach

---

1. A class in C++ is a abstract data structure which both include data members and member functions. The only things visible to the programmer using the class are the functions working on the data structure. Ada's *packages* resemble the class concept very much.

and the existing alternatives. The best approach would be the one where the number of hard registers is not fixed, where it is always set to be the smallest possible number. This would result in fewer stack slots, and a single stack slot may be reused several times in a function.

4.3.1.4 A different strategy with 'PSH' and 'POP'

In order to adjust our port to better suite GNU CC we could also change the overall strategy concerning pushing and popping the stack. Instead of using 'define_expand'-patterns for operations, which generally emits in turn; A push, the operation and a pop, one could try the approach using 'define_split'-patterns. A discussion concerning this matter is found in section *3.2.4 'The representation of 'PSH' and 'POP''*. It is possible that this approach better conserves the semantics of GNU CC concerning the push and pop templates, and some of the problems we have encountered with differences in semantics could be avoided. Many optimizations are better done at a higher level than the push and pop level.

4.3.1.5 Move the 'clobber'-instruction

Today, we emit a 'clobber'-instruction after the pop instruction to indicate that the value at top of stack has been destroyed in the pop instruction. The most logical position for this clobber would be together with the pop, in a parallel construction, since the destruction takes place simultaneously with the popping. The compiler has not been able to interpret the parallel construction correctly, but we have now changed the 'TMP'-register object into a global object and this might remove some of the problems. If it was possible to place the 'clobber' in a parallel expression, we would get a safer solution where no other instruction can accidentally be placed between the pop and the clobber.

4.3.2 Supporting the GNU extended C

As mentioned several times in this document, the GNU CC's parser does not entirely follow ANSI C rules, due to the fact that the back end should be compatible with other front ends (programming languages). For example, it is legal in GNU CC to write nested functions which violate the ANSI C standard. At this moment, we do not support these extensions in our port, but in the long run it is desirable to support these extensions, simply because it is a GNU C Compiler and therefore should act like one.

4.3.3 GNAT and G++

GNU CC is designed to work with several language's front ends, including; Ada, C++, Objective C and Modula-2. In principle it should be a minor task to build a new compiler for a different language, when the original GNU C Compiler is working satisfactorily. Included in the distribution of GNU CC are front ends for C, Objective C (objc) and C++ (g++). The Objective C compiler needs some standard library functions, which are not implemented yet, but a compiler for C++ should be easier to build. The Ada compiler, GNAT, is distributed separately from the normal distribution of GNU CC.

4.3.3.1 G++

G++ is the GNU C++ Compiler (a front end) included in the GNU CC distribution. We have managed to build and install it for Thor, but it is not fully operational. In short, the following things can be said about the G++ compiler:

- You build it with the commands '`make LANGUAGES=c++`' and '`make LANGUAGES=c++ install`' after finishing a normal C language build (*see APPENDIX E -, Installation of GNU CC for Thor*).
- The resulting compiler (thor-gcc or thor-g++) will compile many C++ programs correctly. Even virtual functions[1] seems to work.
- The assembler file will have dollars, '`$`', in identifiers. The assembler must be changed to accept these identifiers or some other solution must be found.
- There is a problem when defining global class objects. The constructors of these objects must be called once in the beginning and never again. The same things is valid for the destructors at the end. GNU CC has several possible solutions to this problem. One solution is to use a program like '`COLLECT2`' to extract information about which global constructors are needed and then arrange a constructor list for the function '`___main`'. Another solution is to have special link sections, which collects information about which global constructors and destructors to call. Read more about this in the GNU CC info pages.
- One has to make certain that the C++ front end does not generate any local stack allocation code. If this happens, it will be impossible to eliminate the frame pointer and the compiler will crash with an unrecognized instruction complaint. If this happens one might handle the situation by changing the stack allocation to a heap allocation instead (for example with function '`malloc`' found in '`stdio`' library), but this might not be possible in all cases.
- There is also a slight danger that G++ uses some features that are equivalent to some of the GNU extensions of the C language. These extension are not supported in the current version of the compiler and this would mean that some G++ features might not work.

4.3.3.2 GNAT

The Ada compiler, GNAT, has not been investigated by us.

4.4 Did we accomplish our goal?

If one compares the goals set in the beginning (*see 1.2, Definition and goal of the GNU CC-Thor project*) with what we actually have accomplished, one can state that we came pretty much halfway. We have succeeded in our primary goal of developing a C compiler based on GNU CC (except the parts described in section *3.5 'Remaining work'*). The secondary goal of making an arbitrary front end work, including a C++ and an Ada compiler, which also utilizes the Ada hardware instructions, has not been accomplished. Instead, a lot of effort has been made to document our solution, in order to make further development of the compiler easier.

1. Virtual functions can be inherited from a parent class to several child classes. When called the program decides at run-time which instance of the function to actually run.

5 Definitions and abbreviations

AG	Address Generation
BOS	Bottom Of Stack
CSE	Common Subexpression Elimination
DMA	Direct Memory Access
EDAC	Error Detection And Correction
EOS	End Of Stack
EX	Execute
GCC	Gnu C Compiler
GNU	Gnu is Not Unix
GNU CC	Gnu C Compiler
IEEE	Institute of Electrical and Electronics Engineers
IF	Instruction Fetch
OF	Operand Fetch
PC	Program Counter
PPC	Prefetch Program Counter
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Language
RTX	An RTL expression
TAP	Test Access Port
TOP	The value residing on TOS
TOS	Top Of Stack

6 Bibliography

The following documents are used as a reference:

[RD1] Emacs info pages of GNU CC.
Version 2.7.0.

[RD2] GNU CC source code.
Version 2.7.0.

[RD3] Compilers Principles, Techniques, and Tools.
Authors: Aho, Sethi and Ullman.
Addison-Wesley Publishing Company 1986.

[RD4] Guide for Microprocessor System Evaluation.
TOR/TNOT/0020/SE, Issue 1, 10 November 1993.

[RD5] Oden Ada Compiler System User's Guide.
TOR/TNOT/0010/SE, Issue 1, 5 November 1993.

[RD6] Stack Risc Microprocessor.
Instruction Set Architecture for Prototype Chip.
TOR/TNOT/0005/SE, Issue 4, 21 October 1992.

[RD7] Stack Risc Microprocessor.
User's Guide for Prototype Chip.
TOR/TNOT/0006/SE, Issue 1, 21 October 1992.

[RD8] Stack Risc Microprocessor.
Evaluation Board Specification.
TOR/TNOT/0012/SE, Issue 2, 2 November 1992.

[RD9] Thor Datasheet.
September 1993.

[RD10] IEEE Trial-Use Standard for Microprocessor.
Universal Format for Object Modules.
9 September 1985.

APPENDIX A - Listing of machine dependent files

The succeeding sections under this paragraph will each give a listing of the machine dependent files we have written to this port. In addition to the thoroughly discussed files, 'thor.md', 'thor.h' and 'thor.c' the listings also include; the 'crt0.asm'-file which could be described as a boot sequence which will be called when executing a program, the 'thor-libgcc1.asm'-file which contains the implicit library functions available, the 'as' and 'ld' scripts which overrides the calls the compiler does to the assembler ('as', i.e. the UNIX assembler) and the linker ('ld', i.e. the UNIX linker) and finally the 't-thor'-file which contains additional rules to the makefile ('Makefile.in') used when building the compiler.

A.1 thor.md

```
/*
 *            thor.md
 *
 *
 * Purpose
 *            The file gives a description of all instructions supported by
 *            the compiler. The purposes of the patterns are described
 *            in GNU CC's documentation.
 *
 * Operations
 *            Along with each pattern follows a description of it's purpose.
 *
 * Usage
 *            Used when building the compiler.
 *
 * I/O        None
 *
 * Machine    (No dependencies)
 *
 *
 * Compiler   (No dependencies)
 *
 *
 * Author     Harry Gunnarsson, Thomas Lundqvist / 951115
 *
 * Revised        by            date    remarks
 *
 *   1.0.0        HG, TL        951115  New version
 *
 */

;;- Machine description for GNU compiler, Thor Version

/*****************************************************************/
/** ATTRIBUTES                                                  **/
/*****************************************************************/

;; Classification of each insn.
;; jump           unconditional jumps
;; condjumpconditional jump
;; call           unconditional call
;; jumpx   indirect jump/call
;; pop            pop and popr
;; popx         indirect pop
;; push           psh, pshr and pshi
;; arith   integer arithmetic instruction
;; floatop floating-point operation
;; nop            no operation
;; misc           default operation

(define_attr "type"
  "jump,condjump,call,jumpx,pop,popx,push,push_normal,arith,compare,nop,misc"
  (const_string "misc"))

;; # instructions
(define_attr "length" "" (const_int 1))

;; Testing if instruction is ok for delay-slot
(define_attr "ok_for_delay_slot" "no,yes"
  (if_then_else (eq_attr "type" "condjump,jump,call,push_normal,jumpx")
                (const_string "no")
                (const_string "yes")))

(define_delay (eq_attr "type" "jump")
  [(and (eq_attr "ok_for_delay_slot" "yes") (eq_attr "length" "1")) (nil) (nil)
   (and (eq_attr "ok_for_delay_slot" "yes") (eq_attr "length" "1")) (nil) (nil)
  ])

;; Condition code effect:
;;
;;   clobber    - cc destroyed
;;   unchanged  - no change
;;   set1       - cc_status.value1 is set from operand[0].
;;   set2       - cc_status.value1 and value2 is set from operand[0,1].
;;   copy       - the tmp-reg is copied to another operand,
;;                update value1 if appropriate.
;;   compare    - a compare, sets value1 with the compare-operand.

(define_attr "cc" "clobber,unchanged,set1,set2,copy,compare"
  (cond [(eq_attr "type" "jump")        (const_string "unchanged")
         (eq_attr "type" "condjump")    (const_string "unchanged")
         (eq_attr "type" "call")        (const_string "clobber")
         (eq_attr "type" "jumpx")       (const_string "clobber")
         (eq_attr "type" "pop")         (const_string "copy")
         (eq_attr "type" "popx")        (const_string "clobber")
         (eq_attr "type" "push")        (const_string "set2")
         (eq_attr "type" "push_normal") (const_string "clobber")
         (eq_attr "type" "arith")       (const_string "set1")
         (eq_attr "type" "compare")     (const_string "compare")
         (eq_attr "type" "nop")         (const_string "unchanged")
         (eq_attr "type" "misc")        (const_string "clobber")
         ]
  (const_string "clobber")))

/*****************************************************************/
/** INSTRUCTIONS                                                **/
/*****************************************************************/

;;- Instruction patterns.  When multiple patterns apply,
;;- the first one in the file is chosen.
;;-
;;- See file "rtl.def" for documentation on define_insn, match_*, et. al.
;;-
;;- cpp macro #define NOTICE_UPDATE_CC in file tm.h handles condition code
;;- updates for most instructions.
;;-
;;- All names beginning with '*' are comments. They are in fact
;;- nameless templates.

;;---------------------------------------------------- Compare, test

(define_expand "tstqi"
  [(set (cc0) (match_operand:QI 0 "nonimmediate_operand" ""))]
  ""
  "
{
/* If operand 0 is not stack relative (is complex or pc relative), we
   must reload it via a pseudo register. */
  if (! simple_operand (operands[0], QImode)
      || GET_CODE (operands[0]) == MEM
         && CONSTANT_ADDRESS_P (XEXP (operands[0], 0)))
    operands[0] = force_reg (QImode, operands[0]);
}")

(define_insn "*tstqi"
  [(set (cc0) (match_operand:QI 0 "stack_relative_operand" "Qt,n"))]
  ""
  "*
{
  if (which_alternative == 0)
    return \"TEST %0\";
```

A.1 thor.md, continued

```
  if (INTVAL (operands[0]) == 0)
    return \"SETF 1\\t\\t; always zero (EQ)\";
  else
    return \"CLRF 1\\t\\t; always non-zero (NE)\";
}"
[(set_attr "type" "compare")])

(define_expand "cmpqi"
  [(set (cc0)
          (compare (match_operand:QI 0 "general_operand" "")
                   (match_operand:QI 1 "general_operand" "")))]
  ""
  "
{
  /* If operand1 is complex, we must reload it via a pseudo register. */
  if (! simple_operand (operands[1], QImode))
    operands[1] = force_reg (QImode, operands[1]);
}")

(define_expand "cmpqf"
  [(set (cc0)
          (compare (match_operand:QF 0 "general_operand" "")
                   (match_operand:QF 1 "general_operand" "")))]
  ""
  "
{
  /* If operand1 is complex, we must reload it via a pseudo register. */
  if (! simple_operand (operands[1], QFmode))
    operands[1] = force_reg (QFmode, operands[1]);
}")

(define_insn "*cmpqi_signed"
  [(set (cc0)
     (compare (match_operand:QI 0 "general_operand" "I,Qt,m,I,Qt,m,I,Qt,m")
                (match_operand:QI 1 "simple_operand" "K,K,K,m,m,m,r,r,r")))]
  "! next_cc0_user_unsigned_jump_p (insn)"
  "*
{
  thor_compare_need_stack_adjust = 1;

  if (which_alternative % 3 == 0)
    output_asm_insn (\"PSHI %0\", operands);
  else if (which_alternative % 3 == 1)
    output_asm_insn (\"PSH %0\", operands);
  else
    output_complex_push (operands[0]);

  return (which_alternative / 3 == 0 ? \"CMPI %1\"
           : which_alternative / 3 == 1 ? \"CMP %z1\"
           : \"CMP %z1\");
}"
[(set_attr "type" "compare,compare,compare,compare,compare,compare,compare,com-
pare,compare")
 (set_attr "length" "2,2,5,2,2,5,2,2,5")])

(define_insn "*cmpqi_unsigned"
  [(set (cc0)
     (compare (match_operand:QI 0 "general_operand" "I,Qt,m,I,Qt,m")
                (match_operand:QI 1 "simple_operand" "m,m,m,r,r,r")))]
  "next_cc0_user_unsigned_jump_p (insn)"
  "*
{
  thor_compare_need_stack_adjust = 2;

  if (which_alternative % 3 == 0)
    output_asm_insn (\"PSHI %0\", operands);
  else if (which_alternative % 3 == 1)
    output_asm_insn (\"PSH %0\", operands);
  else
    output_complex_push (operands[0]);

  return (which_alternative / 3 == 0 ? \"CMPU %z1\;PSHI 0\;CMP 0\"
           : \"CMPU %z1\;PSHI 0\;CMP 0\");
}"
[(set_attr "type" "compare,compare,compare,compare,compare,compare")
 (set_attr "length" "4,4,7,4,4,7")])

(define_insn "*cmpqf"
  [(set (cc0)
          (compare (match_operand:QF 0 "general_operand" "I,Qt,m,I,Qt,m")
                   (match_operand:QF 1 "simple_operand" "m,m,m,r,r,r")))]
  ""
  "*
{
  thor_compare_need_stack_adjust = 1;

  if (which_alternative % 3 == 0)
    output_asm_insn (\"PSHI %0\", operands);
  else if (which_alternative % 3 == 1)
    output_asm_insn (\"PSH %0\", operands);
  else
    output_complex_push (operands[0]);

  return (which_alternative / 3 == 0 ? \"CMPF %z1\"
           : \"CMPF %z1\");
}"
[(set_attr "type" "compare,compare,compare,compare,compare,compare")
 (set_attr "length" "2,2,5,2,2,5")])

;;-------------------------------------------------- Move, push/pop

(define_expand "movqf"
  [(set (match_operand:QF 0 "general_operand" "")
          (match_operand:QF 1 "general_operand" ""))]
  ""
  "
{
  if (emit_move_sequence (operands, QFmode))
    DONE;
}")

(define_expand "movqi"
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "general_operand" ""))]
  ""
  "
{
  if (emit_move_sequence (operands, QImode))
    DONE;
}")

(define_insn "*movqf_push"
  [(set (match_operand:QF 0 "push_operand" "m,m")
          (match_operand:QF 1 "general_operand" "Qt,m"))]
  ""
  "*
{ /* Output insn before incrementing 'thor_top_offset'. */
  static char *table[] = { \"PSH %1\", \"\" };
  if (which_alternative != 1)
    output_asm_insn (table[which_alternative], operands);
  else
    output_complex_push (operands[1]);
```

A.1 thor.md, continued

```
  thor_top_offset++;
  return \"\";
}"
[(set_attr "type" "push_normal,push_normal")
 (set_attr "length" "1,4")])

(define_insn "*movqi_push"
  [(set (match_operand:QI 0 "push_operand" "m,m,m")
          (match_operand:QI 1 "general_operand" "I,Qt,m"))]
  ""
  "*
{ /* Output insn before incrementing 'thor_top_offset'. */
  static char *table[] = {\"PSHI %1\", \"PSH %1\", \"\"};
  if (which_alternative != 2)
    output_asm_insn (table[which_alternative], operands);
  else
    output_complex_push (operands[1]);
  thor_top_offset++;
  return \"\";
}"
[(set_attr "type" "push_normal,push_normal,push_normal")
 (set_attr "length" "1,1,4")])

(define_insn "*movqf_push_tmp"
  [(set (match_operand:QF 0 "tmp_register_operand" "r,r")
          (match_operand:QF 1 "general_operand" "Qt,m"))]
  ""
  "*
{ static char *table[] = {\"PSH %1\", \"\"};
  if (which_alternative != 1)
    output_asm_insn (table[which_alternative], operands);
  else
    output_complex_push (operands[1]);
  return \"\";
}"
[(set_attr "type" "push,push")
 (set_attr "length" "1,4")])

(define_insn "*movqi_push_tmp"
  [(set (match_operand:QI 0 "tmp_register_operand" "r,r,r")
          (match_operand:QI 1 "general_operand" "I,Qt,m"))]
  ""
  "*
{
  static char *table[] = { \"PSHI %1\", \"PSH %1\", \"\"};
  if (which_alternative != 2)
    output_asm_insn (table[which_alternative], operands);
  else
    output_complex_push (operands[1]);
  return \"\";
}"
[(set_attr "type" "push,push,push")
 (set_attr "length" "1,1,4")])

(define_insn "*movqf_pop_tmp"
  [(set (match_operand:QF 0 "general_operand" "Qt,m")
          (match_operand:QF 1 "tmp_register_operand" "r,r"))]
  ""
  "*
{
  if (which_alternative == 0)
    output_asm_insn (\"POP %z0\", operands);
  else
    output_complex_pop (operands[0]);
  return \"\";
}"
[(set_attr "type" "pop,popx")
 (set_attr "length" "1,4")])

(define_insn "*movqi_pop_tmp"
  [(set (match_operand:QI 0 "general_operand" "Qt,m")
          (match_operand:QI 1 "tmp_register_operand" "r,r"))]
  ""
  "*
{ static char *table[] = { \"POP %z0\", \"\"};
  if (which_alternative != 1)
    output_asm_insn (table[which_alternative], operands);
  else
    output_complex_pop (operands[0]);
  return \"\";
}"
[(set_attr "type" "pop,popx")
 (set_attr "length" "1,4")])

;;-------------------------------------------------- Fix,float conversion

(define_expand "floatqiqf2"
  [(set (match_operand:QF 0 "general_operand" "")
          (float:QF (match_operand:QI 1 "general_operand" "")))]
  ""
  "
{
  extern rtx tmp_reg_QI_rtx;
  extern rtx tmp_reg_QF_rtx;

  emit_push (operands[1], QImode);
  emit_insn (gen_rtx (SET, VOIDmode,
                      tmp_reg_QF_rtx,
                      gen_rtx (FLOAT, QFmode,
                               tmp_reg_QI_rtx)));
  emit_pop (operands[0], QFmode);
  DONE;
}")

(define_expand "fix_truncqfqi2"
  [(set (match_operand:QI 0 "general_operand" "")
          (fix:QI (fix:QF (match_operand:QF 1 "general_operand" ""))))]
  ""
  "
{
  extern rtx tmp_reg_QI_rtx;
  extern rtx tmp_reg_QF_rtx;

  emit_push (operands[1], QFmode);
  emit_insn (gen_rtx (SET, VOIDmode,
                      tmp_reg_QI_rtx,
                      gen_rtx (FIX, QImode,
                               gen_rtx (FIX, QFmode,
                                        tmp_reg_QF_rtx))));
  emit_pop (operands[0], QImode);
  DONE;
}")

(define_insn "*floatqiqf2"
  [(set (match_operand:QF 0 "tmp_register_operand" "=r")
          (float:QF (match_operand:QI 1 "tmp_register_operand" "0")))]
  ""
  "FLT"
[(set_attr "type" "arith")])

(define_insn "*fix_truncqfqi2"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r")
```

A.1 thor.md, continued

```
          (fix:QI (fix:QF (match_operand:QF 1 "tmp_register_operand" "0")))))]
  ""
  "INT"
[(set_attr "type" "arith")])

;;--------------------------------------------------- Add

(define_expand "addqf3"
  [(set (match_operand:QF 0 "general_operand" "")
          (plus:QF (match_operand:QF 1 "general_operand" "")
                   (match_operand:QF 2 "general_operand" "")))]
  ""
  "
{
  if (emit_add_sequence (operands, QFmode))
    DONE;
}")

(define_expand "addqi3"
  [(set (match_operand:QI 0 "general_operand" "")
          (plus:QI (match_operand:QI 1 "general_operand" "")
                   (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_add_sequence (operands, QImode))
    DONE;
}")

(define_insn "*addqf"
  [(set (match_operand:QF 0 "tmp_register_operand" "=r,r")
          (plus:QF (match_operand:QF 1 "tmp_register_operand" "0,0")
                   (match_operand:QF 2 "simple_operand" "m,r")))]
  ""
  "@
   ADDF %z2
   ADDF %z2"
[(set_attr "type" "arith,arith")])

(define_insn "*addqi"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r,r")
          (plus:QI (match_operand:QI 1 "tmp_register_operand" "0,0,0")
                   (match_operand:QI 2 "simple_operand" "I,m,r")))]
  ""
  "@
   ADDI %2
   ADD %z2
   ADD %z2"
[(set_attr "type" "arith,arith,arith")])

;; This handles some frame-pointer elimination case:
(define_insn "*addqi_tosadd"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r")
          (plus:QI (match_operand:QI 1 "tos_register_operand" "")
                   (match_operand:QI 2 "immediate_operand" "I")))]
  ""
  "*
{ /* compensate for PSHR-instruction: */
  operands[3] = gen_rtx (CONST_INT, VOIDmode, (INTVAL (operands[2])) + 1);
  return \"PSHR TOS\;ADDI %3\";
}"
[(set_attr "length" "3")])

(define_insn "*addqi_mtos"
  [(set (match_operand:QI 0 "tos_register_operand" "")
          (plus:QI (match_operand:QI 1 "tos_register_operand" "")
                   (match_operand:QI 2 "immediate_operand" "K")))]
  ""
  "*
{
  thor_top_offset -= INTVAL (operands[2]);
  return \"MTOS %2\";
}"
[(set_attr "type" "push_normal")])

;;--------------------------------------------------- Subtract

(define_expand "subqf3"
  [(set (match_operand:QF 0 "general_operand" "")
          (minus:QF (match_operand:QF 1 "general_operand" "")
                    (match_operand:QF 2 "general_operand" "")))]
  ""
  "
{
  if (emit_sub_sequence (operands, QFmode))
    DONE;
}")

(define_expand "subqi3"
  [(set (match_operand:QI 0 "general_operand" "")
          (minus:QI (match_operand:QI 1 "general_operand" "")
                    (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_sub_sequence (operands, QImode))
    DONE;
}")

(define_insn "*subqf3"
  [(set (match_operand:QF 0 "tmp_register_operand" "=r,r")
          (minus:QF (match_operand:QF 1 "tmp_register_operand" "0,0")
                    (match_operand:QF 2 "simple_operand" "m,r")))]
  ""
  "@
   SUBF %z2
   SUBF %z2"
[(set_attr "type" "arith,arith")])

(define_insn "*subqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r")
          (minus:QI (match_operand:QI 1 "tmp_register_operand" "0,0")
                    (match_operand:QI 2 "simple_operand" "m,r")))]
  ""
  "@
   SUB %z2
   SUB %z2"
[(set_attr "type" "arith,arith")])

(define_insn "*subqf3_rev"
  [(set (match_operand:QF 0 "tmp_register_operand" "=r")
          (minus:QF (match_operand:QF 1 "simple_operand" "mt")
                    (match_operand:QF 2 "tmp_register_operand" "0")))]
  ""
  "SBRF %z1"
[(set_attr "type" "arith")])

(define_insn "*subqi3_rev"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r")
          (minus:QI (match_operand:QI 1 "simple_operand" "m,t")
                    (match_operand:QI 2 "tmp_register_operand" "0,0")))]
  ""
```

A.1 thor.md, continued

```
  "@
   SBR %z1
   SBR %z1"
[(set_attr "type" "arith,arith")])

;;------------------------------------------------- Multiply

(define_expand "mulqf3"
  [(set (match_operand:QF 0 "general_operand" "")
        (mult:QF (match_operand:QF 1 "general_operand" "")
                 (match_operand:QF 2 "general_operand" "")))]
  ""
  "
{
  if (emit_commutative_sequence (operands, QFmode, MULT))
    DONE;
}")

(define_expand "mulqi3"
  [(set (match_operand:QI 0 "general_operand" "")
        (mult:QI (match_operand:QI 1 "general_operand" "")
                 (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_commutative_sequence (operands, QImode, MULT))
    DONE;
}")

(define_insn "*mulqf3"
  [(set (match_operand:QF 0 "tmp_register_operand" "=r,r")
        (mult:QF (match_operand:QF 1 "tmp_register_operand" "0,0")
                 (match_operand:QF 2 "simple_operand" "m,r")))]
  ""
  "@
   MULF %z2
   MULF %z2"
[(set_attr "type" "arith,arith")])

(define_insn "*mulqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r,r")
        (mult:QI (match_operand:QI 1 "tmp_register_operand" "0,0,0")
                 (match_operand:QI 2 "simple_operand" "I,m,r")))]
  ""
  "@
   MULI %2
   MUL %z2
   MUL %z2"
[(set_attr "type" "arith,arith,arith")])

;;------------------------------------------------- Divide, Mod

(define_expand "divqf3"
  [(set (match_operand:QF 0 "general_operand" "")
        (div:QF (match_operand:QF 1 "general_operand" "")
                (match_operand:QF 2 "general_operand" "")))]
  ""
  "
{
  if (emit_non_commutative_sequence (operands, QFmode, DIV))
    DONE;
}")

(define_expand "divqi3"
  [(set (match_operand:QI 0 "general_operand" "")
        (div:QI (match_operand:QI 1 "general_operand" "")
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_non_commutative_sequence (operands, QImode, DIV))
    DONE;
}")

(define_expand "modqi3"
  [(set (match_operand:QI 0 "general_operand" "")
        (mod:QI (match_operand:QI 1 "general_operand" "")
                (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_non_commutative_sequence (operands, QImode, MOD))
    DONE;
}")

(define_insn "*divqf3"
  [(set (match_operand:QF 0 "tmp_register_operand" "=r,r")
        (div:QF (match_operand:QF 1 "tmp_register_operand" "0,0")
                (match_operand:QF 2 "simple_operand" "m,r")))]
  ""
  "@
   DIVF %z2
   DIVF %z2"
[(set_attr "type" "arith,arith")])

(define_insn "*divqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r")
        (div:QI (match_operand:QI 1 "tmp_register_operand" "0,0")
                (match_operand:QI 2 "simple_operand" "m,r")))]
  ""
  "@
   DIV %z2
   DIV %z2"
[(set_attr "type" "arith,arith")])

(define_insn "*modqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r")
        (mod:QI (match_operand:QI 1 "tmp_register_operand" "0,0")
                (match_operand:QI 2 "simple_operand" "m,r")))]
  ""
  "@
   PSH 0\;DIV %y2\;MUL %y2\;SBR 1\;POP 1
   PSH 0\;DIV %y2\;MUL %y2\;SBR 1\;POP 1"
[(set_attr "type" "arith,arith")
 (set_attr "length" "5,5")])

;;------------------------------------------------- Bit and,ior,xor

(define_expand "andqi3"
  [(set (match_operand:QI 0 "general_operand" "")
        (and:QI (match_operand:QI 1 "general_operand" "")
                 (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_commutative_sequence (operands, QImode, AND))
    DONE;
}")

(define_expand "iorqi3"
  [(set (match_operand:QI 0 "general_operand" "")
        (ior:QI (match_operand:QI 1 "general_operand" "")
```

A.1 thor.md, continued

```
                    (match_operand:QI 2 "general_operand" ""))))]
  ""
  "
{
  if (emit_commutative_sequence (operands, QImode, IOR))
    DONE;
}")

(define_expand "xorqi3"
  [(set (match_operand:QI 0 "general_operand" "")
          (xor:QI (match_operand:QI 1 "general_operand" "")
                    (match_operand:QI 2 "general_operand" ""))))]
  ""
  "
{
  if (emit_commutative_sequence (operands, QImode, XOR))
    DONE;
}")

(define_insn "*andqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r,r")
          (and:QI (match_operand:QI 1 "tmp_register_operand" "0,0,0")
                    (match_operand:QI 2 "simple_operand" "I,m,r"))))]
  ""
  "@
   ANDI %h2
   AND %z2
   AND %z2"
[(set_attr "type" "arith,arith,arith")])

(define_insn "*iorqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r,r")
          (ior:QI (match_operand:QI 1 "tmp_register_operand" "0,0,0")
                    (match_operand:QI 2 "simple_operand" "I,m,r"))))]
  ""
  "@
   ORI %h2
   OR %z2
   OR %z2"
[(set_attr "type" "arith,arith,arith")])

(define_insn "*xorqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r")
          (xor:QI (match_operand:QI 1 "tmp_register_operand" "0,0")
                    (match_operand:QI 2 "simple_operand" "m,r"))))]
  ""
  "@
   XOR %z2
   XOR %z2"
[(set_attr "type" "arith,arith")])

;;----------------------------------------------------- neg, one_cmpl

(define_expand "negqf2"
  [(set (match_operand:QF 0 "general_operand" "")
          (neg:QF (match_operand:QF 1 "general_operand" "")))]
  ""
  "
{
  extern rtx tmp_reg_QF_rtx;

  emit_push (operands[1], QFmode);
  emit_insn (gen_rtx (SET,VOIDmode,
                         tmp_reg_QF_rtx,
                         gen_rtx (MINUS, QFmode,
                                  force_const_mem (QFmode, CONST0_RTX (QFmode)),
                                  tmp_reg_QF_rtx)));
  emit_pop (operands[0], QFmode);
  DONE;
}")

(define_expand "negqi2"
  [(set (match_operand:QI 0 "general_operand" "")
          (neg:QI (match_operand:QI 1 "general_operand" "")))]
  ""
  "
{
  extern rtx tmp_reg_QI_rtx;

  emit_push (operands[1], QImode);
  emit_insn (gen_rtx (SET,VOIDmode,
                         tmp_reg_QI_rtx,
                         gen_rtx (NEG, QImode,
                                  tmp_reg_QI_rtx)));
  emit_pop (operands[0], QImode);
  DONE;
}")

(define_expand "one_cmplqi2"
  [(set (match_operand:QI 0 "general_operand" "")
          (not:QI (match_operand:QI 1 "general_operand" "")))]
  ""
  "
{
  extern rtx tmp_reg_QI_rtx;

  emit_push (operands[1], QImode);
  emit_insn (gen_rtx (SET,VOIDmode,
                         tmp_reg_QI_rtx,
                         gen_rtx (NOT, QImode,
                                  tmp_reg_QI_rtx)));
  emit_pop (operands[0], QImode);
  DONE;
}")

(define_insn "*negqi2"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r")
          (neg:QI (match_operand:QI 1 "tmp_register_operand" "0")))]
  ""
  "NOT -1"
[(set_attr "type" "arith")])

(define_insn "*one_cmplqi2"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r")
          (not:QI (match_operand:QI 1 "tmp_register_operand" "0")))]
  ""
  "NOT 0"
[(set_attr "type" "arith")])


;;----------------------------------------------------- Shift

(define_expand "ashrqi3"
  [(set (match_operand:QI 0 "general_operand" "")
          (ashiftrt:QI (match_operand:QI 1 "general_operand" "")
                         (match_operand:QI 2 "general_operand" ""))))]
  ""
  "
{
  if (emit_non_commutative_sequence (operands, QImode, ASHIFTRT))
    DONE;
}")
```

A.1 thor.md, continued

```
(define_expand "ashlqi3"
  [(set (match_operand:QI 0 "general_operand" "")
          (ashift:QI (match_operand:QI 1 "general_operand" "")
                     (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_non_commutative_sequence (operands, QImode, ASHIFT))
    DONE;
}")

(define_expand "lshrqi3"
  [(set (match_operand:QI 0 "general_operand" "")
          (lshiftrt:QI (match_operand:QI 1 "general_operand" "")
                       (match_operand:QI 2 "general_operand" "")))]
  ""
  "
{
  if (emit_non_commutative_sequence (operands, QImode, LSHIFTRT))
    DONE;
}")

(define_insn "*ashrqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r,r")
          (ashiftrt:QI (match_operand:QI 1 "tmp_register_operand" "0,0,0")
                       (match_operand:QI 2 "simple_operand" "N,m,r")))]
  ""
  "@
   SRA %2
   SRAD %z2
   SRAD %z2"
[(set_attr "type" "arith,arith,arith")])

(define_insn "*ashlqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r,r")
          (ashift:QI (match_operand:QI 1 "tmp_register_operand" "0,0,0")
                     (match_operand:QI 2 "simple_operand" "N,m,r")))]
  ""
  "@
   SL %2
   SLD %z2
   SLD %z2"
[(set_attr "type" "arith,arith,arith")])

(define_insn "*lshrqi3"
  [(set (match_operand:QI 0 "tmp_register_operand" "=r,r,r")
          (lshiftrt:QI (match_operand:QI 1 "tmp_register_operand" "0,0,0")
                       (match_operand:QI 2 "simple_operand" "N,m,r")))]
  ""
  "@
   SR %2
   SRD %z2
   SRD %z2"
[(set_attr "type" "arith,arith,arith")])

;; This insn is only here to be recognized in init_expmed (expmed.c).
;; Without it the 'mult'-insn will be used instead (we want ashl to
;; be used).
;; The combine-pass should not be allowed to use it.

(define_insn "*ashlqi3-precombine"
  [(set (match_operand:QI 0 "register_operand" "=g")
          (ashift:QI (match_operand:QI 1 "register_operand" "g")
                     (match_operand:QI 2 "general_operand" "g")))]
  "THOR_BEFORE_COMBINE"
  "error: ashlqi3-precombine")

;;-------------------------------------------------- Branch

(define_insn "jump"
  [(set (pc)
          (label_ref (match_operand 0 "" "")))]
  ""
  "*
{ /* update the initial thor_top_offset for the basic block with this label: */
  int index = CODE_LABEL_NUMBER (operands[0]) - get_first_label_num ();
  thor_top_offset_at[index] = thor_top_offset;

  return \"JR %0%#\";
}"
[(set_attr "type" "jump")])

(define_expand "beq"
  [(set (pc) (if_then_else (eq (cc0)
                                   (const_int 0))
                               (label_ref (match_operand 0 "" ""))
                               (pc)))]
  ""
  "")

(define_expand "bne"
  [(set (pc) (if_then_else (ne (cc0)
                                   (const_int 0))
                               (label_ref (match_operand 0 "" ""))
                               (pc)))]
  ""
  "")

(define_expand "bgt"
  [(set (pc) (if_then_else (gt (cc0)
                                   (const_int 0))
                               (label_ref (match_operand 0 "" ""))
                               (pc)))]
  ""
  "")

(define_expand "bgtu"
  [(set (pc) (if_then_else (gtu (cc0)
                                    (const_int 0))
                               (label_ref (match_operand 0 "" ""))
                               (pc)))]
  ""
  "")

(define_expand "blt"
  [(set (pc) (if_then_else (lt (cc0)
                                   (const_int 0))
                               (label_ref (match_operand 0 "" ""))
                               (pc)))]
  ""
  "")

(define_expand "bltu"
  [(set (pc) (if_then_else (ltu (cc0)
                                    (const_int 0))
                               (label_ref (match_operand 0 "" ""))
                               (pc)))]
  ""
  "")

(define_expand "bge"
```

A.1 thor.md, continued

```
  [(set (pc) (if_then_else (ge (cc0)
                                  (const_int 0))
                              (label_ref (match_operand 0 "" ""))
                              (pc)))]
  ""
  "")

(define_expand "bgeu"
  [(set (pc) (if_then_else (geu (cc0)
                                   (const_int 0))
                              (label_ref (match_operand 0 "" ""))
                              (pc)))]
  ""
  "")

(define_expand "ble"
  [(set (pc) (if_then_else (le (cc0)
                                  (const_int 0))
                              (label_ref (match_operand 0 "" ""))
                              (pc)))]
  ""
  "")

(define_expand "bleu"
  [(set (pc) (if_then_else (leu (cc0)
                                   (const_int 0))
                              (label_ref (match_operand 0 "" ""))
                              (pc)))]
  ""
  "")

(define_insn "*condjump"
  [(set (pc)
        (if_then_else (match_operator 1 "comparison_operator"
                                       [(cc0) (const_int 0)])
                       (label_ref (match_operand 0 "" ""))
                       (pc)))]
  ""
  "*
{
  output_condjump (operands[0], operands[1], 0);

  /* if the previous instruction was a compare we must emit an
     MTOS to restore the stack in the first delay slot. */
  if (thor_compare_need_stack_adjust == 1)
    output_asm_insn (\"MTOS 1\", operands);
  else if (thor_compare_need_stack_adjust == 2)
    output_asm_insn (\"MTOS 2\", operands);
  else
    output_asm_insn (\"NOP\", operands);

  thor_compare_need_stack_adjust = 0;

  /* The second delay slot will never be filled: */
  return \"NOP\";
}"
[(set_attr "type" "condjump")
 (set_attr "length" "3")])

;; We also define an inversed conditional jump, to be used by the jump
;; pass. This pattern is only necessary if one wants to allow the
;; reversing of all floating point jumps. Some floating point jumps
;; can be illegal to reverse according to the IEEE standard (if one
;; needs to handle illegal operands correctly).
;; On Thor an exception is raised in case of an illegal operand, so there
;; is no need for special caution when reversing the jumps - just do it!
```

```
;; (Conditional jumps are trichotomous.)

(define_insn "*condjump_reversed"
  [(set (pc)
        (if_then_else (match_operator 1 "comparison_operator"
                                        [(cc0) (const_int 0)])
                      (pc)
                      (label_ref (match_operand 0 "" ""))))]
  ""
  "*
{
  output_condjump (operands[0], operands[1], 1);

  /* if the previous instruction was a compare we must emit an
     MTOS to restore the stack in the first delay slot. */
  if (thor_compare_need_stack_adjust == 1)
    output_asm_insn (\"MTOS 1\", operands);
  else if (thor_compare_need_stack_adjust == 2)
    output_asm_insn (\"MTOS 2\", operands);
  else
    output_asm_insn (\"NOP\", operands);

  thor_compare_need_stack_adjust = 0;

  /* The second delay slot will never be filled: */
  return \"NOP\";
}"
[(set_attr "type" "condjump")
 (set_attr "length" "3")])

;;-------------------------------------------------- call

(define_insn "call_pop"
  [(call (match_operand:QI 0 "memory_operand" "Q,m")
         (match_operand:QI 1 "general_operand" "g,g"))
   (set (reg:QI 2) (plus:QI (reg:QI 2)
;; Beware! Must be STACK_POINTER_REGNUM (== 2)
                              (match_operand:QI 3 "immediate_operand" "i,i"))))]
  ""
  "*
{
  char *callstring = \"\";

  if (which_alternative == 0)
    callstring = \"CALL %0\;NOP\;NOP\";
  else
    output_indirect_call (XEXP (operands[0], 0));

  thor_top_offset -= INTVAL (operands[3]);
  return callstring;
}"
[(set_attr "type" "call,jumpx")
 (set_attr "length" "3,6")])

(define_insn "call_value_pop"
  [(set (match_operand 0 "top_register_operand" "=r,r")
        (call (match_operand:QI 1 "memory_operand" "Q,m")
              (match_operand:QI 2 "general_operand" "g,g")))
   (set (reg:QI 2) (plus:QI (reg:QI 2)
;; Beware! Must be STACK_POINTER_REGNUM (== 2)
                              (match_operand:QI 4 "immediate_operand" "i,i"))))]
  ""
  "*
{
  char *callstring = \"\";
```

A.1 thor.md, continued

```
  if (which_alternative == 0)
    callstring = \"CALL %1\;NOP\;NOP\";
  else
    output_indirect_call (XEXP (operands[1], 0));

  thor_top_offset -= INTVAL (operands[4]);
  return callstring;
}"
[(set_attr "type" "call,jumpx")
 (set_attr "length" "3,6")])

(define_insn "call"
  [(call (match_operand:QI 0 "memory_operand" "Q,m")
           (match_operand:QI 1 "general_operand" "g,g"))]
  ""
  "*
{
  if (which_alternative == 0)
    return \"CALL %0\;NOP\;NOP\";

  output_indirect_call (XEXP (operands[0], 0));
  return \"\";
}"
[(set_attr "type" "call,jumpx")
 (set_attr "length" "3,6")])

;; The call_value must clean up the stack afterwards. It must pop
;; the return value from the top of the stack to the TOP register
;; and also output an extra MTOS to compensate for this pop instruction.

(define_insn "call_value"
  [(set (match_operand 0 "top_register_operand" "=r,r")
          (call (match_operand:QI 1 "memory_operand" "Q,m")
                (match_operand:QI 2 "general_operand" "g,g")))]
  ""
  "*
{
  if (which_alternative == 0)
    return \"CALL %1\;NOP\;NOP\;POP %2\;MTOS -1\";

  output_indirect_call (XEXP (operands[1], 0));
  return \"POP %2\;MTOS -1\";
}"
[(set_attr "type" "call,jumpx")
 (set_attr "length" "5,8")])

;;---------------------------------------------------- Misc insns

;; UNSPEC_VOLATILE is considered to use and clobber all hard registers and
;; all of memory.  This blocks insns from being moved across this point.

(define_insn "blockage"
  [(unspec_volatile [(const_int 0)] 0)]
  ""
  "")

(define_insn "nop"
  [(const_int 0)]
  ""
  "NOP"
[(set_attr "type" "nop")])

(define_insn "indirect_jump"
  [(set (pc) (match_operand:QI 0 "general_operand" "I,Qt,m"))]
  ""
  "*
{
  if (which_alternative == 0)
    output_asm_insn (\"PSHI %0\", operands);
  else if (which_alternative == 1)
    output_asm_insn (\"PSH %0\", operands);
  else
    output_complex_push (operands[0]);

  return \"SL 1\;POPR PC\;NOP\;NOP\";
}"
[(set_attr "type" "jumpx,jumpx,jumpx")
 (set_attr "length" "5,5,8")])

;; casesi: actually there is no case-instruction but we can
;;         generate better code in this way compared to a tablejump.
;;
;; operand 0 is index
;; operand 1 is the minimum bound
;; operand 2 is the maximum bound - minimum bound
;; operand 3 is CODE_LABEL for the table;
;; operand 4 is the CODE_LABEL to go to if index out of range.
;;
;; The actual jump-table stores absolute adresses. The first
;; entry is the address of the table itself.

(define_insn "casesi"
  [(set (pc)
     (if_then_else (leu (minus:QI
                            (match_operand:QI 0 "general_operand" "I,Qt,m")
                            (match_operand:QI 1 "immediate_operand" "i,i,i"))
                           (match_operand:QI 2 "general_operand" "m,m,m"))
                      (mem:QI (plus:QI (pc)
                                       (minus:QI (match_dup 0)
                                                 (match_dup 1))))
                      (label_ref:QI (match_operand 4 "" ""))))
   (use (label_ref (match_operand 3 "" "")))]
  ""
  "*
{
  if (which_alternative == 0)
    output_asm_insn (\"PSHI %0\", operands);
  else if (which_alternative == 1)
    output_asm_insn (\"PSH %0\", operands);
  else
    output_complex_push (operands[0]);

  output_casesi (operands);
  return \"\";
}"
[(set_attr "type" "jumpx,jumpx,jumpx")
 (set_attr "length" "11,11,14")])

;; An unused casesi-pattern. Only here to be recognised in
;; instatiate_virtual_regs_1, who try to change only the first
;; operand and then rerecognise the instruction. The insn
;; will then fail with the match_dup part.

(define_insn "*casesi_instantiate_virtual_regs_1"
  [(set (pc)
     (if_then_else (leu (minus:QI
                            (match_operand:QI 0 "general_operand" "g")
                            (match_operand:QI 1 "immediate_operand" "g"))
                           (match_operand:QI 2 "general_operand" "g"))
                      (mem:QI (plus:QI (pc)
                        (minus:QI (match_operand:QI 5 "general_operand" "g")
                                  (match_operand:QI 6 "immediate_operand" "g"))))
```

A.1 thor.md, continued

```
                     (label_ref:QI (match_operand 4 "" ""))))
   (use (label_ref (match_operand 3 "" ""))))]
  ""
  "error - wrong insn (casesi) chosen");

/****************************************************************/
/** PEEPHOLES                                                  **/
/****************************************************************/

;;-------------------------------------------------- push, pop
;; A nop insn that moves itself to itself, is in our
;; case represented by a push followed by a pop to
;; the same location/register. Just remove it!

(define_peephole
  [(set (match_operand:QF 0 "tmp_register_operand" "")
          (match_operand:QF 1 "general_operand" ""))
   (set (match_dup 1)
          (match_operand:QF 2 "tmp_register_operand" ""))]
   ""
   "\\t\\t; PEEP - push,pop"
[(set_attr "cc" "unchanged")])

(define_peephole
  [(set (match_operand:QI 0 "tmp_register_operand" "")
          (match_operand:QI 1 "general_operand" ""))
   (set (match_dup 1)
          (match_operand:QI 2 "tmp_register_operand" ""))]
   ""
   "\\t\\t; PEEP - push,pop"
[(set_attr "cc" "unchanged")])

;;-------------------------------------------------- pop, push
;; This represent an intermediate store-load, ie, a
;; pop-push to the same register/memory-address, where the
;; register/memory-address dies after the push.

(define_peephole
  [(set (match_operand:QF 0 "general_operand" "")
          (match_operand:QF 1 "tmp_register_operand" ""))
   (set (match_operand:QF 2 "tmp_register_operand" "")
          (match_dup 0))]
   "dead_p (insn, operands[0], 0)"
   "\\t\\t; PEEP - pop,push (dead)"
[(set_attr "cc" "unchanged")])

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (match_operand:QI 2 "tmp_register_operand" "")
          (match_dup 0))]
   "dead_p (insn, operands[0], 0)"
   "\\t\\t; PEEP - pop,push (dead)"
[(set_attr "cc" "unchanged")])

;; Same as above, but with a normal push following the pop:

(define_peephole
  [(set (match_operand:QF 0 "general_operand" "")
          (match_operand:QF 1 "tmp_register_operand" ""))
   (set (match_operand:QF 2 "push_operand" "")
          (match_dup 0))]
   "dead_p (insn, operands[0], 1)"
   "*
{
  output_asm_insn(\"\\t\\t; PEEP - pop,normal push (dead)\", operands);
  thor_top_offset++;
  return \"\";
}")

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (match_operand:QI 2 "push_operand" "")
          (match_dup 0))]
   "dead_p (insn, operands[0], 1)"
   "*
{
  output_asm_insn(\"\\t\\t; PEEP - pop,normal push (dead)\", operands);
  thor_top_offset++;
  return \"\";
}")

;;-------------------------------------------------- tos add
;; When the address of a specific position in the stack is needed,
;; a frame pointer reference will be generated in the rtl.
;; The frame pointer is later eliminated and the result
;; looks something like this:
;;
;;    PSHR TOS
;;    ADDI xxx
;;    ADDI 1
;;    POP  intermediate location
;;    PSH  intermediate location
;;    ADDI yyy
;;
;; This is changed by the peephole into:
;;
;;    PSHR TOS
;;    ADDI zzz

(define_peephole
  [(set (match_operand:QI 0 "tmp_register_operand" "")
          (plus:QI (match_operand:QI 1 "tos_register_operand" "")
                   (match_operand:QI 2 "immediate_operand" "")))
   (set (match_dup 0)
          (plus:QI (match_dup 0)
                   (match_operand:QI 3 "immediate_operand" "")))
   (set (match_operand:QI 4 "general_operand" "")
          (match_dup 0))
   (set (match_dup 0)
          (match_dup 4))
   (set (match_dup 0)
          (plus:QI (match_dup 0)
                   (match_operand:QI 5 "immediate_operand" ""))))]
   "dead_p (prev_nonnote_insn (insn), operands[4], 0)"
   "*
{
  operands[9] = gen_rtx (PLUS, QImode,
                         operands[5],
                         gen_rtx (PLUS, QImode,
                                  operands[3],
                                  operands[2]));
  output_asm_insn(\"PSHR TOS\\t; PEEP - tos add\", operands);
  output_asm_insn(\"ADDI %9\", operands);
  return \"\";
}")

;; Sometimes it's more complicated than the previous case.
;; It's probably hard to catch all these cases, but they
;; all start with the same three instruction:
;;
```

A.1 thor.md, continued

```
;;    PSHR TOS
;;    ADDI xxx
;;    ADDI 1
;;
;; This is changed by this peephole into:
;;
;;    PSHR TOS
;;    ADDI zzz

(define_peephole
  [(set (match_operand:QI 0 "tmp_register_operand" "")
          (plus:QI (match_operand:QI 1 "tos_register_operand" "")
                   (match_operand:QI 2 "immediate_operand" "")))
   (set (match_dup 0)
          (plus:QI (match_dup 0)
                   (match_operand:QI 3 "immediate_operand" ""))))]
   ""
   "*
{
  operands[9] = gen_rtx (PLUS, QImode,
                           operands[3],
                           operands[2]);
  output_asm_insn(\"PSHR TOS\\t; PEEP - tos add\", operands);
  output_asm_insn(\"ADDI %9\", operands);
  return \"\";
}")

;;-------------------------------------------------- PSH, POPX
;; When a simple push instruction is in front of an indirect pop
;; instruction, the push (PSH) can be moved into the indirect
;; pop and replace a NOP instruction.
;;
;;   PSH xxx              PSH vvv      (vvv = yyy - 1 if stack rel)
;;   PSH yyy      -->     MTOS 1
;;   MTOS 1               PSH xxx
;;   NOP                  POPX zzz
;;   POPX zzz

(define_peephole
  [(set (match_operand:QI 0 "tmp_register_operand" "")
          (match_operand:QI 1 "simple_operand" ""))
   (set (match_operand:QI 2 "complex_operand" "")
          (match_dup 0))]
   ""
   "*
{
  cc_status.value1 = operands[2];
  cc_status.value2 = operands[1];
  output_complex_pop_with_push (operands, QImode);
}")

(define_peephole
  [(set (match_operand:QF 0 "tmp_register_operand" "")
          (match_operand:QF 1 "simple_operand" ""))
   (set (match_operand:QF 2 "complex_operand" "")
          (match_dup 0))]
   ""
   "*
{
  cc_status.value1 = operands[2];
  cc_status.value2 = operands[1];
  output_complex_pop_with_push (operands, QFmode);
}")

;;-------------------------------------------------- POP, PSHX
;; A pop followed by a indirect push can be simplified if
;; the pop location is used as an address in the indirect push
;; and this location/address is temporary (dies).
;;
;;   POP temporary loc          MTOS 1
;;   PSH temporary loc   -->    NOP
;;   MTOS 1                     PSHX xxx
;;   NOP
;;   PSHX xxx

;; Two templates are needed, because 'plus' is missing if offset == 0.

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (match_operand:QI 3 "tmp_or_push_operand" "")
          (mem:QI (match_dup 0)))]
   "dead_p (insn, operands[0], 0)"
   "*
{
  cc_status.value1 = operands[3];
  cc_status.value2 = gen_rtx (MEM, QImode, operands[0]);

  if (push_operand (operands[3], QImode))
    thor_top_offset++;

  return \"\\t\\t; PEEP - pop,push base address (dead)\;MTOS 1\;NOP\;PSHX 0\";
}")

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (match_operand:QI 3 "tmp_or_push_operand" "")
          (mem:QI (plus:QI (match_dup 0)
                           (match_operand:QI 2 "const_int_operand" ""))))]
   "dead_p (insn, operands[0], 0)"
   "*
{
  cc_status.value1 = operands[3];
  cc_status.value2 = gen_rtx (MEM, QImode,
                              gen_rtx (PLUS, QImode,
                                       operands[0],
                                       operands[2]));

  if (push_operand (operands[3], QImode))
    thor_top_offset++;

  return \"\\t\\t; PEEP - pop,push base address (dead)\;MTOS 1\;NOP\;PSHX %2\";
}")

;; QFmode:

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (match_operand:QF 3 "tmp_or_push_operand" "")
          (mem:QF (match_dup 0)))]
   "dead_p (insn, operands[0], 0)"
   "*
{
  cc_status.value1 = operands[3];
  cc_status.value2 = gen_rtx (MEM, QFmode, operands[0]);

  if (push_operand (operands[3], QFmode))
    thor_top_offset++;

  return \"\\t\\t; PEEP - pop,push base address, QF (dead)\;MTOS 1\;NOP\;PSHX 0\";
```

A.1 thor.md, continued

```
}")

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (match_operand:QF 3 "tmp_or_push_operand" "")
          (mem:QF (plus:QI (match_dup 0)
                           (match_operand:QI 2 "const_int_operand" "")))))]
   "dead_p (insn, operands[0], 0)"
   "*
{
  cc_status.value1 = operands[3];
  cc_status.value2 = gen_rtx (MEM, QFmode,
                                gen_rtx (PLUS, QImode,
                                         operands[0],
                                         operands[2]));

  if (push_operand (operands[3], QFmode))
    thor_top_offset++;

  return \"\\t\\t; PEEP - pop,push base address, QF (dead)\;MTOS 1\;NOP\;PSHX
%2\";
}")

;;--------------------------------------------------- POP, CMP
;; A pop instruction (POP) followed by a compare instruction,
;; where the POP is to the same temporary location as the first
;; PSH of the compare, can be simplified.
;;
;;   POP temporary loc  -->   CMP yyy
;;   PSH temporary loc
;;   CMP yyy

;; signed version:

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (cc0)
     (compare (match_dup 0)
                (match_operand:QI 2 "simple_operand" "")))]
  "! next_cc0_user_unsigned_jump_p (insn)
   && dead_p (insn, operands[0], 0)"
  "*
{
  thor_compare_need_stack_adjust = 1;
  cc_status.value1 = gen_rtx (COMPARE, VOIDmode, operands[0], operands[2]);

  if (GET_CODE (operands[2]) == CONST_INT
      && CONST_OK_FOR_LETTER_P (INTVAL (operands[2]), 'K'))
    return \"\\t\\t; PEEP - pop,push compare (dead)\;CMPI %2\";
  else
    return \"\\t\\t; PEEP - pop,push compare (dead)\;CMP %z2\";
}")

;; unsigned:

(define_peephole
  [(set (match_operand:QI 0 "general_operand" "")
          (match_operand:QI 1 "tmp_register_operand" ""))
   (set (cc0)
     (compare (match_dup 0)
                (match_operand:QI 2 "simple_operand" "")))]
  "next_cc0_user_unsigned_jump_p (insn)
   && dead_p (insn, operands[0], 0)"
  "*
{
  thor_compare_need_stack_adjust = 2;
  cc_status.value1 = gen_rtx (COMPARE, VOIDmode, operands[0], operands[2]);

  return \"\\t\\t; PEEP - pop,push compare (dead)\;CMPU %z2\;PSHI 0\;CMP 0\";
}")

;; QFmode:

(define_peephole
  [(set (match_operand:QF 0 "general_operand" "")
          (match_operand:QF 1 "tmp_register_operand" ""))
   (set (cc0)
          (compare (match_dup 0)
                   (match_operand:QF 2 "simple_operand" "")))]
  "dead_p (insn, operands[0], 0)"
  "*
{
  thor_compare_need_stack_adjust = 1;
  cc_status.value1 = gen_rtx (COMPARE, VOIDmode, operands[0], operands[2]);

  return \"\\t\\t; PEEP - pop,push compare (dead)\;CMPF %z2\";
}")

;;--------------------------------------------------- pop, test
;; A simple pop followed by a test and then a conditional jump,
;; can often be simplified.
;;
;;  POP xxx        TEST yyy
;;  TEST yyy  -->  JRcc
;;  JRcc           POP xxx
;;  NOP            NOP
;;  NOP
;;
;; Sometimes the TEST instruction can be removed (depending on
;; cc_prev_status). Sometimes the POP can be changed into
;; an MTOS 1 instruction (when xxx dies).
;;
;; The condition in the template is always true. It has the side-
;; effect of checking if operand 0 dies somewhere, and storing
;; this information for later use in the output. This is
;; necessary because the death-notes are no longer available
;; when the instructions are output.

(define_peephole
 [(set (match_operand:QI 0 "simple_operand" "")
       (match_operand:QI 1 "tmp_register_operand" ""))
  (set (cc0)
       (match_operand:QI 2 "stack_relative_operand" ""))
  (set (pc)
       (if_then_else (match_operator 3 "comparison_operator"
                                       [(cc0) (const_int 0)])
                       (label_ref (match_operand 4 "" ""))
                       (pc)))]
  "(thor_peep_pop_operand_dies =
    (dead_p (prev_nonnote_insn (insn), operands[0], 0)
     || dead_p (prev_nonnote_insn (prev_nonnote_insn (insn)), operands[0], 0)))
   || 1"
  "*
{
  int pop_before_test = 0;

  output_asm_insn (\"\\t\\t; PEEP - pop, test, QI\", 0);

  /* Update cc_status (CC_COPY due to pop instruction): */
  if (cc_status.value1
```

A.1 thor.md, continued

```
      && REG_P (cc_status.value1)
      && REGNO (cc_status.value1) == TMP_REGNUM)
    {
      cc_status.value1 = operands[0];
    }

 /* check if the test instruction is needed: */
  if (! ((cc_status.value1 != 0
            && rtx_equal_p (operands[2], cc_status.value1))
           || (cc_status.value2 != 0
               && rtx_equal_p (operands[2], cc_status.value2))))
    {
      /* A test instruction is needed, check if the pop
           instruction uses the same operand as the test does.
           In that case we must output the pop instruction
           before the test (and we will win nothing). */
      if (rtx_equal_p (operands[2], operands[0]))
          {
            output_asm_insn (\"POP %0\", operands);
            output_asm_insn (\"TEST %2\", operands);
            pop_before_test = 1;
          }
      else
          output_asm_insn (\"TEST %z2\", operands);

      /* update cc_status: */
      CC_STATUS_INIT;
      cc_status.flags |= CC_NO_OVERFLOW;
      cc_status.value1 = operands[2];
    }

  output_condjump (operands[4], operands[3], 0);

  /* Output the pop in the delay-slot. If the pop operand dies we
     can make it into an MTOS 1 (preferable). If pop_before_test
     is true, the pop instruction was needed before the test. */
  if (pop_before_test)
    output_asm_insn (\"NOP\", 0);
  else if (thor_peep_pop_operand_dies)
    output_asm_insn (\"MTOS 1\", operands);
  else
    output_asm_insn (\"POP %z0\", operands);

  /* The second delay slot will never be filled: */
  return \"NOP\";
}"
[(set_attr "cc" "unchanged")])

;; QFmode, this is only a delay-slot filling:
;; The only danger is if we are testing the same operand as
;; are popping (very unlikely but possible), i.e. operand 0
;; and 2 represents the same operand with different machine
;; modes. The condition in the template takes care of this case.

(define_peephole
 [(set (match_operand:QF 0 "simple_operand" "")
       (match_operand:QF 1 "tmp_register_operand" ""))
  (set (cc0)
       (match_operand:QI 2 "stack_relative_operand" ""))
  (set (pc)
       (if_then_else (match_operator 3 "comparison_operator"
                                       [(cc0) (const_int 0)])
                       (label_ref (match_operand 4 "" ""))
                       (pc)))]
  "! (top_register_operand (operands[0], VOIDmode)
      && top_register_operand (operands[2], VOIDmode)
      || (GET_CODE (operands[0]) == MEM
          && GET_CODE (operands[2]) == MEM
          && rtx_equal_p (XEXP (operands[0], 0), XEXP (operands[2], 0))))"
  "*
{
  output_asm_insn (\"\\t\\t; PEEP - pop, test, QF\", 0);

  output_asm_insn (\"TEST %z2\", operands);

  output_condjump (operands[4], operands[3], 0);

  /* Output the pop in the delay-slot: */
  output_asm_insn (\"POP %0\", operands);

  /* The second delay slot will never be filled: */
  return \"NOP\";
}")

;;------------------------------------------------- call_value, mtos
;; A call_value is usually followed by an MTOS instruction.
;; As the call_value also contains an internal MTOS instruction,
;; we can simplify things by combining the two MTOS instructions.
;;
;;  CALL fun         CALL fun
;;  NOP              NOP
;;  NOP        -->   NOP
;;  POP x            POP x
;;  MTOS -1          MTOS x-1
;;  MTOS x
;;

(define_peephole
  [(set (match_operand 0 "top_register_operand" "")
         (call (match_operand:QI 1 "memory_operand" "")
               (match_operand:QI 2 "general_operand" "")))
   (set (match_operand:QI 3 "tos_register_operand" "")
         (plus:QI (match_dup 3)
                  (match_operand:QI 4 "immediate_operand" "")))]
  ""
  "*
{
  output_asm_insn (\"\\t\\t; PEEP - call_value, mtos\", 0);
  if (EXTRA_CONSTRAINT (operands[1], 'Q'))
    output_asm_insn (\"CALL %1\;NOP\;NOP\", operands);
  else
    output_indirect_call (XEXP (operands[1], 0));

  thor_top_offset -= INTVAL (operands[4]);

  operands[5] = GEN_INT (INTVAL (operands[4]) - 1);
  if (INTVAL (operands[5]) > 0)
    return \"POP %2\;MTOS %5\";
  else
    return \"POP %2\";
}"
[(set_attr "type" "call")
 (set_attr "length" "5")])
```

A.2 thor.h

```
/*
 *           thor.h
 *
 *
 * Purpose
 *           The entire file consists of macros which each describe hardware
 *           constraints in Thor. Such as addressing modes, internal
 *           registers, word size etc.
 * Operations
 *           With each macro follows a description of it's function
 * Usage
 *           Used by GNU CC when the compiler is built.
 *           The macros are utilized in almost every source file.
 *
 * I/O       None
 *
 *
 *
 * Machine   (No dependencies)
 *
 *
 * Compiler  (No dependencies)
 *
 *
 * Author    Harry Gunnarsson, Thomas Lundqvist / 951115
 *
 * Revised       by          date    remarks
 *
 *   1.0.0       HG, TL      951115  New version
 *
 */

/* Definitions of target machine for GNU compiler.  Thor version. */

/* Thor, the Saab Ericsson Space Microprocessor.

   The Thor microprocessor is a 32 bit RISC. It is targeted for
   embedded computers in real time systems and for the execution
   of programs written in Ada. It is designed for highly
   dependable space and military applications.

   Instead of a more traditional register based microprocessor,
   it has a stack oriented instruction set. This makes the
   generated code compact. */

/******************************************************************/
/** Prototypes and externals for thor.c **/
/******************************************************************/

/* For more info, see thor.c */

extern int thor_top_offset;
extern int *thor_top_offset_at;
extern int thor_compare_need_stack_adjust;
extern int thor_peep_pop_operand_dies;

extern void thor_fatal ();
extern int tmp_register_operand ();
extern int top_register_operand ();
extern int tos_register_operand ();
extern int pop_operand ();
extern int simple_operand ();
extern int complex_operand ();
extern int stack_relative_operand ();
extern int indirect_jump_operand ();
extern int next_cc0_user_unsigned_jump_p ();
extern void emit_push ();
extern void emit_pop ();
extern int check_swap_operands ();
extern int emit_move_sequence ();
extern int emit_add_sequence ();
extern int emit_sub_sequence ();
extern int emit_commutative_sequence ();
extern int emit_non_commutative_sequence ();
extern int emit_ashl_sequence ();
extern int dead_p ();
extern void output_complex_push ();
extern void output_complex_pop ();
extern void output_complex_pop_with_push ();
extern void output_condjump ();
extern void output_indirect_call ();
extern void output_casesi ();
extern void notice_update_cc ();
extern void output_hex ();
extern void output_ascii ();
extern void set_new_thor_top_offset ();
extern void output_char ();
extern void print_operand ();

/******************************************************************/
/** EXTRA STUFF FOR THOR                              **/
/******************************************************************/

/* 'n_basic_blocks' get its value in flow.c. It is used here
    to indicate that the combine-pass hasn't been run yet. */

extern int n_basic_blocks;
#define THOR_BEFORE_COMBINE (n_basic_blocks == 0)

/* On Thor we want the jump2-pass to preserve death-info notes,
   to make the peep-hole optimization work better.
   We could have used PRESERVE_DEATH_INFO_REGNO_P but it does
   not work (won't compile). */

#define THOR_PRESERVE_DEATH_INFO_REGNO_P(REGNO)1
/*  ((REGNO) == TOP_REGNUM) */

/* In stmt.c a new macro THOR_CASE_VALUES_SPARSENESS has been introduced.
   It is used in combination with CASE_VALUES_THRESHOLD to determine
   when jump-tables should be chosen instead of branch-trees. The
   value give the maximum gap between cases allowed in a jump-
   table. The default is 10. A smaller value can be appropriate
   if size is more important than speed. */
/* On Thor we want to favour small code-size rather than speed. A value
   of 10 can cause huge jump-tables to be generated with a lot of
   unused entries. A lower limit favours the branch-trees in these
   cases. */

#define THOR_CASE_VALUES_SPARSENESS 4

/* In stmt.c we had to change the way casesi works. We want
   QImode instead of SImode. */

#define THOR_CASESI_QIMODE

/* In 'function.c' we want to inhibit unnessecary copy instructions
   of the parameters to registers when entering a function.*/

#define THOR_NO_INITIAL_REG 1

/* 'Init_reload' measures 'indirect_spill_levels' wrongly, therefore
   we make it count the number of levels right. */
```

A.2 thor.h, continued

```
#define THOR_HARD_FRAME_POINTER_INIT_RELOAD

/* In 'find_reloads_address' in reload.c, a pseudo-reg can be
   changed into an equivalent constant. This fails in the case with a
   PLUS (SYMBOL_REF, CONST_INT) - memory address. There should be
   a CONST before this expression. The macro THOR_RELOAD_CHECK_FOR_CONST
   enables code to correct this case. */

#define THOR_RELOAD_CHECK_FOR_CONST

/* Bug fix in module 'reorg.c' concerning delay-slots */

#define THOR_DELAY_SLOT_FIX

/* In 'fixup_var_refs_1' in function.c, the function 'single_set'
   is called with the wrong parameter. Probably a bug! */

#define THOR_FIXUP_VAR_REFS_1

/*****************************************************************/
/** THINGS MISSING IN INFO PAGES  **/
/*****************************************************************/

/* A C statement that outputs something just before the starting
   label for a function is output. */
/* On Thor we need to make sure all functions is aligned to
   a 32 bit (word) boundary. Otherwise, it will not be possible
   to use its address in a normal pointer (pointers point to
   bytes which in our case are 32 bits). */

#define ASM_OUTPUT_FUNCTION_PREFIX(FILE, NAME)\
  if (get_pool_size () == 0)       \
    fprintf((FILE), "\tDATA 0\t\t; align function\n");

/* Tell GCC to put a space after -L when generating such options.  */

#define SPACE_AFTER_L_OPTION

/* Tell init_emit to call our function, to do function level
   initialization. */

#define INIT_EXPANDERS thor_init_expanders ()

/*****************************************************************/
/** THINGS MENTIONED BUT NOT ACTUALLY BELONGING IN THIS FILE**/
/*****************************************************************/

/* Define this macro to be a C string representing the suffix for
   object files on your machine. If you do not define this macro,
   GNU CC will use `.o' as the suffix for object files. */
/* According to the info pages, this macro should be defined
   in the 'config' file (xm-file), not in this file. We find
   this somewhat odd because we only want to change the object
   suffix for our target, not for the host machine (sparc).
   The macro is used in the driver part of GCC (gcc.c) and
   should, in our opinion, belong to the DRIVER part below.

   ".obj" is the suffix used by the Oden assembler and linker. */

#define OBJECT_SUFFIX ".obj"

/*****************************************************************/
/** DRIVER                                                     **/
/*****************************************************************/

/* A string-valued C expression which is nonempty if the linker needs
   a space between the `-L' or `-o' option and its argument. */

#define SWITCHES_NEED_SPACES "Yes"

/* A C string constant that tells the GNU CC driver program options to
   pass to the linker. It can also specify how to translate options
   you give to GNU CC into options for GNU CC to pass to the linker.
   `LIB_SPEC' is used at the end of the command given to the linker.

   If this macro is not defined, a default is provided that loads the
   standard C library from the usual place. See `gcc.c'. */
/* The thor-linker implements libraries in the form of directories
   filled with '.obj' files. So instead of the normal LIB_SPEC give
   the -L option for each directory in startfile_prefixes.
   The ld script will then convert this to -l options to the 'link'
   command. */

#define LIB_SPEC "%D"

/* Another C string constant that tells the GNU CC driver program how
   and when to place a reference to `libgcc.a' into the linker
   command line. This constant is placed both before and after the
   value of `LIB_SPEC'.

   If this macro is not defined, the GNU CC driver provides a default
   that passes the string `-lgcc' to the linker unless the `-shared'
   option is specified. */
/* The thor-linker finds the library libgcc.obj in the directory
   specified by 'LIB_SPEC'. 'LIBGCC_SPEC' is not needed and would
   just confuse the ld script. */

#define LIBGCC_SPEC ""

/* C string constant that tells the GNU CC driver program how
   and when to place a reference to a startfile, like 'crt0.o',
   into the linker command line. */
/* The thor-linker finds the object file crt0.obj in the subdir
   specified by 'LIB_SPEC'. 'crt0.obj' should not be mentioned
   explicitly. */

#define STARTFILE_SPEC ""

/* Define this macro meaning that `gcc' should find the library
   `libgcc.a' by hand, rather than passing the argument `-lgcc' to
   tell the linker to do the search; also, `gcc' should not generate
   `-L' options to pass to the linker (as it normally does). */
/* The thor-linker can handle all things with the LIB_SPEC, don't
   give any more arguments, it will only confuse the ld script. */

#define LINK_LIBGCC_SPECIAL

/*****************************************************************/
/** RUN-TIME TARGET                                           **/
/*****************************************************************/

/* Names to predefine in the preprocessor for this target machine.  */

#define CPP_PREDEFINES "-Dthor -D__thor__ -Acpu(thor) -Amachine(thor)"

/* This flag should be present! */

extern int target_flags;

/*   This macro defines names of command options to set and clear bits
     in `target_flags'.  Its definition is an initializer with a
```

A.2 thor.h, continued

```
     subgrouping for each command option.*/

#ifndef TARGET_DEFAULT
#define TARGET_DEFAULT 0
#endif

#define TARGET_SWITCHES  \
  {{ "", TARGET_DEFAULT}}

/*   This macro is a C statement to print on `stderr' a string
     describing the particular machine description choice.  Every
     machine description should define `TARGET_VERSION'.*/

#define VERSION_INFO "Thor C Cross Compiler, version 1.0.0"

#if !defined(__DATE__)
#define TARGET_VERSION fprintf (stderr, " (%s)", VERSION_INFO)
#else
#define TARGET_VERSION fprintf (stderr, " (%s, %s)", VERSION_INFO, __DATE__)
#endif

/* OVERRIDE_OPTIONS takes care of the following:
     - setting: flag_omit_frame_pointer, we must get rid
       of the frame pointer.
     - resetting: flag_defer_pop, if any pop is deferred
       we will lose control of our TOP-register. */

#define OVERRIDE_OPTIONS                  \
{                                                        \
  if (flag_omit_frame_pointer)            \
    warning("-fomit-frame-pointer not needed, always active!");\
  flag_omit_frame_pointer = 1;            \
  if (flag_defer_pop)                             \
    warning("-fdefer-pop not allowed");\
  flag_defer_pop = 0;                             \
}

/* flag_omit_frame_pointer is activated in OVERRIDE_OPTIONS,
   so reset it now to avoid warning. */

#define OPTIMIZATION_OPTIONS(LEVEL)\
{                                                 \
  flag_defer_pop = 0;             \
  flag_omit_frame_pointer = 0;\
}

/* Define this macro if debugging can be performed even without a
   frame pointer. */

#define CAN_DEBUG_WITHOUT_FP

/*****************************************************************/
/** STORAGE LAYOUT                                             **/
/*****************************************************************/

/* Define this if most significant bit is lowest numbered
   in instructions that operate on numbered bit-fields. */

#define BITS_BIG_ENDIAN 0

/* Define this if most significant byte of a word is the lowest numbered.  */

#define BYTES_BIG_ENDIAN 0

/* Define this if most significant word of a multiword number is the lowest
   numbered.  */

#define WORDS_BIG_ENDIAN 0

/* Number of bits in an addressable storage unit.  */

#define BITS_PER_UNIT 32

/* Width in bits of a "word", which is the contents of a machine register. */

#define BITS_PER_WORD 32
#define MAX_BITS_PER_WORD 32

/* Width of a word, in units (bytes).  */

#define UNITS_PER_WORD 1
#define MAX_UNITS_PER_WORD 1

/* Width in bits of a pointer.
   See also the macro `Pmode' defined below.  */

#define POINTER_SIZE 32

/* Allocation boundary (in *bits*) for storing arguments in argument list.  */

#define PARM_BOUNDARY 32

/* Allocation boundary (in *bits*) for the code of a function.  */

#define FUNCTION_BOUNDARY 32

/* Alignment of field after `int : 0' in a structure.  */

#define EMPTY_FIELD_BOUNDARY 32

/* Every structure's size must be a multiple of this.  */

#define STRUCTURE_SIZE_BOUNDARY 32

/* A bitfield declared as `int' forces `int' alignment for the struct.  */

#define PCC_BITFIELD_TYPE_MATTERS 1

/* No data type wants to be aligned rounder than this.  */

#define BIGGEST_ALIGNMENT 32

/* No structure field wants to be aligned rounder than this.  */

#define BIGGEST_FIELD_ALIGNMENT 32

/* Set this nonzero if move instructions will actually fail to work
   when given unaligned data.  */

#define STRICT_ALIGNMENT 1

/* Let's keep the stack somewhat aligned.  */

#define STACK_BOUNDARY 32

/* An integer expression for the size in bits of the largest integer
   machine mode that should actually be used.  All integer machine
   modes of this size or smaller can be used for structures and
   unions with the appropriate sizes.  If this macro is undefined,
   `GET_MODE_BITSIZE (DImode)' is assumed. */
/* On Thor we want all structs bigger than 1 word, to use BLKmode. */
```

A.2 thor.h, continued

```
#define MAX_FIXED_MODE_SIZE GET_MODE_BITSIZE (QImode)

/*****************************************************************/
/** TYPE LAYOUT                                                **/
/*****************************************************************/

/* On Thor we set the size of all integer types to 32 bits. */

#define CHAR_TYPE_SIZE 32
#define SHORT_TYPE_SIZE 32
#define INT_TYPE_SIZE 32
#define LONG_TYPE_SIZE 32
#define LONG_LONG_TYPE_SIZE 32

/* We also set the size of all real-value types to 32 bits. */

#define FLOAT_TYPE_SIZE 32
#define DOUBLE_TYPE_SIZE 32
#define LONG_DOUBLE_TYPE_SIZE 32

/* Define this as 1 if `char' should by default be signed; else as 0.  */

#define DEFAULT_SIGNED_CHAR 1

/* A C constant expression for the integer value for escape sequence
   `\a', `\b', `\t', `\n', `\v', `\f' and `\r'. */

#define TARGET_BELL 007
#define TARGET_BS 010
#define TARGET_TAB 011
#define TARGET_NEWLINE 012
#define TARGET_VT 013
#define TARGET_FF 014
#define TARGET_CR 015

/*****************************************************************/
/** REGISTERS:     ** Register Basics      **/
/*****************************************************************/

/* Number of actual hardware registers.
   The hardware registers are assigned numbers for the compiler
   from 0 to just below FIRST_PSEUDO_REGISTER.
   All registers that the compiler knows about must be given numbers,
   even those that are not normally considered general registers.  */

#define FIRST_PSEUDO_REGISTER 4

/* 1 for registers that have pervasive standard uses
   and are not available for the register allocator. */
/* In our case we only have one general register, that can be used
   for register allocation, the TOP-register. This register resides
   on the TOP of the stack. It seems that GCC works better when it
   have at least one register to play with. In the end, the TOP-
   register is just like any other stack slot, but used more
   frequently and therefore easier to eliminate with peep-hole
   optimizations.

   The TMP-register is used internally in expand instructions. This
   is a temporary top of stack. A store into this register represents
   a PSH-instruction, a load from this register is a POP-instruction.

   The TOS-register is just the stack pointer register. The FP
   register must be included here. But we don't have anyone, and
   if it's not eliminated the assembler will fail.

                        TOP,TMP,TOS,FP.*/
#define FIXED_REGISTERS {0,  1,  1, 1}

/* 1 for registers not available across function calls.
   These must include the FIXED_REGISTERS and also any
   registers that can be used without being saved.
   The latter must include the registers where values are returned
   and the register where structure-value addresses are passed.
   Aside from that, you can include as many other registers as you like.  */

#define CALL_USED_REGISTERS {1, 1, 1, 1}

/* Our own define for the TOP-reg */

#define TOP_REGNUM 0

#define TMP_REGNUM 1

/*****************************************************************/
/** REGISTERS:     ** Allocation Order     **/
/*****************************************************************/

/*****************************************************************/
/** REGISTERS:     ** Values in Registers  **/
/*****************************************************************/

/* Return number of consecutive hard regs needed starting at reg REGNO
   to hold something of mode MODE.
   This is ordinarily the length in words of a value of mode MODE
   but can be less for certain modes in special long registers. */

#define HARD_REGNO_NREGS(REGNO, MODE)   \
 ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)

/* Value is 1 if hard register REGNO can hold a value of machine-mode MODE.*/

#define HARD_REGNO_MODE_OK(REGNO, MODE)\
  (MODE == QImode || MODE == QFmode)

/* Value is 1 if it is a good idea to tie two pseudo registers
   when one has mode MODE1 and one has mode MODE2.
   If HARD_REGNO_MODE_OK could produce different values for MODE1 and MODE2,
   for any hard reg, then this must be 0 for correct output.  */

#define MODES_TIEABLE_P(MODE1, MODE2)  1

/*****************************************************************/
/** REGISTERS:     ** Leaf Functions       **/
/*****************************************************************/

/*****************************************************************/
/** REGISTERS:     ** Stack Registers      **/
/*****************************************************************/

/*****************************************************************/
/** REGISTERS CLASSES                                 **/
/*****************************************************************/

/* Define the classes of registers for register constraints in the
   machine description.  Also define ranges of constants.

   One of the classes must always be named ALL_REGS and include all hard regs.
   If there is more than one class, another class must be named NO_REGS
   and contain no registers.

   The name GENERAL_REGS must be the name of a class (or an alias for
   another name such as ALL_REGS).  This is the class of registers
```

A.2 thor.h, continued

```
   that is allowed by "g" or "r" in a register constraint.
   Also, registers outside this class are allocated only when
   instructions express preferences for them.

   The classes must be numbered in nondecreasing order; that is,
   a larger-numbered class must never be contained completely
   in a smaller-numbered class.

   For any two classes, it is very desirable that there be another
   class that represents their union.  */

enum reg_class {NO_REGS, TOP_REG, TMP_REG, TOPTMP_REGS,
                 ALL_REGS, LIM_REG_CLASSES };

#define N_REG_CLASSES (int) LIM_REG_CLASSES

/* Since GENERAL_REGS is the same class as ALL_REGS,
   don't give it a different class number; just make it an alias.  */

#define GENERAL_REGS TOPTMP_REGS

/* Give names of register classes as strings for dump file.   */

#define REG_CLASS_NAMES \
 {"NO_REGS", "TOP_REG", "TMP_REG", "TOPTMP_REGS", "ALL_REGS" }

/* Define which registers fit in which classes.
   This is an initializer for a vector of HARD_REG_SET
   of length N_REG_CLASSES.  */

#define REG_CLASS_CONTENTS {0x0000, 0x0001, 0x0002, 0x0003, 0xffff}

/* The same information, inverted:
   Return the class number of the smallest class containing
   reg number REGNO.  This could be a conditional expression
   or could index an array.  */

#define REGNO_REG_CLASS(REGNO)\
  ((REGNO) == 0 ? TOP_REG  \
   : (REGNO) == 1 ? TMP_REG : ALL_REGS)

/* The class value for index registers, and the one for base regs.  */

#define INDEX_REG_CLASS ALL_REGS
#define BASE_REG_CLASS ALL_REGS

/* Get reg_class from a letter such as appears in the machine description.  */

#define REG_CLASS_FROM_LETTER(C)\
  ((C) == 't' ? TOP_REG : NO_REGS )

/*   Normally the compiler avoids choosing registers that have been
     explicitly mentioned in the rtl as spill registers (these
     registers are normally those used to pass parameters and return
     values).  However, some machines have so few registers of certain
     classes that there would not be enough registers to use as spill
     registers if this were done.

     Define `SMALL_REGISTER_CLASSES' on these machines.  When it is
     defined, the compiler allows registers explicitly used in the rtl
     to be used as spill registers but avoids extending the lifetime of
     these registers.

     It is always safe to define this macro, but if you unnecessarily
     define it, you will reduce the amount of optimizations that can be
     performed in some cases.  If you do not define this macro when it
     is required, the compiler will run out of spill registers and
     print a fatal error message.  For most machines, you should not
     define this macro.*/

#define SMALL_REGISTER_CLASSES

/* The letters I, J, K, L, M, N, O and P in a register constraint string
   can be used to stand for particular ranges of immediate operands.
   This macro defines what the ranges are.
   C is the letter, and VALUE is a constant value.
   Return 1 if VALUE is in the range specified by C. */

#define CONST_OK_FOR_LETTER_P(VALUE, C)   \
  ((C) == 'I' ? ((VALUE) >= -0x800000  && (VALUE) < 0x1000000) :\
   (C) == 'J' ? ((VALUE) >= 0          && (VALUE) < 0x1000000) :\
   (C) == 'K' ? ((VALUE) >= -0x800000  && (VALUE) < 0x800000) :\
   (C) == 'L' ? ((VALUE) >= -128       && (VALUE) < 128) :\
   (C) == 'M' ? ((VALUE) >= 0          && (VALUE) < 256) :\
   (C) == 'N' ? ((VALUE) >= 0          && (VALUE) <= 32) : 0)

/* Similar, but for floating constants, and defining letters G and H.
   Here VALUE is the CONST_DOUBLE rtx itself. */

#define CONST_DOUBLE_OK_FOR_LETTER_P(VALUE, C) 0

/* Similar, but used to segregate specific types of operands,
   usually memory references. Letters permitted: Q,R,S,T,U. */
/* For Thor we define:
     Q - A PC or stack relative memory address. */

#define EXTRA_CONSTRAINT(OP, C)                    \
  ((C) == 'Q' ?                                                \
     (REG_P (OP) && REGNO (OP) > LAST_VIRTUAL_REGISTER\
      || GET_CODE (OP) == MEM && PC_OR_STACK_RELATIVE_P (XEXP (OP, 0)))\
   : 0)

/* Given an rtx X being reloaded into a reg required to be
   in class CLASS, return the class of reg to actually use.
   In general this is just CLASS; but on some machines
   in some cases it is preferable to use a more restrictive class.  */

#define PREFERRED_RELOAD_CLASS(X,CLASS)  (CLASS)

/* Return the maximum number of consecutive registers
   needed to represent mode MODE in a register of class CLASS.  */

#define CLASS_MAX_NREGS(CLASS, MODE)\
 ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)

/* These assume that REGNO is a hard or pseudo reg number.
   They give nonzero only if REGNO is a hard reg of the suitable class
   or a pseudo reg currently allocated to a suitable hard reg.
   Since they use reg_renumber, they are safe only once reg_renumber
   has been allocated, which happens in local-alloc.c.  */

#define REGNO_OK_FOR_BASE_P(regno)\
((regno) ==  STACK_POINTER_REGNUM ||\
   (regno) == FRAME_POINTER_REGNUM  ||\
   (regno) == TOP_REGNUM)

#define REGNO_OK_FOR_INDEX_P(regno)  ((regno) == TOP_REGNUM)

/*****************************************************************/
/** STACK AND CALLING:** Frame Layout     **/
/*****************************************************************/
/*
```

A.2 thor.h, continued

```
  Example of stack layout. View of the stack when the function prologue
  (only containing a 'MTOS -3' instruction) has been executed for
  the program:

    int fun (int a, int b)
    {
      int c,d;
    }

                  |                    |     stack/frame grows downwards
                  +--------------------+   |
                  |  caller's TOP-reg  |   |
                  +--------------------+  \|/
                  |        b           |   '
          +--------------------+
     AP + 2 ->|        a           | <- first parameter
          +--------------------+
    AP = FP ->|   return address   | <- EFL
                  +--------------------+
          |        c           |\
          +--------------------+ | frame size = 2
                  |        d           |/
          +--------------------+
            SP -> |   TOP - register   |
          +--------------------+

  AP = Argument Pointer. Same register number as FP.
  FP = Frame Pointer. Defined as a hard register but doesn't really exist.
  SP = Stack Pointer. The only register that actually can be used.

  EFL = End of First Local. This is used as a reference when calculating
        STARTING_FRAME_OFFSET.
*/

/* Define this if pushing a word on the stack
   makes the stack pointer a smaller address.  */

#define STACK_GROWS_DOWNWARD

/* Define this if the nominal address of the stack frame
   is at the high-address end of the local variables;
   that is, each additional local variable allocated
   goes at a more negative offset in the frame.  */

#define FRAME_GROWS_DOWNWARD

/* Offset within stack frame to start allocating local variables at.
   If FRAME_GROWS_DOWNWARD, this is the offset to the END of the
   first local allocated.  Otherwise, it is the offset to the BEGINNING
   of the first local allocated.  */
/* The offset from the FP to EFL. Look at the example above to understand
   this. */

#define STARTING_FRAME_OFFSET 0

/* Offset of first parameter from the argument pointer register value.  */

#define FIRST_PARM_OFFSET(FNDECL) 1

/******************************************************************/
/** STACK AND CALLING:** Frame Registers**/
/******************************************************************/

/* Register to use for pushing function arguments.  */

#define STACK_POINTER_REGNUM 2
```

```
/* Base register for access to local variables of the function.  */

#define FRAME_POINTER_REGNUM 3

/* Base register for access to arguments of the function.  */

#define ARG_POINTER_REGNUM FRAME_POINTER_REGNUM

/******************************************************************/
/** STACK AND CALLING:** Elimination      **/
/******************************************************************/

/* Value should be nonzero if functions must have frame pointers.
   Zero means the frame pointer need not be set up (and parms
   may be accessed via the stack pointer) in functions that seem suitable.
   This is computed in `reload', in reload1.c.  */

#define FRAME_POINTER_REQUIRED 0

/* A C statement to store in the variable DEPTH-VAR the difference
   between the frame pointer and the stack pointer values immediately
   after the function prologue.  The value would be computed from
   information such as the result of `get_frame_size ()' and the
   tables of registers `regs_ever_live' and `call_used_regs'. */
/* This offset is given by the frame size plus 1 extra for the
   TOP - register. */

#define INITIAL_FRAME_POINTER_OFFSET(DEPTH) \
  (DEPTH) = get_frame_size() + 1;

/******************************************************************/
/** STACK AND CALLING:** Stack Arguments**/
/******************************************************************/

/*   A C expression that is the number of bytes actually pushed onto the
     stack when an instruction attempts to push NPUSHED bytes. */

#define PUSH_ROUNDING(BYTES) (BYTES)

/*   A C expression that should indicate the number of bytes of its own
     arguments that a function pops on returning, or 0 if the function
     pops no arguments and the caller must therefore pop them all after
     the function returns. */

/* On Thor we return SIZE when we have a fixed number of arguments or
   when we have a LIBCALL. If the function have a variable number of
   arguments we return 0. */

#define RETURN_POPS_ARGS(FUNDECL,FUNTYPE,SIZE)\
  ((TREE_CODE (FUNTYPE) == IDENTIFIER_NODE\
    || (TYPE_ARG_TYPES (FUNTYPE) == 0\
          || (TREE_VALUE (tree_last (TYPE_ARG_TYPES (FUNTYPE)))\
              == void_type_node)))       \
   ? (SIZE) : 0)

/******************************************************************/
/** STACK AND CALLING:** Register Arguments**/
/******************************************************************/

/* 1 if N is a possible register number for function argument passing.
   This is always false for Thor, since there are no registers */

#define FUNCTION_ARG_REGNO_P(N) 0

/*   A C type for declaring a variable that is used as the first
```

A.2 thor.h, continued

```
     argument of `FUNCTION_ARG' and other related values.  For some
     target machines, the type `int' suffices and can hold the number
     of bytes of argument so far.

     There is no need to record in `CUMULATIVE_ARGS' anything about the
     arguments that have been passed on the stack.  The compiler has
     other variables to keep track of that.  For target machines on
     which all arguments are passed on the stack, there is no need to
     store anything in `CUMULATIVE_ARGS'; however, the data structure
     must exist and should not be empty, so use `int'. */

#define CUMULATIVE_ARGS int

/*   A C statement (sans semicolon) for initializing the variable CUM
     for the state at the beginning of the argument list.  The variable
     has type `CUMULATIVE_ARGS'.  The value of FNTYPE is the tree node
     for the data type of the function which will receive the args, or 0
     if the args are to a compiler support library function.

     When processing a call to a compiler support library function,
     LIBNAME identifies which one.  It is a `symbol_ref' rtx which
     contains the name of the function, as a string.  LIBNAME is 0 when
     an ordinary C function call is being processed.  Thus, each time
     this macro is called, either LIBNAME or FNTYPE is nonzero, but
     never both of them at once. */

#define INIT_CUMULATIVE_ARGS(CUM,FNTYPE,LIBNAME) ((CUM) = 0)

/*   A C statement (sans semicolon) to update the summarizer variable
     CUM to advance past an argument in the argument list.  The values
     MODE, TYPE and NAMED describe that argument.  Once this is done,
     the variable CUM is suitable for analyzing the *following*
     argument with `FUNCTION_ARG', etc.

     This macro need not do anything if the argument in question was
     passed on the stack.  The compiler knows how to track the amount
     of stack space used for arguments without any special help. */

#define FUNCTION_ARG_ADVANCE(CUM, MODE, TYPE, NAMED)

/* Define where to put the arguments to a function.
   Value is zero to push the argument on the stack,
   or a hard register in which to store the argument.

   MODE is the argument's machine mode.
   TYPE is the data type of the argument (as a tree).
    This is null for libcalls where that information may
    not be available.
   CUM is a variable of type CUMULATIVE_ARGS which gives info about
    the preceding args and about the function being called.
   NAMED is nonzero if this argument is a named parameter
    (otherwise it is an extra parameter matching an ellipsis).
    On Thor all arguments are passed on the stack.*/

#define FUNCTION_ARG(CUM, MODE, TYPE, NAMED) 0

/*****************************************************************/
/** STACK AND CALLING:** Scalar Return**/
/*****************************************************************/

/* Define how to find the value returned by a function.
   VALTYPE is the data type of the value (as a tree).
   If the precise function being called is known, FUNC is its FUNCTION_DECL;
   otherwise, FUNC is 0.  */

#define FUNCTION_VALUE(VALTYPE, FUNC)  \
  gen_rtx (REG, TYPE_MODE (VALTYPE), TOP_REGNUM)

/* Define how to find the value returned by a library function
   assuming the value has mode MODE.  */

#define LIBCALL_VALUE(MODE)  gen_rtx (REG, MODE, TOP_REGNUM)

/* 1 if N is a possible register number for a function value.*/

#define FUNCTION_VALUE_REGNO_P(N) ((N) == TOP_REGNUM)

/*****************************************************************/
/** STACK AND CALLING:** Aggregate Return**/
/*****************************************************************/

/* For Thor we make the caller responsible for reserving place
   for the returning aggregate type. We pass the address to the
   reserved memory area in register 0 (i.e. stack pointer) */

#define DEFAULT_PCC_STRUCT_RETURN 1

/*   If the structure value address is not passed in a register, define
     `STRUCT_VALUE' as an expression returning an RTX for the place
     where the address is passed.  If it returns 0, the address is
     passed as an "invisible" first argument. */

#define STRUCT_VALUE 0

/*****************************************************************/
/** STACK AND CALLING:** Caller Saves     **/
/*****************************************************************/

/*****************************************************************/
/** STACK AND CALLING:** Function Entry**/
/*****************************************************************/

/* A C compound statement that outputs the assembler code for entry
   to a function.  The prologue is responsible for setting up the
   stack frame, initializing the frame pointer register, saving
   registers that must be saved, and allocating SIZE additional bytes
   of storage for the local variables.  SIZE is an integer.  FILE is
   a stdio stream to which the assembler code should be output. */
/* On Thor we allocate 1 additional word for the TOP register.
   We also allocate memory for the thor_top_offset_at array. */

#define FUNCTION_PROLOGUE(FILE, SIZE)     \
{                                                                          \
  int num_labels = max_label_num () - get_first_label_num ();\
  int f;                                                           \
                                                                           \
  if (frame_pointer_needed)                       \
    fprintf(FILE, "; Warning: frame pointer not eliminated.\n");\
  fprintf (FILE, "\tMTOS -%d\t\t; local vars(%d) + TOP reg(1)\n",\
           (SIZE) + 1, (SIZE));                   \
                                                                           \
  thor_top_offset = 0;                                    \
  thor_top_offset_at = (int *) xmalloc (num_labels * sizeof (int));\
  for (f = 0 ; f < num_labels ; f++)       \
    thor_top_offset_at[f] = 0;                    \
}

/* EXIT_IGNORE_STACK should be nonzero if, when returning from a function,
   the stack pointer does not matter.  The value is tested only in
   functions that have frame pointers.
   No definition is equivalent to always zero. */
/* On Thor we always need to keep track of the stack-pointer */
```

A.2 thor.h, continued

```
#define EXIT_IGNORE_STACK 0

/*   A C compound statement that outputs the assembler code for exit
     from a function.  The epilogue is responsible for restoring the
     saved registers and stack pointer to their values when the
     function was called, and returning control to the caller.  This
     macro takes the same arguments as the macro `FUNCTION_PROLOGUE',
     and the registers to restore are determined from `regs_ever_live'
     and `CALL_USED_REGISTERS' in the same way. */
/* On Thor we need to copy the TOP register to the caller's TOP
   register. We also free thor_top_offset_at and check the value
   of thor_top_offset. */

#define FUNCTION_EPILOGUE(FILE, SIZE)           \
{ fprintf(FILE, "\tRET %d\n", (SIZE) + 1);      \
  fprintf(FILE, "\tPOP %d\t\t; copy TOP-reg to caller's TOP-reg\n",      \
          (SIZE) + 2 + current_function_pops_args);      \
  fprintf(FILE, "\tMTOS %d\t\t; local vars(%d) + 1 + pop_args(%d)\n",      \
          (SIZE) + 1 + current_function_pops_args, SIZE,      \
          current_function_pops_args);          \
  if (thor_top_offset != 0)                           \
    {                                                                    \
      fprintf(FILE, "; Error: thor_top_offset not 0.\n");      \
      fprintf(stderr,"Error: thor_top_offset not 0 in function epilogue.\n"); \
    }                                                                    \
  fprintf(FILE, "\n");                                        \
                                                                         \
  free (thor_top_offset_at);                          \
}

/*****************************************************************/
/** STACK AND CALLING:** Profiling        **/
/*****************************************************************/
/*   A C statement or compound statement to output to FILE some
     assembler code to call the profiling subroutine `mcount'.  Before
     calling, the assembler code must load the address of a counter
     variable into a register where `mcount' expects to find the
     address.  The name of this variable is `LP' followed by the number
     LABELNO, so you would generate the name using `LP%d' in a
     `fprintf'.

     The details of how the address should be passed to `mcount' are
     determined by your operating system environment, not by GNU CC.  To
     figure them out, compile a small program for profiling using the
     system's installed C compiler and look at the assembler code that
     results. */
/*   Must be defined. */

#define FUNCTION_PROFILER(FILE, LABELNO)

/*****************************************************************/
/** VARARGS                                                    **/
/*****************************************************************/

/*****************************************************************/
/** TRAMPOLINES                                                **/
/*****************************************************************/

/* Output assembler code for a block containing the constant parts
   of a trampoline, leaving space for the variable parts. */
/* Must be defined. */

#define TRAMPOLINE_TEMPLATE(FILE)

/* Length in units of the trampoline for entering a nested function.  */
#define TRAMPOLINE_SIZE 0

/* Emit RTL insns to initialize the variable parts of a trampoline.
   FNADDR is an RTX for the address of the function's pure code.
   CXT is an RTX for the static chain value for the function.  */

/* We copy the register-mask from the function's pure code
   to the start of the trampoline.
/* Must be defined. */

#define INITIALIZE_TRAMPOLINE(TRAMP, FNADDR, CXT)

/*****************************************************************/
/** LIBRARY CALLS                                     **/
/*****************************************************************/

#define INIT_TARGET_OPTABS                   \
  init_libfuncs (umod_optab, QImode, QImode, "umod", '3');\
  init_libfuncs (udiv_optab, QImode, QImode, "udiv", '3');

/*****************************************************************/
/** ADDRESSING MODES                                  **/
/*****************************************************************/

/* Maximum number of registers that can appear in a valid memory address.  */

#define MAX_REGS_PER_ADDRESS 1

/* A C expression that is 1 if the RTX X is a constant which is a
   valid address.  On most machines, this can be defined as
   `CONSTANT_P (X)', but a few machines are more restrictive in which
   constant addresses are supported. */

#define CONSTANT_ADDRESS_P(X)\
  (GET_CODE (X) == LABEL_REF\
   || GET_CODE (X) == SYMBOL_REF\
   || GET_CODE (X) == CONST)

/* Nonzero if the constant value X is a legitimate general operand.
   It is given that X satisfies CONSTANT_P or is a CONST_DOUBLE.  */

#define LEGITIMATE_CONSTANT_P(X)\
  (GET_CODE (X) == LABEL_REF\
   || GET_CODE (X) == SYMBOL_REF\
   || GET_CODE (X) == CONST_INT\
   || GET_CODE (X) == CONST\
   || GET_CODE (X) == CONST_DOUBLE)

/* The macros REG_OK_FOR..._P assume that the arg is a REG rtx
   and check its validity for a certain class.
   We have two alternate definitions for each of them.
   The usual definition accepts all pseudo regs; the other rejects
   them unless they have been allocated suitable hard regs.
   The symbol REG_OK_STRICT causes the latter definition to be used.

   Most source files want to accept pseudo regs in the hope that
   they will get allocated to the class that the insn wants them to be in.
   Source files for reload pass need to be strict.
   After reload, it makes no difference, since pseudo regs have
   been eliminated by then.  */

#ifndef REG_OK_STRICT

/* Nonzero if X is a hard reg that can be used as an base
   or if it is a pseudo reg.  */
```

A.2 thor.h, continued

```
#define REG_OK_FOR_BASE_P(X)  \
   (REGNO_OK_FOR_BASE_P (REGNO (X)) || REGNO (X) >= FIRST_PSEUDO_REGISTER)

/* Nonzero if X is a hard reg that can be used as a index reg
   or if it is a pseudo reg.  */

#define REG_OK_FOR_INDEX_P(X) \
   (REGNO_OK_FOR_INDEX_P (REGNO (X)) || REGNO (X) >= FIRST_PSEUDO_REGISTER)

#else

/* Nonzero if X is a hard reg that can be used as a base reg.  */

#define REG_OK_FOR_BASE_P(X)  REGNO_OK_FOR_BASE_P  (REGNO (X))

/* Nonzero if X is a hard reg that can be used as an index.  */

#define REG_OK_FOR_INDEX_P(X) REGNO_OK_FOR_INDEX_P (REGNO (X))

#endif

/* GO_IF_LEGITIMATE_ADDRESS recognizes an RTL expression
   that is a valid memory address for an instruction.
   The MODE argument is the machine mode for the MEM expression
   that wants to use this address. */

/* Stack relative addresses also allows virtual registers
   (virtual_stack_vars_rtx), so it will recognize the stack/frame
   pointer before register allocation. */

#define STACK_RELATIVE_P(X)                        \
  ((GET_CODE (X) == PLUS                           \
    && REG_P (XEXP (X, 0))                         \
    && (REGNO (XEXP (X, 0)) == STACK_POINTER_REGNUM\
          || REGNO (XEXP (X, 0)) == FRAME_POINTER_REGNUM\
          || (REGNO (XEXP (X, 0)) >= FIRST_PSEUDO_REGISTER\
              && REGNO (XEXP (X, 0)) <= LAST_VIRTUAL_REGISTER))\
    && GET_CODE (XEXP (X, 1)) == CONST_INT)\
   || (REG_P (X) && (REGNO (X) == STACK_POINTER_REGNUM\
                       || REGNO (X) == FRAME_POINTER_REGNUM\
                       || (REGNO (X) >= FIRST_PSEUDO_REGISTER\
                           && REGNO (X) <= LAST_VIRTUAL_REGISTER))))

#define PC_OR_STACK_RELATIVE_P(X)\
  (CONSTANT_ADDRESS_P (X) || STACK_RELATIVE_P (X))

/* We must permit TOP relative addresses (TOP as a base register).
   It's not possible to generate RTL with only one base register
   (the TOS - stack pointer).

   Our TOP-register resides on the stack. So we must be able to
   handle stack indirect addresses. Symbol_refs is handled in the
   same way as stack references. We can therefore permit all kind
   of MEM (PC or stack relative) indirect addresses and also
   use them in relative/"base register"-addressing.

   A pseudo register is eventually transformed into the TOP-
   register or a stack-position. So, like the TOP-register,
   all pseudo registers are also equivalent to a MEM-indirect
   address. */

#define MEM_INDIRECT_P(X)                  \
  (GET_CODE (X) == MEM && PC_OR_STACK_RELATIVE_P (XEXP (X, 0))\
   || (REG_P (X)                               \
       && (REGNO (X) == TOP_REGNUM\
             || REGNO (X) > LAST_VIRTUAL_REGISTER)))

#define MEM_INDIRECT_RELATIVE_P(X)\
  (GET_CODE (X) == PLUS            \
   && MEM_INDIRECT_P (XEXP (X, 0))\
   && GET_CODE (XEXP (X, 1)) == CONST_INT\
   && INTVAL (XEXP (X, 1)) >= 0\
   || MEM_INDIRECT_P (X))

#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR)\
{ if (PC_OR_STACK_RELATIVE_P (X))\
    goto ADDR;                          \
  if (MEM_INDIRECT_RELATIVE_P (X))\
    goto ADDR;                          \
}

/* Try machine-dependent ways of modifying an illegitimate address
   to be legitimate.  If we find one, return the new, valid address.
   This macro is used in only one place: `memory_address' in explow.c.

   OLDX is the address as it was before break_out_memory_refs was called.
   In some cases it is useful to look at this to decide what needs to be done.

   MODE and WIN are passed so that this macro can use
   GO_IF_LEGITIMATE_ADDRESS.

   It is always safe for this macro to do nothing.  It exists to recognize
   opportunities to optimize the output. */

#define LEGITIMIZE_ADDRESS(X,OLDX,MODE,WIN)  {}

/* Go to LABEL if ADDR (a legitimate address expression)
   has an effect that depends on the machine mode it is used for. */

#define GO_IF_MODE_DEPENDENT_ADDRESS(ADDR,LABEL)  { }

/*****************************************************************/
/** CONDITION CODE                                             **/
/*****************************************************************/

/*   A C compound statement to set the components of `cc_status'
     appropriately for an insn INSN whose body is EXP.  It is this
     macro's responsibility to recognize insns that set the condition
     code as a byproduct of other activity as well as those that
     explicitly set `(cc0)'.

     This macro is not used on machines that do not use `cc0'.

     If there are insns that do not set the condition code but do alter
     other machine registers, this macro must check to see whether they
     invalidate the expressions that the condition code is recorded as
     reflecting.  For example, on the 68000, insns that store in address
     registers do not set the condition code, which means that usually
     `NOTICE_UPDATE_CC' can leave `cc_status' unaltered for such insns.
     But suppose that the previous insn set the condition code based
     on location `a4@(102)' and the current insn stores a new value in
     `a4'.  Although the condition code is not changed by this, it will
     no longer be true that it reflects the contents of `a4@(102)'.
     Therefore, `NOTICE_UPDATE_CC' must alter `cc_status' in this case
     to say that nothing is known about the condition code value.

     The definition of `NOTICE_UPDATE_CC' must be prepared to deal with
     the results of peephole optimization: insns whose patterns are
     `parallel' RTXs containing various `reg', `mem' or constants which
     are just the operands.  The RTL structure of these insns is not
     sufficient to indicate what the insns actually do.  What
```

A.2 thor.h, continued

```
     `NOTICE_UPDATE_CC' should do when it sees one is just to run
     `CC_STATUS_INIT'.

     A possible definition of `NOTICE_UPDATE_CC' is to call a function
     that looks at an attribute (*note Insn Attributes::.) named, for
     example, `cc'.  This avoids having detailed information about
     patterns in two places, the `md' file and in `NOTICE_UPDATE_CC'.  */

#define NOTICE_UPDATE_CC(EXP, INSN)       \
      notice_update_cc (EXP, INSN);

/*****************************************************************/
/** COSTS                                                      **/
/*****************************************************************/

/* A part of a C `switch' statement that describes the relative costs
   of constant RTL expressions.  It must contain `case' labels for
   expression codes `const_int', `const', `symbol_ref', `label_ref'
   and `const_double'.  Each case must ultimately reach a `return'
   statement to return the relative cost of the use of that kind of
   constant value in an expression.  The cost may depend on the
   precise value of the constant, which is available for examination
   in X, and the rtx code of the expression in which it is contained,
   found in OUTER_CODE. */
/* A cost of 3 or greater, is expensive and may force the operand
   into a register. This is not good on Thor, so only use costs < 3.

   Cost: 0 - CONST_INT that fits in a halfword instruction
         1 - CONST_INT that fits in a word instruction
         2 - CONST_INT that is too large to fit in a word, or other
             things that need to be put in a DATA directive
             (the constant pool). */

#define CONST_COSTS(X,CODE,OUTER_CODE)\
  case CONST_INT:                     \
    if (CONST_OK_FOR_LETTER_P (INTVAL (X), 'L')\
        || CONST_OK_FOR_LETTER_P (INTVAL (X), 'M'))\
      return 0;                               \
    if (CONST_OK_FOR_LETTER_P (INTVAL (X), 'I'))\
      return 1;                               \
    return 2;                                 \
  case CONST:                                 \
  case LABEL_REF:                     \
  case SYMBOL_REF:                    \
  case CONST_DOUBLE:                  \
    return 2;

/* Like `CONST_COSTS' but applies to nonconstant RTL expressions.
   This can be used, for example, to indicate how costly a multiply
   instruction is. In writing this macro, you can use the construct
   `COSTS_N_INSNS (N)' to specify a cost equal to N fast
   instructions. OUTER_CODE is the code of the expression in which X
   is contained. */

#define RTX_COSTS(X,CODE,OUTER_CODE)\
  case MEM:                                   \
    return (simple_operand ((X), VOIDmode) ? 1 : 13);\
    break;                                    \
  case MULT:                                  \
    if (FLOAT_MODE_P (GET_MODE (X)))\
      return COSTS_N_INSNS (3);   \
    else                                      \
      return COSTS_N_INSNS (4);   \
    break;                                    \
  case DIV:                                   \
    return COSTS_N_INSNS (15);    \
    break;                                       \
  case MOD:                                      \
    return COSTS_N_INSNS (20);    \
    break;                                       \
  case UDIV:                                     \
    return COSTS_N_INSNS (30);    \
    break;                                       \
  case UMOD:                                     \
    return COSTS_N_INSNS (45);    \
    break;

/* An expression giving the cost of an addressing mode that contains
   ADDRESS.  If not defined, the cost is computed from the ADDRESS
   expression and the `CONST_COSTS' values.

   For most CISC machines, the default cost is a good approximation
   of the true cost of the addressing mode.  However, on RISC
   machines, all instructions normally have the same length and
   execution time.  Hence all addresses will have equal costs.

   In cases where more than one form of an address is known, the form
   with the lowest cost will be used.  If multiple forms have the
   same, lowest, cost, the one that is the most complex will be used. */

#define ADDRESS_COST(ADDRESS)\
  (PC_OR_STACK_RELATIVE_P (ADDRESS) ? 1 : 13)

/* We want to inhibit moves between memory and register */

#define MEMORY_MOVE_COST(M) ((M) == QImode ? 0 : 2)

/* Nonzero if access to memory by bytes is slow and undesirable.  */

#define SLOW_BYTE_ACCESS 0

/* Define this macro if it is as good or better to call a constant
   function address than to call an address kept in a register. */
/* We don't want any calls via a register. */

#define NO_FUNCTION_CSE
#define NO_RECURSIVE_FUNCTION_CSE

/*****************************************************************/
/** SECTIONS                                                   **/
/*****************************************************************/

/* Output before read-only data.  */

#define TEXT_SECTION_ASM_OP "code\tSECT 1,R,C"

/* Output before writable data.  */

#define DATA_SECTION_ASM_OP "data\tSECT 2,W,C"

/*****************************************************************/
/** PIC                                                          **/
/*****************************************************************/

/*****************************************************************/
/** ASSEMBLER FORMAT:** File Framework**/
/*****************************************************************/

/* Output at beginning of assembler file.  */

#define ASM_FILE_START(FILE)\
   fprintf (FILE, "; Assembler file from %s\n",\
```

A.2 thor.h, continued

```
            VERSION_INFO);

/* Output to assembler file text saying following lines
   may contain character constants, extra white space, comments, etc.  */

#define ASM_APP_ON "; Start of user assembler.\n"

/* Output to assembler file text saying following lines
   no longer contain unusual constructs.  */

#define ASM_APP_OFF "; End of user assembler\n"

#define ASM_IDENTIFY_GCC(FILE)            \
   fprintf(FILE,"; Generated by thor-gcc. \n");

#define ASM_COMMENT_START ";"

/****************************************************************/
/** ASSEMBLER FORMAT:** Data Output       **/
/****************************************************************/

/* A C statement to output to the stdio stream STREAM an assembler
   instruction to assemble a floating-point constant of `QFmode',
   whose value is VALUE. VALUE will be a C expression
   of type `REAL_VALUE_TYPE'. */

#define ASM_OUTPUT_BYTE_FLOAT(STREAM, VALUE)\
{ char dstr[30];                  \
  REAL_VALUE_TO_DECIMAL (VALUE, "%.10e", dstr);\
  fprintf (STREAM, "\tDATAF %s\n", dstr);\
}

/* A C statement to output to the stdio stream STREAM an assembler
   instruction to assemble an integer of 1 bytes, whose value is X.
   The argument VALUE will be an RTL expression which represents a
   constant value. */
/* On Thor, special treatment is needed for symbol_refs. */

#define ASM_OUTPUT_CHAR(FILE,X)\
  output_char (FILE, X);

/* This is how to output an assembler line for a numeric constant byte.  */

#define ASM_OUTPUT_BYTE(FILE,VALUE)\
{                                                  \
  fprintf (FILE, "\tDATA ");\
  output_hex (FILE, VALUE);\
}

/* A C statement to output to the stdio stream STREAM an assembler
   instruction to assemble a string constant containing the LEN bytes
   at PTR. */
/* For Thor we need to output the characters one by one. One character
   should be placed in one word (32 bits). Non-ascii is output as
   integers. */

#define ASM_OUTPUT_ASCII(STREAM, PTR, LEN)\
  output_ascii (STREAM, PTR, LEN);

/* Define the parentheses used to group arithmetic operations
   in assembler code.  */

#define ASM_OPEN_PAREN "("
#define ASM_CLOSE_PAREN ")"

/****************************************************************/
```

```
/** ASSEMBLER FORMAT:** Uninitialized Data**/
/****************************************************************/

/* This says how to output an assembler line
   to define a global common symbol.  */

#define ASM_OUTPUT_COMMON(FILE, NAME, SIZE, ROUNDED)\
{                                                          \
  if (in_text_section())          \
    data_section();                       \
                                                           \
  ASM_GLOBALIZE_LABEL((FILE),(NAME));\
  assemble_name ((FILE), (NAME));\
  fprintf ((FILE), "\tREP 0,%u\n", (ROUNDED));\
}

/* This says how to output an assembler line
   to define a local common symbol.  */

#define ASM_OUTPUT_LOCAL(FILE, NAME, SIZE, ROUNDED)\
{                                                          \
  if (in_text_section())          \
    data_section();                       \
                                                           \
  assemble_name ((FILE), (NAME));\
  fprintf ((FILE), "\tREP 0,%u\n", (ROUNDED));\
}

/****************************************************************/
/** ASSEMBLER FORMAT:** Label Output      **/
/****************************************************************/

/* This is how to output the definition of a user-level label named NAME,
   such as the label on a static function or variable NAME.  */

#define ASM_OUTPUT_LABEL(FILE,NAME)\
  do { assemble_name (FILE, NAME); fputs (":\n", FILE); } while (0)

/* This is how to output a command to make the user-level label named NAME
   defined for reference from other files.  */

#define ASM_GLOBALIZE_LABEL(FILE,NAME)\
  do { assemble_name (FILE, NAME); fputs ("\tXDEF\n", FILE);} while (0)

/* We need this macro to ensure the linker that a variable is declared
   externaly */
#define ASM_OUTPUT_EXTERNAL(FILE,DECL,NAME)\
  do { assemble_name (FILE, NAME); fputs ("\tXREF\n", FILE);} while (0)

/* This is how to output a reference to a user-level label named NAME.  */

#define ASM_OUTPUT_LABELREF(FILE,NAME)\
  fprintf (FILE, "_%s", NAME)

/* This is how to output an internal numbered label where
   PREFIX is the class of label and NUM is the number within the class.  */
/* On Thor we also set thor_top_offset to the initial value for the new
   basic block. */

#define ASM_OUTPUT_INTERNAL_LABEL(FILE,PREFIX,NUM)\
{                                                                      \
  fprintf (FILE, "%s%d:", PREFIX, NUM);   \
                                                                       \
  /* set thor_top_offset to the new basic blocks initial value: */\
  if (! strcmp (PREFIX, "L"))                    \
    set_new_thor_top_offset (FILE, NUM);\
```

A.2 thor.h, continued

```
                                                                           \
  fputs ("\n", FILE);                                      \
}

/* This is how to store into the string LABEL
   the symbol_ref name of an internal numbered label where
   PREFIX is the class of label and NUM is the number within the class.
   This is suitable for output with `assemble_name'.  */

#define ASM_GENERATE_INTERNAL_LABEL(LABEL,PREFIX,NUM)\
  sprintf (LABEL, "*%s%d", PREFIX, NUM)

/* Store in OUTPUT a string (made with alloca) containing
   an assembler-name for a local static variable named NAME.
   LABELNO is an integer which is different for each call.  */

#define ASM_FORMAT_PRIVATE_NAME(OUTPUT, NAME, LABELNO)\
( (OUTPUT) = (char *) alloca (strlen ((NAME)) + 10),\
  sprintf ((OUTPUT), "%s.%d", (NAME), (LABELNO)))

/******************************************************************/
/** ASSEMBLER FORMAT:** Initialization**/
/******************************************************************/

/******************************************************************/
/** ASSEMBLER FORMAT:** Macros for Initialization**/
/******************************************************************/

/* If defined, `main' will not call `__main'. We don't actually have
   any init section in the Thor assembler. The define is made for
   the sole purpose of getting rid of the unnessecary '__main' call
   (only needed for C++). */

#define HAS_INIT_SECTION

/******************************************************************/
/** ASSEMBLER FORMAT:** Instruction Output**/
/******************************************************************/

/*   A C initializer containing the assembler's names for the machine
     registers, each one as a C string constant.  This is what
     translates register numbers in the compiler into assembler
     language.  */

#define REGISTER_NAMES \
{"TOP", "TMP", "TOS", "FP"}

/* This is how to output an insn to push/pop a register on the stack.
   It need not be very fast code.  */
/* On Thor we don't need these push and pop insns. They are only
   used for certain regs that we don't use. */
/* #define ASM_OUTPUT_REG_PUSH(FILE,REGNO) */
/* #define ASM_OUTPUT_REG_POP(FILE,REGNO)  */

#define PRINT_OPERAND_PUNCT_VALID_P(CHAR) \
  ((CHAR) == '#')

/* A C compound statement to output to stdio stream STREAM the
   assembler syntax for an instruction operand X.  X is an RTL
   expression.

   If the specification was just `%DIGIT' then CODE is 0;
   if the specification was `%LTR DIGIT' then CODE is the ASCII code
   for LTR. LTR can be any letter but a,c,l,n.
 */
#define PRINT_OPERAND(FILE, X, CODE)\
  print_operand (FILE, X, CODE);

/* Print a memory operand whose address is X, on file FILE. */

#define PRINT_OPERAND_ADDRESS(FILE, X)    \
  thor_fatal ("PRINT_OPERAND_ADDRESS!\tDo not use %%a directive.");

/******************************************************************/
/** ASSEMBLER FORMAT:** Dispatch Tables**/
/******************************************************************/

/* This is how to output an element of a case-vector that is absolute. */
/* On Thor all case table-elements are absolute. */

#define ASM_OUTPUT_ADDR_VEC_ELT(FILE, VALUE)  \
  fprintf (FILE, "\tDATA L%d/2\n", VALUE)

/* This is how to output an element of a case-vector that is relative.  */

#define ASM_OUTPUT_ADDR_DIFF_ELT(FILE, VALUE, REL)  \
  fprintf (FILE, "\tDATA (L%d-L%d)/2\n", VALUE, REL)

/* Define this if the label before a jump-table needs to be output
   specially. */
/* On Thor we want to output the address of the table as the
   first element (it's needed in the casesi-pattern). We do
   nothing special with the label. */

#define ASM_OUTPUT_CASE_LABEL(STREAM, PREFIX, NUM, TABLE)\
{                                                                  \
  char buf[256];                                 \
                                                                   \
  ASM_OUTPUT_INTERNAL_LABEL (STREAM, PREFIX, NUM);\
                                                                   \
  ASM_GENERATE_INTERNAL_LABEL (buf, PREFIX, NUM);\
  fputs ("\tDATA ", STREAM);              \
  assemble_name (STREAM, buf);            \
  fputs ("/4\n", STREAM);                        \
}

/******************************************************************/
/** ASSEMBLER FORMAT:** Alignment Output**/
/******************************************************************/

/* This is how to output an assembler line
   that says to advance the location counter
   to a multiple of 2**LOG bytes.  */

#define ASM_OUTPUT_ALIGN(FILE,LOG)  \
  fprintf (FILE, "\tDATA 0\t; align %d\n", (LOG))

/* This is how to output an assembler line
   that says to advance the location counter by SIZE bytes.  */

#define ASM_OUTPUT_SKIP(FILE,SIZE)  \
  fprintf (FILE, "\tREP 0,%u\t\t; skip\n", (SIZE))

/******************************************************************/
/** DEBUGGING INFO:** All debuggers**/
/******************************************************************/

/******************************************************************/
/** CROSS-COMPILATION                             **/
/******************************************************************/
```

A.2 thor.h, continued

```
/*****************************************************************/
/** MISC.                                                      **/
/*****************************************************************/

/*   An alias for a machine mode name.  This is the machine mode that
     elements of a jump-table should have. */
#define CASE_VECTOR_MODE QImode

/* Define this if the case instruction expects the table
   to contain offsets from the address of the table.
   Do not define this if the table should contain absolute addresses.  */
                                  /* AWAY COMMENTED 950629 */
/*#define CASE_VECTOR_PC_RELATIVE*/

/* Define this if the case instruction drops through after the table
   when the index is out of range.  Don't define it if the case insn
   jumps to the default label instead.  */
                                  /* AWAY COMMENTED 950629 */
/*#define CASE_DROPS_THROUGH*/

/* Operations with different modes occurs in a whole register */
#define WORD_REGISTER_OPERATIONS

/* Specify the tree operation to be used to convert reals to integers.  */
#define IMPLICIT_FIX_EXPR FIX_ROUND_EXPR

/* This is the kind of divide that is easiest to do in the general case.  */
#define EASY_DIV_EXPR TRUNC_DIV_EXPR

/* This flag, if defined, says the same insns that convert to a signed fixnum
   also convert validly to an unsigned one.  */
#define FIXUNS_TRUNC_LIKE_FIX_TRUNC

/* Max number of bytes we can move from memory to memory
   in one reasonably fast instruction.  */
#define MOVE_MAX 1

/* Value is 1 if truncating an integer of INPREC bits to OUTPREC bits
   is done just by pretending it is already truncated.  */
#define TRULY_NOOP_TRUNCATION(OUTPREC, INPREC) 1

/* Specify the machine mode that pointers have.
   After generation of rtl, the compiler makes no further distinction
   between pointers and any other objects of this machine mode.  */
#define Pmode QImode

/* A function address in a call instruction
   is a byte address (for indexing purposes)
   so give the MEM rtx a byte's mode.  */
#define FUNCTION_MODE QImode
```

A.3 thor.c

```
/*
 *           thor.c
 *
 *
 * Purpose
 *           Help functions used by patterns and macros in 'thor.md'- and
 *           'thor.h'-file.
 *
 * Operations
 *  TMP_REG_RTX                   Macro to check if 'mode' is 'QImode'
 *  thor_fatal                    Fatal error handling function
 *  tmp_register_operand          Predicate to check if argument is 'TMP'
 *  top_register_operand          Predicate to check if argument is 'TOP'
 *  tos_register_operand          Predicate to check if argument is 'TOS'
 *  pop_operand                   Check if argument is valid for popping
 *  simple_operand                Check the simplicity of the argument
 *  complex_operand               Check the complexity of the argument
 *  stack_relative_operand        True if argument is stack relative
 *  indirect_jump_operand         True if argument is a indirect jump operand
 *  tmp_or_push_operand           True if argument is 'TMP' or pushable
 *  next_cc0_user_unsigned_jump_p Returns former user of CC
 *  thor_init_expanders           Initializing values in function entry
 *  emit_push                     Emits a push pattern
 *  emit_pop                      Emits a pop pattern
 *  complex_rating                Gives a value of argument's complexity
 *  check_swap_operands           Check if operands could be swapped
 *  emit_move_sequence            Moves data from one operand to the other
 *  emit_add_sequence             Performs a addition
 *  emit_sub_sequence             Performs a subtraction
 *  emit_commutative_sequence     Performs a commutative operation
 *  emit_non_commutative_sequence Performs a noncommutative operation
 *  emit_ashl_sequence            Performs arithmetic shift right
 *  dead_p                        True if operand dies
 *  output_complex_address_fetch  Fetch a data by complex addressing
 *  output_complex_push           Push data by complex addressing
 *  output_complex_pop            Pop data by complex addressing
 *  output_complex_pop_with_push  Perform a simple push in complex pop
 *  output_condjump               Perform a conditional jump
 *  output_indirect_call          Perform a indirect call to a function
 *  output_casesi                 Implementing the C switch construct
 *  notice_update_cc              Updating the CC register
 *  output_hex                    Output constants hex-style
 *  output_ascii                  Output ascii characters
 *  set_new_thor_top_offset       Manipulating global variable
 *  output_char                   Output characters
 *  print_operand                 When arbitary operands are to be output
 *  thor_output_addr_const        Outputting constant addresses
 *
 * Usage
 *           Called by patterns and macros in the 'thor.md'- and
 *           the 'thor.h'-file.
 *
 *
 *
 * I/O       None
 *
 *
 *
 * Machine   (No dependencies)
 *
 *
 * Compiler  (No dependencies)
 *
 *
 * Author    Harry Gunnarsson, Thomas Lundqvist / 951115
 *
 * Revised       by          date    remarks
 *
 *   1.0.0       HG, TL      951115  New version
 *
 */
/* Subroutines for assembler code output on Thor. */

#include <stdio.h>
#include "config.h"
#include "tree.h"
#include "rtl.h"
#include "insn-attr.h"
#include "insn-config.h"
#include "recog.h"
#include "conditions.h"
#include "regs.h"

extern int reload_in_progress;
extern int reload_completed;
extern short *reg_renumber;

static void thor_output_addr_const ();
static int complex_rating ();

/* When pushing (normal argument pushes) something on the stack, the
   TOP-register will be hidden inside the stack. This is the offset
   (stack-slots) from the new top of stack. */

int thor_top_offset = 0;

/* The thor_top_offset will normally be 0 in the beginning of each
   basic block. But sometimes, in the cross-jump optimization,
   some normal pushes can be merged and the initial offset for a
   basic block will no longer be 0.

   To keep track of this initial offset, we need to register the value
   of thor_top_offset at each label in the function. It works in about
   the same way as in the reload-pass (reload1.c) when eliminating the
   frame pointer. There is a big difference though. We only make one
   sweep over the instructions when the assembler code is output. So
   we can't update any initial offsets before the current insn (ie, we
   can't change any offsets in the code that we already have output),
   only after.  But that is enough to solve the cross-jump problem.

   The thor_top_offset_at is an array allocated in the
   FUNCTION_PROLOUGE and freed in FUNCTION_EPILOUGE, indexed by a
   label number. */

int *thor_top_offset_at;

/* When a compare instruction is output, it pushes new data on the stack.
   The conditional jump that follows need to throw this data away.
   This flag tells the condjump if a MTOS 1 needs to be output. */

int thor_compare_need_stack_adjust = 0;

/* This variable is used internally in a peephole to hold
   information about death notes. */

int thor_peep_pop_operand_dies = 0;

/* This variable contains a pointer to the operand used in
   the latest emit_pop call (set by emit_pop). It is used in
   emit_commutative_sequence (via the function complex_rating),
   to determine which operand should be placed first. It is
   better to emit instructions using the same operand
```

A.3 thor.c, continued

```
   together so that the peephole optimization can find
   and remove temporary store/load instructions. */

static rtx thor_last_pop_operand = 0;

/* Shared rtx's for the TMP-register, to be used by all RTL-generating
   expands. A shared TMP-register RTX saves memory and makes
   jump-optimization work better. Without it, the function
   'delete_computation' in jump.c will fail to remove a 'push_tmp'
   instruction when removing the corresponding 'pop_tmp' instruction.
   (Initialized in 'thor_init_expanders'.) */

rtx tmp_reg_QI_rtx = 0;
rtx tmp_reg_QF_rtx = 0;

#define TMP_REG_RTX(mode) \
  ((mode) == QImode ? tmp_reg_QI_rtx : tmp_reg_QF_rtx)

/* Our own error-abort function: */

void
thor_fatal (msg)
     char *msg;
{
  extern char *input_filename;

  fprintf (stderr, "%s: Internal THOR-GCC abort.\n%s\n", input_filename, msg);
  exit (FATAL_EXIT_CODE);
}

/*******************************************************************/
/** Functions used as predicates in match_operand expressions**/
/** in the machine description:           **/
/*******************************************************************/

/* Return 1 if OP is a TMP register reference of mode MODE. */

int
tmp_register_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return (GET_CODE (op) == REG
          && REGNO (op) == TMP_REGNUM
          && GET_MODE (op) == mode);
}

/* Return 1 if OP is a TOP register reference of mode MODE. */

int
top_register_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return (GET_CODE (op) == REG
          && REGNO (op) == TOP_REGNUM
          && (mode == VOIDmode
              || mode == GET_MODE (op)));
}

/* Return 1 if OP is a TOS (stack pointer) register reference
   of mode MODE. */

int
tos_register_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return (GET_CODE (op) == REG
          && REGNO (op) == STACK_POINTER_REGNUM
          && GET_MODE (op) == mode);
}

/* Return 1 if OP is a valid operand that stands for popping a
   value of mode MODE onto the stack. */

int
pop_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  if (GET_CODE (op) != MEM)
    return 0;

  if (GET_MODE (op) != mode)
    return 0;

  op = XEXP (op, 0);

  if (GET_CODE (op) != POST_INC)
    return 0;

  return XEXP (op, 0) == stack_pointer_rtx;
}

/* Return 1 if OP is a simple operand, ie, an immediate operand,
   a stack/pc relative address or the TOP-register. */

int
simple_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return (complex_rating (op, 1) < 30);
}

/* Return 1 if OP is a complex operand, ie, an indirect operand. */

int
complex_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return (complex_rating (op, 1) >= 30);
}

/* Return 1 if OP is a stack relative operand. */

int
stack_relative_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return (complex_rating (op, 1) / 10 == 1
          && ! (GET_CODE (op) == MEM && CONSTANT_ADDRESS_P (XEXP (op, 0))));
}

/* Return 1 if OP is an indirect jump operand, ie, a stack relative
   operand or a MEM (MEM (const/pc relative)) operand. */

int
indirect_jump_operand (op, mode)
```

A.3 thor.c, continued

```
     rtx op;
     enum machine_mode mode;
{
  return (stack_relative_operand (op, mode)
            || (GET_CODE (op) == MEM
                && GET_CODE (XEXP (op, 0)) == MEM
                && CONSTANT_ADDRESS_P (XEXP (XEXP (op, 0), 0))));
}

/* Return 1 if OP is an tmp register operand or a normal push
   operand. */

int
tmp_or_push_operand (op, mode)
     rtx op;
     enum machine_mode mode;
{
  return (tmp_register_operand (op, mode)
            || push_operand (op, mode));
}

/* Return true if the next cc0 user (a conditional jump instruction)
   is unsigned, otherwise false. */

int
next_cc0_user_unsigned_jump_p (insn)
     rtx insn;
{
  enum rtx_code code;

  if ( !(insn = next_cc0_user (insn)))
    thor_fatal ("No next_cc0_user in 'next_cc0_user_unsigned_jump_p'!");

  if (GET_CODE (insn) == JUMP_INSN
      && GET_CODE (PATTERN (insn)) == SET
      && GET_CODE (SET_SRC (PATTERN (insn))) == IF_THEN_ELSE)
    code = GET_CODE (XEXP (SET_SRC (PATTERN (insn)), 0));
  else
    thor_fatal ("jump insn expected in 'next_cc0_user_unsigned_jump_p'!");

  return (code == LTU || code == GTU || code == LEU || code == GEU);
}

/*******************************************************************/
/** Functions used as subroutines in the machine**/
/** descriptions, that emit rtl:  **/
/*******************************************************************/

/* Initialize things that need to be initialized once every new
   function. Used in the INIT_EXPANDERS macro. */
/* We need to initialize the rtx's for the TMP-register, to be
   shared by all RTL-generating expands. */

void
thor_init_expanders ()
{
  tmp_reg_QI_rtx = gen_rtx (REG, QImode, TMP_REGNUM);
  tmp_reg_QF_rtx = gen_rtx (REG, QFmode, TMP_REGNUM);
}

/* Emit a push instruction that pushes 'operand' on stack. */

void
emit_push (operand, mode)
     rtx operand;
     enum machine_mode mode;
{
  emit_insn (gen_rtx (SET, VOIDmode,
                      TMP_REG_RTX (mode),
                      operand));
}

/* Emit a pop instruction that pops 'operand' from stack. */

void
emit_pop (operand, mode)
     rtx operand;
     enum machine_mode mode;
{
  emit_insn (gen_rtx (SET, VOIDmode,
                      operand,
                      TMP_REG_RTX (mode)));
  emit_insn (gen_rtx (CLOBBER, VOIDmode,
                      TMP_REG_RTX (mode)));

  /* set thor_last_pop_operand for complex_rating: */
  thor_last_pop_operand = operand;
}

/* Help function used in check_swap_operands and in simple_operand.
   Examines operand x and returns a relative rating, A lower value
   for simple operands and a higher value for complex ones.

   Rating   Operand

      0     an immediate operand if 'immediate_simple' is true.

     10     a register/pseudo operand or a stack relative operand.

     15     a pseudo register operand
            if it was used earlier as an operand in emit_pop.

     20     an immediate operand if 'immediate_simple' is false.

     30     other operands, indirect addressing.

     40     a virtual register, like the virtual_stack_reg.

   The higher rating if the operand was used in the latest emit_pop,
   is because of the peephole optimization. If the operand was
   used in the previous instruction we want to make this operand
   come first in a commutative operation. If it succeeds we would
   get two instructions using the same operand adjacent to each
   other, and if the operand is temporary, a peephole will
   remove the two instructions.

   'immediate_simple' is 1 if immediate operands is extra simple
   (rating 0), 0 if an immediate instruction don't exist (rating 20). */

static int
complex_rating (x, immediate_simple)
     rtx x;
     int immediate_simple;
{
  if (immediate_operand (x, VOIDmode))
    return (immediate_simple ? 0 : 20);

  if (REG_P (x) && REGNO (x) > LAST_VIRTUAL_REGISTER
      && thor_last_pop_operand != NULL
      && REG_P (thor_last_pop_operand)
      && REGNO (thor_last_pop_operand) == REGNO (x))
    return 15;
```

A.3 thor.c, continued

```
  if (REG_P (x) && (REGNO (x) == TOP_REGNUM
                    || REGNO (x) == TMP_REGNUM
                    || REGNO (x) > LAST_VIRTUAL_REGISTER)
      || GET_CODE (x) == MEM && PC_OR_STACK_RELATIVE_P (XEXP (x, 0)))
    return 10;

  if (REG_P (x) && REGNO (x) <= LAST_VIRTUAL_REGISTER)
    return 40;

  return 30;
}

/* The most complex operand of 'op1' and 'op2' is moved to 'op1' and
   the most simple one into 'op2'. For all commutative operators
   it's better to push a complex 'op1' and to do a simple
   operation with 'op2'.

   Simple operands are all pseudo registers (not including virtual
   ones), the TOP-register or a PC/stack-relative address.

   Extra complex is an virtual register (stack pointer etc). If such
   an operand ends up in 'op2' the frame pointer elimination will
   produce incorrect code.

   if 'immediate_simple' is 1, an immediate operand is treated as
   an extra simple operand, otherwise it is treated as a complex
   operand.

   Return 1 if 'op1' and 'op2' is swapped, otherwise 0. */

int
check_swap_operands (op1, op2, immediate_simple)
     rtx *op1;
     rtx *op2;
     int immediate_simple;
{
  rtx tmp;

  if (complex_rating (*op2, immediate_simple)
      > complex_rating (*op1, immediate_simple))
    {
      tmp  = *op1;
      *op1 = *op2;
      *op2 = tmp;

      return 1;
    }

  return 0;
}

/* Emit insns to move operands[1] into operands[0].

   Return 1 if we have written out everything that needs to be done to
   do the insn.  Otherwise, return 0 and the caller will emit the insn
   normally. */

int
emit_move_sequence (operands, mode)
     rtx *operands;
     enum machine_mode mode;
{
  register rtx operand0 = operands[0];
  register rtx operand1 = operands[1];

  if (! push_operand (operand0, mode))
    {
      /* Emit a move by first pushing to the TMP-register
         and then popping from the TMP-register. */

      emit_push (operand1, mode);

      /* If the TOP register is used explicitly, we are generating
            a store into the function value register (TOP). We then have
            to emit a clobber TOP in between so that the jump2-pass
            don't mess with our pop-insn. */

      if (top_register_operand (operand0, mode))
          emit_insn (gen_rtx (CLOBBER, VOIDmode,
                              gen_rtx (REG, mode, TOP_REGNUM)));

      emit_pop (operand0, mode);
      return 1;
    }
  else
    { /* Normal push (PRE_DEC) */
      emit_insn (gen_rtx (SET, VOIDmode, operand0, operand1));
      emit_insn (gen_rtx (CLOBBER, VOIDmode,
                          TMP_REG_RTX (mode)));
      return 1;
    }

/*  return 0; */
}

/* Emit insns to add operands[1] and operands[2] into operands[0].

   Return 1 if we have written out everything that needs to be done to
   do the insn.  Otherwise, return 0 and the caller will emit the insn
   normally.  */

int
emit_add_sequence (operands, mode)
     rtx *operands;
     enum machine_mode mode;
{
  rtx operand0 = operands[0];
  rtx operand1 = operands[1];
  rtx operand2 = operands[2];

  if (! tos_register_operand (operand0, QImode))
    {
      check_swap_operands (&operand1, &operand2, 1);

      /* If check_swap_operands didn't succeed in making operand2
         simple, we must reload it via a pseudo register. */
      if (! simple_operand (operand2, mode))
          operand2 = force_reg (mode, operand2);

      emit_push (operand1, mode);

      emit_insn (gen_rtx (SET, VOIDmode,
                          TMP_REG_RTX (mode),
                          gen_rtx (PLUS, mode,
                                   TMP_REG_RTX (mode),
                                   operand2)));

      emit_pop (operand0, mode);

      return 1;
    }
```

A.3 thor.c, continued

```
  return 0;  /* There is a special insn 'MTOS' handling the tos-case. */
}

/* Emit insns to subtract operands[1] with operands[2] into operands[0].

   Return 1 if we have written out everything that needs to be done to
   do the insn.  Otherwise, return 0 and the caller will emit the insn
   normally.  */

int
emit_sub_sequence (operands, mode)
     rtx *operands;
     enum machine_mode mode;
{
  rtx operand0 = operands[0];
  rtx operand1 = operands[1];
  rtx operand2 = operands[2];
  int swapped;

  swapped = check_swap_operands (&operand1, &operand2, 0);

  /* If check_swap_operands didn't succeed in making operand2
     simple, we must reload it via a pseudo register. */
  if (! simple_operand (operand2, mode))
    operand2 = force_reg (mode, operand2);

  emit_push (operand1, mode);

  if (swapped)
    emit_insn (gen_rtx (SET, VOIDmode,
                        TMP_REG_RTX (mode),
                        gen_rtx (MINUS, mode,
                                 operand2,
                                 TMP_REG_RTX (mode))));
  else
    emit_insn (gen_rtx (SET, VOIDmode,
                        TMP_REG_RTX (mode),
                        gen_rtx (MINUS, mode,
                                 TMP_REG_RTX (mode),
                                 operand2)));

  emit_pop (operand0, mode);

  return 1;
}

/* Emit insns for a normal commutative operator. 'code' can be one
   of MULT, AND, IOR, XOR.

   Return 1 if we have written out everything that needs to be done to
   do the insn.  Otherwise, return 0 and the caller will emit the insn
   normally.  */

int
emit_commutative_sequence (operands, mode, code)
     rtx *operands;
     enum machine_mode mode;
     enum rtx_code code;
{
  rtx operand0 = operands[0];
  rtx operand1 = operands[1];
  rtx operand2 = operands[2];

  check_swap_operands (&operand1, &operand2, 1);

  /* If check_swap_operands didn't succeed in making operand2
     simple, we must reload it via a pseudo register. */
  if (! simple_operand (operand2, mode))
    operand2 = force_reg (mode, operand2);

  emit_push (operand1, mode);

  emit_insn (gen_rtx (SET, VOIDmode,
                      TMP_REG_RTX (mode),
                      gen_rtx (code, mode,
                               TMP_REG_RTX (mode),
                               operand2)));

  emit_pop (operand0, mode);

  return 1;
}

/* Emit insns for a normal non commutative insn. 'code' can be one of
   DIV, MOD.

   Return 1 if we have written out everything that needs to be done to
   do the insn.  Otherwise, return 0 and the caller will emit the insn
   normally.  */

int
emit_non_commutative_sequence (operands, mode, code)
     rtx *operands;
     enum machine_mode mode;
     enum rtx_code code;
{
  rtx operand0 = operands[0];
  rtx operand1 = operands[1];
  rtx operand2 = operands[2];

  /* If operand2 is complex, we must reload it via a pseudo register. */
  if (! simple_operand (operand2, mode))
    operand2 = force_reg (mode, operand2);

  emit_push (operand1, mode);

  emit_insn (gen_rtx (SET, VOIDmode,
                      TMP_REG_RTX (mode),
                      gen_rtx (code, mode,
                               TMP_REG_RTX (mode),
                               operand2)));

  emit_pop (operand0, mode);

  return 1;
}

/* Emit insns for ashl.

   Return 1 if we have written out everything that needs to be done to
   do the insn.  Otherwise, return 0 and the caller will emit the insn
   normally.  */

int
emit_ashl_sequence (operands)
     rtx *operands;
{
  register rtx operand0 = operands[0];
  register rtx operand1 = operands[1];
  register rtx operand2 = operands[2];
  enum machine_mode mode = QImode;
```

A.3 thor.c, continued

```
  emit_push (operand1, mode);

  emit_insn (gen_rtx (SET, VOIDmode,
                      TMP_REG_RTX (mode),
                      gen_rtx (ASHIFT, mode,
                               TMP_REG_RTX (mode),
                               operand2)));

  emit_pop (operand0, mode);

  return 1;

/*  return 0; */
}

/*******************************************************************/
/** Functions used as subroutines in the machine**/
/** descriptions, in peep-hole:           **/
/*******************************************************************/

/* Return nonzero if OPERAND dies in INSN, ie, if there is a
   REG_DEAD-note matching OPERAND. OPERAND can be either a register
   operand or a memory-address operand (a register swapped out to
   the stack). if PRE_DEC is 1, all stack-relative addresses in the
   dead-notes will be decremented by 1 before comparing. */

int
dead_p (insn, operand, pre_dec)
     rtx insn;
     rtx operand;
     int pre_dec;
{
  register rtx link;

  for (link = REG_NOTES (insn); link; link = XEXP (link, 1))
    if (REG_NOTE_KIND (link) == REG_DEAD)
      if (! (pre_dec && GET_CODE (operand) == MEM
               && STACK_RELATIVE_P (XEXP (operand, 0))))
          {
            if (rtx_equal_p (operand, XEXP (link, 0)))
              return 1;
          }
      else
          {
            rtx x = XEXP (link, 0);

            if (GET_CODE (x) == MEM
                && STACK_RELATIVE_P (XEXP (x, 0)))
              {
                HOST_WIDE_INT off1, off2;

                off1 = (REG_P (XEXP (operand, 0)) ? 0
                          : INTVAL (XEXP (XEXP (operand, 0), 1)));
                off2 = (REG_P (XEXP (x, 0)) ? 0
                          : INTVAL (XEXP (XEXP (x, 0), 1)));
                if (off1 == off2 - 1)
                  return 1;
              }
          }
  return 0;
}

/*******************************************************************/
/** Functions used as subroutines in the machine**/
/** descriptions, that emit assembler-insns:**/
```

```
/*******************************************************************/

/* Help function used in output_complex_push/pop.
   'popflag' is 1 if an extra offset of +1 is needed when
   outputting an operand, otherwise 0. */

static rtx
output_complex_address_fetch (operand, popflag)
     rtx operand;
     int popflag;
{
  rtx x;
  rtx address;
  rtx offset;

  if (GET_CODE (operand) != MEM
      || ! MEM_INDIRECT_RELATIVE_P (XEXP (operand, 0)))
    fatal_insn ("Non MEM operand in output_complex_address_fetch:", operand);

  x = XEXP (operand, 0);
  if (GET_CODE (x) == PLUS)
    {
      address = XEXP (x, 0);
      offset  = XEXP (x, 1);
    }
  else
    {
      address = x;
      offset  = const0_rtx;
    }

  output_asm_insn (popflag ? "PSH %z0" : "PSH %0", &address);
  output_asm_insn ("MTOS 1", 0);

  return offset;
}

/* Output instructions to make an indirect push.

   All valid memory addresses not satisfying the 'Q'-constraint
   must be handled here, ie, all memory-indirect relative. */

void
output_complex_push (operand)
     rtx operand;
{
  rtx offset;

  offset = output_complex_address_fetch (operand, 0);
  output_asm_insn ("NOP", 0);
  output_asm_insn ("PSHX %0", &offset);
}

/* Output instructions to make an indirect pop. */

void
output_complex_pop (operand)
     rtx operand;
{
  rtx offset;

  offset = output_complex_address_fetch (operand, 1);
  output_asm_insn ("NOP", 0);
  output_asm_insn ("POPX %0", &offset);
}
```

A.3 thor.c, continued

```
/* Output instruction to make an simple push inside
   an indirect pop. Used in the peep-hole optimizations.

     operands[0] - not used
     operands[1] - the operand of the simple push
     operands[2] - the operand of the indirect pop */

void
output_complex_pop_with_push (operands, mode)
     rtx *operands;
     enum machine_mode mode;
{
  rtx offset;

  output_asm_insn ("\t\t; PEEP - push inside popx", 0);

  /* Call output_compex_address_fetch with popflag = 0,
     to make stack relative addressing correct (we don't
     have a PSH in front of the first PSH in the POPX).
     Normally, a NOP instruction follows, but here we
     output a PSH instruction instead. */

  offset = output_complex_address_fetch (operands[2], 0);

  if (immediate_operand (operands[1], mode))
    output_asm_insn ("PSHI %1", operands);
  else
    output_asm_insn ("PSH %1", operands);

  output_asm_insn ("POPX %0", &offset);
}

/* Output instructions for a conditional jump of type 'code'. Used in
   define_insn "*condjump" and in some peepholes.

   'reversed' is 1 if the jump condition is to be reversed. */

void
output_condjump (label_ref, operator, reversed)
     rtx label_ref;
     rtx operator;
     int reversed;
{
  int index = CODE_LABEL_NUMBER (label_ref) - get_first_label_num ();
  enum rtx_code cond = GET_CODE (operator);

  /* update the initial thor_top_offset for the basic block with this label: */
  thor_top_offset_at[index] = thor_top_offset;

  if (reversed)
    cond = reverse_condition (cond);

  switch (cond)
    {
    case EQ:
      output_asm_insn ("JREQ %0", &label_ref);
      break;
    case NE:
      output_asm_insn ("JRNE %0", &label_ref);
      break;
    case GT:
    case GTU:
      output_asm_insn ("JRGT %0", &label_ref);
      break;
    case LT:
    case LTU:
      output_asm_insn ("JRLT %0", &label_ref);
      break;
    case GE:
    case GEU:
      output_asm_insn ("JRGE %0", &label_ref);
      break;
    case LE:
    case LEU:
      output_asm_insn ("JRLE %0", &label_ref);
      break;
    default:
      thor_fatal ("wrong type of code in output_condjump");
      break;
    }
}

/* Output instructions to make an indirect call.  The PC register will
   be pushed to emulate the behaviour of a call instruction. */

void
output_indirect_call (address)
     rtx address;
{
  rtx offset = CONST0_RTX (QImode);

  output_asm_insn ("PSHR PC", &address);

  if (GET_CODE (address) == MEM
      && PC_OR_STACK_RELATIVE_P (XEXP (address, 0)))
    output_asm_insn ("PSH %0", &address);
  else
    fatal_insn ("unexpected address in output_indirect_call:", address);

  output_asm_insn ("SL 1", 0);
  output_asm_insn ("POPR PC", 0);
  output_asm_insn ("ADDI 2", 0);
  output_asm_insn ("NOP", 0);
}

/* Output instructions to make an 'casesi', except the starting
   push instruction:

   operand 0 is index
   operand 1 is the minimum bound
   operand 2 is the maximum bound - minimum bound
   operand 3 is CODE_LABEL for the table;
   operand 4 is the CODE_LABEL to go to if index out of range.

   The casesi instruction is really a caseqi instruction
   (changed to QImode in stmt.c).
 */

void
output_casesi (operands)
     rtx *operands;
{
  /* Subtract the lower bound: */
  if (GET_CODE (operands[1]) != CONST_INT)
    thor_fatal ("Illegal operand 1 in output_casesi!");
  else if (INTVAL (operands[1]) != 0)
    {
      operands[5] = gen_rtx (CONST_INT, VOIDmode, -(INTVAL (operands[1])));
      output_asm_insn ("ADDI %5", operands);
    }

  /* Compare with the upper limit. The CONST_INT value
```

A.3 thor.c, continued

```
     is put in the constant-pool by the md-constraint. */
  if (! (GET_CODE (operands[2]) == MEM
         && CONSTANT_ADDRESS_P (XEXP (operands[2], 0))))
    thor_fatal ("Illegal operand 2 in output_casesi!");

  output_asm_insn ("CMPU %2", operands);
  output_asm_insn ("PSHI 0", operands);
  output_asm_insn ("CMP 0", operands);
  output_asm_insn ("MTOS 1", operands);

  /* Jump if not OK to index out of range label. */
  output_asm_insn ("JRGT %4", operands);

  /* Add the address of the table (the first DATA-word in the
     table contains the address of the table) */
  output_asm_insn ("ADD %3", operands);

  output_asm_insn ("MTOS 1", operands);
  output_asm_insn ("NOP", operands);

  /* Jump indirect via the table: (address in stack-top 2 insns
     prior to this one). An offset of 1 is given to the JRX
     to skip the first DATA-word (the address of the table). */
  output_asm_insn ("JRX 1", operands);
  output_asm_insn ("NOP", operands);
  output_asm_insn ("NOP", operands);
}

/*****************************************************************/
/** Misc.                                                      **/
/*****************************************************************/

void
notice_update_cc (exp, insn)
     rtx exp;
     rtx insn;
{
  enum attr_cc cc = get_attr_cc (insn);
  rtx dest = SET_DEST (exp);
  rtx src = SET_SRC (exp);
  switch (cc)
    {
    case CC_CLOBBER:
      CC_STATUS_INIT;
      cc_status.flags |= CC_NO_OVERFLOW;
      break;

    case CC_UNCHANGED:
      break;

    case CC_SET1:
      CC_STATUS_INIT;
      cc_status.flags |= CC_NO_OVERFLOW;
      cc_status.value1 = recog_operand[0];
      break;

    case CC_SET2:
      CC_STATUS_INIT;
      cc_status.flags |= CC_NO_OVERFLOW;
      cc_status.value1 = recog_operand[0];
      cc_status.value2 = recog_operand[1];
      break;

    case CC_COPY:
      /* If the tmp-reg is copied to a new operand and value1
           contains the tmp-reg, then set value1 to the new
           operand. The new operand might be used in a succeding
           test instruction, but the tmp-reg operand will probably
           never be refered to again. */
      if (cc_status.value1
            && REG_P (cc_status.value1)
            && REGNO (cc_status.value1) == TMP_REGNUM
            && REG_P (src)
            && REGNO (src) == TMP_REGNUM)
          {
            cc_status.value1 = dest;
          }
      break;

    case CC_COMPARE:
      CC_STATUS_INIT;
      cc_status.flags |= CC_NO_OVERFLOW;
      cc_status.value1 = src;
      break;

    default:
      fatal_insn ("Error in notice_update_cc", insn);
    }
}

/* print an operand in hex-format: */

void
output_hex (file, x)
     FILE *file;
     rtx x;
{
  HOST_WIDE_INT n = INTVAL (x);

  if (n >= 0)
    fprintf (file, "16#%04x_%04x#", (n >> 16) & 0x7fff, n & 0xffff);
  else if (n != -0x80000000)
    fprintf (file, "-16#%04x_%04x#", ((-n) >> 16) & 0x7fff, (-n) & 0xffff);
  else
    fprintf (file, "-16#7fff_ffff#-1");
}

/* Output a string. Used by ASM_OUTPUT_ASCII. A typical example of
   the output looks like this (each DATAS row will have a maximum of
   DATAS_MAX_CHARS chars):

           DATAS "H","e","l","l","o"," ","W","o","r","l","d"
             DATA 0
*/

#define DATAS_MAX_CHARS 12

void
output_ascii (file, ptr, len)
     FILE *file;
     char *ptr;
     int len;
{
  int i;
  int size_of_datas = 0;

  for (i=0 ; i < len ; i++)
    {
      if (ptr[i] >= 32 && ptr[i] < 127)
          {
            if (size_of_datas == 0)
              fprintf (file, "\tDATAS \"%c\"", ptr[i]);
```

A.3 thor.c, continued

```
            else
              fprintf (file, ",\"%c\"", ptr[i]);
            size_of_datas++;
          }
      else
          {
            if (size_of_datas > 0)
              fprintf (file, "\n");
            fprintf (file, "\tDATA %d\n", (int)(ptr[i]));
            size_of_datas = 0;
          }

      if (size_of_datas >= DATAS_MAX_CHARS)
          {
            fprintf (file, "\n");
            size_of_datas = 0;
          }
    }
}

/* Set thor_top_offset to the new basic blocks initial value.  This
   function is called when a normal jump instruction is output. The
   initial offsets of each basic block is stored in
   thor_top_offset_at. This array is updated when a jump instruction
   (conditionally or unconditionally jump) is output. The updates
   only work in the forward direction. But even if we use an entry in
   thor_top_offset_at before it has been updated, everything should
   work out correctly (the new value will 0).  */

void
set_new_thor_top_offset (file, label_num)
     FILE *file;
     int label_num;
{
  int newvalue = 0;
  int index;

  index = label_num - get_first_label_num ();

  if (label_num < max_label_num ()
      && index >= 0)
    newvalue = thor_top_offset_at[index];

  if (newvalue != thor_top_offset)
    fprintf (file, "\t\t\t; thor_top_offset new value = %d", newvalue);
  thor_top_offset = newvalue;
}

/* Called by the macro ASM_OUTPUT_CHAR i the .h file.
   The function outputs an instruction to assemble an integer of 1 byte,
   whose value is x.

   A const_int that lies outside the 'I'-interval is output as
   a hexadecimal constant.

   A special treatment is needed for symbol_refs. The assembler always
   treats these references as 8-bit byte pointers. But our pointers
   are 32-bit long. Therefore, we need to divide all these references
   by 4. All symbol_refs are aligned to 32-bit word boundaries, so
   there should be no loss of information/rounding. */

void
output_char (file, x)
     FILE *file;
     rtx x;
{
  fputs ("\tDATA ", file);
  if (GET_CODE (x) == CONST_INT
      && ! CONST_OK_FOR_LETTER_P (INTVAL (x), 'I'))
    output_hex (file, x);
  else if (GET_CODE (x) == SYMBOL_REF)
    { /* All symbol_refs must by divided by 4: */
      output_addr_const (file, x);
      fputs ("/4", file);
    }
  else if (GET_CODE (x) == CONST
             && GET_CODE (XEXP (x, 0)) == PLUS
             && GET_CODE (XEXP (XEXP (x, 0), 0)) == SYMBOL_REF
             && GET_CODE (XEXP (XEXP (x, 0), 1)) == CONST_INT)
    { /* Symbol_ref plus constant, only divide the symbol_ref
           not the const_int: */
      rtx symref = XEXP (XEXP (x, 0), 0);
      rtx constint  = XEXP (XEXP (x, 0), 1);

      output_addr_const (file, symref);
      fprintf (file, "/4");
      if (INTVAL (constint) >= 0)
          fprintf (file, "+");
      output_addr_const (file, constint);
    }
  else
    output_addr_const (file, x);

  fputs ("\n", file);
}

/* Called by the macro PRINT_OPERAND in the .h file.
   The function outputs an operand 'x' in assembler format.

   If the specification was just `%DIGIT' then 'code' is 0;
   if the specification was `%LTR DIGIT' then 'code' is the ASCII code
   for LTR. LTR can be any letter but a,c,l,n.

   On Thor we define the following special letters:

     h - print a CONST_INT in hexadecimal: 16#xxxx_xxxx#.
     z - add an offset of 1 to all stack-relative addresses.
     y - add an offset of 2 to all stack-relative addresses.
 */

void
print_operand (file, x, code)
     FILE *file;
     rtx x;
     int code;
{
  if (x == NULL)
    switch (code)
      {
      case '#':
          /* Output a 'nop' if there's nothing for the delay slot.  */
          if (dbr_sequence_length () == 0)
            fputs ("\n\tNOP\n\tNOP", file);
          if (dbr_sequence_length () == 1)
            fputs ("\n\tNOP", file);
          break;
      default:
          fputc(code, file);
          break;
      }
  else if (REG_P (x))
    {
```

A.3 thor.c, continued

```
      if (REGNO (x) == TOP_REGNUM)
           fprintf (file, "%d\t\t; TOP", thor_top_offset
                                        + (code == 'y' ? 2
                                           : (code == 'z' ? 1 : 0)));
      else if (REGNO (x) == TMP_REGNUM)
           fprintf (file, "0\t\t; TMP");
      else
           fprintf (file, "%s", reg_names[REGNO (x)]);
    }
  else if (GET_CODE (x) == MEM)
    {
      int offset = (code == 'y' ? 2
                      : code == 'z' ? 1 : 0);
      rtx addr = XEXP (x, 0);

      if (CONSTANT_ADDRESS_P (addr))
           thor_output_addr_const (file, addr);
      else if (GET_CODE (addr) == PLUS && REG_P (XEXP (addr, 0))
                  && GET_CODE (XEXP (addr, 1)) == CONST_INT
                  && REGNO (XEXP (addr, 0)) == STACK_POINTER_REGNUM)
           fprintf(file, "%d\t\t; S+%d",
                   INTVAL (XEXP (addr, 1)) + offset,
                   INTVAL (XEXP (addr, 1)) - thor_top_offset);
      else if (REG_P (addr) &&
                  REGNO (addr) == STACK_POINTER_REGNUM)
           fprintf(file, "%d\t\t; S+%d", offset, - thor_top_offset);
      else
           fprintf(file,"address");
    }
  else if (GET_CODE (x) == CONST_INT && (code) == 'h')
    output_hex (file, x);
  else
    thor_output_addr_const (file, x);
}

/* Help function used in output_operand. Calls the normal
   output_addr_const except when a symbol_ref with an offset is
   output. The offset added to the symbol_ref must be multiplied by 4
   to convert it from the normal 32-bit word offset to the 8-bit byte
   offset used by the assembler. */

static void
thor_output_addr_const (file, x)
     FILE *file;
     rtx x;
{
  if (GET_CODE (x) == CONST
      && GET_CODE (XEXP (x, 0)) == PLUS
      && GET_CODE (XEXP (XEXP (x, 0), 0)) == SYMBOL_REF
      && GET_CODE (XEXP (XEXP (x, 0), 1)) == CONST_INT)
    {
      rtx symref = XEXP (XEXP (x, 0), 0);
      rtx constint  = XEXP (XEXP (x, 0), 1);

      output_addr_const (file, symref);
      if (INTVAL (constint) >= 0)
           fprintf (file, "+");

      fprintf (file,
#if HOST_BITS_PER_WIDE_INT == HOST_BITS_PER_INT
                 "%d",
#else
                 "%ld",
#endif
                 INTVAL (constint) * 4);
    }
  else
    output_addr_const (file, x);
}
```

A.4 crt0.asm

```
;/*
; *           crt0.asm
; *
; *
; * Purpose
; *           Initialization of PC- TOS- and SR-register.
; *           Will be linked to every executable file.
; *
; * Operations
; *           No functions, just a small assembler extract.
; *
; * Usage
; *           Will be assembled when the compiler is built.
; *           The file will be used and linked with every executable file
; *           produced.
; *
; * I/O       When this file is assembled when building the compiler, an
; *           object-file is produced.
; *
; *
; * Machine   (No dependencies)
; *
; *
; * Compiler  (No dependencies)
; *
; *
; * Author    Harry Gunnarsson, Thomas Lundqvist / 951115
; *
; * Revised        by            date    remarks
; *
; *   1.0.0        HG, TL        951115  New version
; *
; */
; crt0.asm
;
; This file contains the startup code for Thor.
;
; The code's job is to call _main and then halt.
;
          ORG  0
          DATA ___crt0/2  ; Initial value for PC
          DATA 16#1000#   ; Initial value for TOS
          DATA 16#30e#    ; Initial value for SR

_main      XREF

___crt0:
          MTOS -1         ; Make space for TOP-reg (return value of _main)
          CALL _main
          NOP
          NOP
          HLT             ; return value on top
          NOP
          NOP
```

A.5 t-thor

```
#/*
# *           t-thor
# *
# *
# * Purpose
# *           Additional rules to GNU CC's makefile.
# *
# * Operations
# *           No functions, just make rules
# *
# * Usage
# *           Used when building the compiler. Thor's makefile 'Makefile.in'
# *           needs additional information about library functions etc.
# *
# *
# * I/O       None
# *
# *
# *
# * Machine   (No dependencies)
# *
# *
# * Compiler  (No dependencies)
# *
# *
# * Author    Harry Gunnarsson, Thomas Lundqvist / 951115
# *
# * Revised        by            date    remarks
# *
# *   1.0.0        HG, TL        951115  New version
# *
# */

# We need to supply our own assembly version of the libgcc1.c file.

CROSS_LIBGCC1 =
LIBGCC1_TEST =

# The libgcc2.c file is not needed. It's only needed to support the
# longlong data type, C++, profiling or exceptions.

LIBGCC2 =

# Don't use the normal rules when building and installing LIBGCC.

LIBGCC = object/libgcc.obj
INSTALL_LIBGCC = install-libgcc.obj

thordir=$(srcdir)/config/thor

##############################################################
# Rules for building libgcc.
#
object/libgcc.obj:$(thordir)/thor-libgcc1.asm
          cp $(thordir)/thor-libgcc1.asm libgcc1.asm
          -assemble libgcc1.asm
          mv object/libgcc1.obj object/libgcc.obj

install-libgcc.obj:object/libgcc.obj install-object-dir
          -if [ -f object/libgcc.obj ] ; then \
            rm -f $(libsubdir)/object/libgcc.obj; \
            $(INSTALL_DATA) object/libgcc.obj $(libsubdir)/object/libgcc.obj; \
            chmod a-x $(libsubdir)/object/libgcc.obj; \
          else true; fi

##############################################################
# Create extra subdirectory called 'object'.
#
install-object-dir:install-dir
          -if [ -d $(libsubdir)/object ] ; then true ; else \
            mkdir $(libsubdir)/object ; chmod a+rx $(libsubdir)/object ; fi

##############################################################
# Rules for installing the 'as' and 'ld' scripts and building
# the startup file crt0.obj.
#
# 'install-normal' already exist in the Makefile.in. The line
# below only adds an extra dependency.
#
install-normal:install-thor

install-thor:install-asld object/crt0.obj

install-asld:$(thordir)/as $(thordir)/ld install-object-dir
          rm -f $(libsubdir)/as
          rm -f $(libsubdir)/ld
          cp $(thordir)/as $(libsubdir)/as
          cp $(thordir)/ld $(libsubdir)/ld

object/crt0.obj:$(thordir)/crt0.asm install-object-dir
          rm -f crt0.asm
          cp $(thordir)/crt0.asm crt0.asm
          -assemble crt0.asm
          rm -f $(libsubdir)/object/crt0.obj
          cp object/crt0.obj $(libsubdir)/object/crt0.obj
```

A.6 thor-libgcc1.asm

```
;/*
; *           thor-libgcc1.asm
; *
; *
; * Purpose
; *           Provide the compiler with impicit library functions.
; *
; * Operations
; *           ___udivqi3   unsigned 32-bit division
; *           ___umodqi3   unsigned 32-bit modulo
; *
; *
; * Usage
; *           Called impicitely by the compiler when an operation not
; *           supported by the hardware is desired.
; *
; *
; * I/O       None
; *
; *
; *
; * Machine   (No dependencies)
; *
; *
; * Compiler  (No dependencies)
; *
; *
; * Author    Harry Gunnarsson, Thomas Lundqvist / 951115
; *
; * Revised        by            date    remarks
; *
; *   1.0.0        HG, TL        951115  New version
; *
; */

; libgcc1.asm
;
; This file contains subroutines needed by THOR-GCC. Some instructions
; in the output of THOR-GCC lack a corresponding machine instruction.
; In these cases THOR-GCC outputs a call to a function in this file.
;
; These functions are needed:
;
;   ___udivqi3 - unsigned 32-bit division
;   ___umodqi3 - unsigned 32-bit modulo
;

;----------------------------------------------------------------------
; __udivqi3 (unsigned int nominator, unsigned int denominator)
;
; Unsigned division is performed with the help of the normal
; signed DIV instruction.
;
; The sign bit (bit 31) of the nominator and denominator
; determines the action to take. The four different cases are:
;
;    Sign of:  nom den  Action:
;                0   0   perform normal DIV
;                0   1   result is always 0
;                1   0   tricky case, use the following algorithm:
;                           (divide nom with 2, do a normal signed
;                            division and correct result)
;                           nom2    = nom DIV 2
;                           res2    = nom2 DIV den
;                           mod     = nom - (res2 * den) * 2
;                           corr    = mod >= den
;                              result  = res2 * 2 (+ 1 if corr)
;                1   1   result is always 0 if nom < den otherwise 1
;
;
code       SECT 1,R,C
           DATA 0          ; align function
___udivqi3 XDEF
___udivqi3:

; Parameters. When calling, the denominator is pushed first:

frame_startSREL
nom                 SREL
den                 SREL

; Start of program:
           ; Check sign bit on 'nom'
           TEST nom
           JRLT nom1
           NOP
           TEST den

           ; Check sign bit on 'den'
           JRLT nom0_den1
           NOP
           NOP

           ; -------------------------------------------------- nom 0, den 0
           ; nom and den both 0, perform normal div
           JR   end
           PSH  nom
           DIV  den

nom0_den1:
           ORGS frame_start
           ; -------------------------------------------------- nom 0, den 1
           ; result always 0
           JR   end
           PSHI 0
           NOP

nom1:
           ORGS frame_start
           ; Check sign bit on 'den'
           JRLT nom1_den1
           NOP
           NOP

           ; -------------------------------------------------- nom 1, den 0
           ; The tricky case:
           ; nom2 = nom DIV 2
           PSH  nom
           SR   1

           ; res2 = nom2 DIV den
           DIV  den
res2       SREL

           ; mod = nom - (res2 * den) * 2
           PSH  res2
           MUL  den
           SL   1
           SBRU nom
           CLRF 12

           ; corr = mod >= den
```

A.6 thor-libgcc1.asm, continued

```
          CMPU den
          PSHI 0
          CMP  0

          ; result = res2 * 2 (+ 1 if corr)
          JRLT end
          MTOS 2          ; throw away PSHI 0 and mod, revealing res2
          SL   1

          JR   end
          ADDI 1
          NOP

nom1_den1:
          ORGS frame_start
          ; -------------------------------------------------- nom 1, den 1
          ; Return 0 if nom < den, otherwise 1.
          ; An unsigned cmp is called for here, but both nom and den have
          ; the sign bit set so a signed compare will work as well.
          PSH  nom
          CMP  den
          JRLT end; jump if nom < den
          MTOS 1
          PSHI 0
          ADDI 1

end:
          RET 1
          POP 4           ; copy result to callers TOP reg
          MTOS 3


;----------------------------------------------------------------------
; __umodqi3 (unsigned int nominator, unsigned int denominator)
;
; Unsigned modulo (remainder) is done with the help of ___udivqi3.
;
;   mod = nom - (nom DIV den) * den
;
code      SECT 1,R,C
          DATA 0          ; align function
___umodqi3 XDEF
___umodqi3:

; Parameters. When calling, the denominator is pushed first:

frame_startSREL
nom                SREL
den                SREL

; Start of program:
          MTOS -1         ; the TOP register (return value of ___udivqi3)
div       SREL

          PSH  den
          PSH  nom
          CALL ___udivqi3
          NOP
          NOP
          ORGS div
                          ; 'nom DIV den' now at top
          MULU den
          CLRF 12         ; '(nom DIV den) * den'

          SBRU nom
          CLRF 12         ; 'nom - (nom DIV den) * den'
          RET 1
          POP 4           ; copy result (mod) to callers TOP reg
          MTOS 3
```

A.7 as

```
#!/bin/csh -f
#
#/*
# *           as
# *
# *
# * Purpose
# *           To override UNIX assembler 'as' and get Thor's assembler
# *           'assemble'
# *
# * Operations
# *           No functions, just UNIX commands
# *
# * Usage
# *           Used by the compiler when it want to assemble a file
# *
# * I/O
# *           A temporary assembler file is produced with suffix '.asm'
# *
# *
# * Machine   (No dependencies)
# *
# *
# * Compiler  (No dependencies)
# *
# *
# * Author    Harry Gunnarsson, Thomas Lundqvist / 951115
# *
# * Revised        by            date    remarks
# *
# *   1.0.0        HG, TL        951115  New version
# *
# */
#
# This script runs the Thor/Oden assembler, 'assemble'.
#
# GCC calls this script (it must have the name 'as') when assembling
# a program.
#
# Syntax:
#
#   as [normal assemble options] -o objfile filename.s
#
# The last argument is the name of the input file ending with a '.s'
# suffix. This file must be renamed to get a '.asm' suffix, because
# that's the only suffix the assembler can handle.
#
# The object file generated by the assemble command, will always
# have the name filename.obj. We must change it explicitly to
# objfile afterwards.
#
# The assemble options are passed directly to the assemble command.
#

@ NORMAL_ARG   = $#argv - 3
@ O_ARG        = $#argv - 2
@ OBJFILE_ARG  = $#argv - 1
@ FILENAME_ARG = $#argv

if ($O_ARG < 1) then
  echo "Error in as script: wrong number of arguments." ; exit 1
endif

set FILENAME = $argv[$FILENAME_ARG]

if ("$argv[$O_ARG]" != "-o" || $FILENAME:e != "s" ) then
  echo "Error in as script: wrong type of arguments." ; exit 1
endif

set OBJFILE  = $argv[$OBJFILE_ARG]
set NORMALOPTIONS = $argv[1-$NORMAL_ARG]

# First change the 'filename.s' to 'filename.asm'

if (! -e $FILENAME) then
  echo "Error in as script: can't find the file $FILENAME" ; exit 1
endif

if (-e $FILENAME:r.asm || ! { mv $FILENAME $FILENAME:r.asm } ) then
  echo "Error in as script: can't rename $FILENAME to $FILENAME:r.asm" ; exit 1
endif

# Assemble:
assemble -q $NORMALOPTIONS $FILENAME:r.asm

# Restore the filename to the '.s' suffix:
mv $FILENAME:r.asm $FILENAME

# Build the filename for the output of assemble:
set OUTPUT_HEAD  = $FILENAME:h
set OUTPUT_TRAIL = $FILENAME:t
if ("$OUTPUT_HEAD" == "$OUTPUT_TRAIL") then
  set OUTPUT_NAME = object/$OUTPUT_TRAIL:r.obj
else
  set OUTPUT_NAME = $OUTPUT_HEAD/object/$OUTPUT_TRAIL:r.obj
endif

# Build the filename for the 'objfile', also check if destination dir
# exists:
set OBJFILE_HEAD  = $OBJFILE:h
set OBJFILE_TRAIL = $OBJFILE:t
if ("$OBJFILE_HEAD" == "$OBJFILE_TRAIL") then
  set OBJFILE_NAME = object/$OBJFILE_TRAIL
  if (! -d object) mkdir object
else
  set OBJFILE_NAME = $OBJFILE_HEAD/object/$OBJFILE_TRAIL
  if (! -d $OBJFILE_HEAD/object) mkdir $OBJFILE_HEAD/object
endif

# Change the name of the object file to 'objfile'
if ("$OUTPUT_NAME" != "$OBJFILE_NAME") \
  mv $OUTPUT_NAME $OBJFILE_NAME
```

A.8 ld

```
#!/bin/csh -f
#
#/*
# *           ld
# *
# *
# * Purpose
# *           To override UNIX linker 'ld and get Thor's linker 'link'.
# *
# * Operations
# *           No functions, just UNIX commands
# *
# * Usage
# *           Used by the compiler when it want to link '.o'-files into an
# *           executable file.
# *
# *
# * I/O       An executable file is produced
# *
# *
# *
# * Machine   (No dependencies)
# *
# *
# * Compiler  (No dependencies)
# *
# *
# * Author    Harry Gunnarsson, Thomas Lundqvist / 951115
# *
# * Revised        by            date    remarks
# *
# *   1.0.0        HG, TL        951115  New version
# *
# */
#
# This script runs the Thor/Oden linker, 'link'.
#
# GCC calls this script (it must have the name 'ld') when linking
# programs.
#
# Syntax:
#
#   ld [normal link options] [-o outfile] [-L libdir] objfiles
#
# The order of the arguments does not matter. There can be
# several '-L' options if more than one library directory
# are to be included. If no '-o' option is given the output
# file will be named 'a.lnk'.
#
# All arguments not preceeded by '-o' or '-L', or ending with
# '.obj' are considered to be normal link options and are
# passed directly to the link command.
#

# Parse arguments:
#
set NORMAL   =
set OUTFILE  = a.lnk
set OBJFILES =
set LIBDIRS  =

while ($#argv > 0)
  switch ($1)
    case -o:
      shift; set OUTFILE = $1
      breaksw

    case -L:
      shift; set LIBDIRS = "$LIBDIRS -L $1"
      breaksw

    default:
      if ("$1:e" == "obj") then
           set OBJFILES = "$OBJFILES $1"
      else
           set NORMAL = "$NORMAL $1"
      endif
      breaksw
  endsw
  shift
end

# OUTFILE should end with a '.lnk'
if ("$OUTFILE:e" == "") set OUTFILE = ${OUTFILE}.lnk

echo "Normal options:$NORMAL"
echo "Outfile:        $OUTFILE"
echo "Objfiles:      $OBJFILES"
echo "Libdirs:       $LIBDIRS"

link${NORMAL}${LIBDIRS}${OBJFILES}
```

APPENDIX B - Diff files of the changes in GNU CC source

Each of the succeeding sections will show the diff file produced when comparing the changed source files with the original ones. We have followed the convention to save the original source files by appending an extra suffix '`.org`' onto the file name. The command line given when producing the diff files is '`diff -c FILE.c FILE.c.org`'.

B.1 reload.c versus reload.c.org

```
*** reload.cThu Sep 21 13:06:29 1995
--- reload.c.orgThu Jun 15 14:01:03 1995
***************
*** 4353,4377 ****
        return 1;
      }

- #ifdef THOR_RELOAD_CHECK_FOR_CONST
-   /* Check the case when a constant expression has been created
-      but fails to be valid because of a missing CONST in the beginning. */
-   if (GET_CODE (ad) == PLUS
-       && GET_CODE (XEXP (ad, 0)) == SYMBOL_REF
-       && GET_CODE (XEXP (ad, 1)) == CONST_INT)
-     {
-       /* Add a CONST at the front of the expression and check to
-      see if we are successful: */
-
-       tem = gen_rtx (CONST, mode, ad);
-       if (strict_memory_address_p (mode, tem))
-     {
-       *loc = tem;
-       return 1;
-     }
-     }
- #endif
-
    return find_reloads_address_1 (ad, 0, loc, opnum, type, ind_levels);
  }


--- 4353,4358 ----
```

B.2 reload1.c versus reload1.c.org

```
*** reload1.cFri Aug 18 13:52:36 1995
--- reload1.c.orgThu Aug  3 11:26:39 1995
***************
*** 386,397 ****
    register rtx tem
      = gen_rtx (MEM, Pmode,
             gen_rtx (PLUS, Pmode,
! #ifndef THOR_HARD_FRAME_POINTER_INIT_RELOAD
!                gen_rtx (REG, Pmode, LAST_VIRTUAL_REGISTER + 1),
```

```
! #else
! /* Changed to work on Thor! */
!               gen_rtx (REG, Pmode, HARD_FRAME_POINTER_REGNUM),
! #endif
                GEN_INT (4)));
    spill_indirect_levels = 0;

--- 386,392 ----
    register rtx tem
      = gen_rtx (MEM, Pmode,
             gen_rtx (PLUS, Pmode,
!               gen_rtx (REG, Pmode, LAST_VIRTUAL_REGISTER + 1),
                GEN_INT (4)));
    spill_indirect_levels = 0;
```

B.3 reorg.c versus reorg.c.org

```
*** reorg.cWed Oct 18 13:05:18 1995
--- reorg.c.orgThu Jun 15 14:02:32 1995
***************
*** 2241,2257 ****
    if (INSN_FROM_TARGET_P (insn))
      return;

- #ifdef THOR_DELAY_SLOT_FIX
-   /* Change made to work on Thor. When using two delay-slots
-      'where' can point to a deleted insn. Then emit_insn_before will
-      abort. */
-   if (! INSN_DELETED_P (where))
-     emit_insn_before (gen_rtx (USE, VOIDmode, insn), where);
-   else
-     emit_insn_after (gen_rtx (USE, VOIDmode, insn), PREV_INSN (where));
- #else
    emit_insn_before (gen_rtx (USE, VOIDmode, insn), where);
- #endif

    /* INSN might be making a value live in a block where it didn't use to
       be.  So recompute liveness information for this block.  */
--- 2241,2247 ----
```

B.4 stmt.c versus stmt.c.org

```
*** stmt.cMon Sep 18 16:15:13 1995
--- stmt.c.orgThu Jun 15 14:08:17 1995
***************
*** 4784,4799 ****
  #endif /* HAVE_casesi */
  #endif /* CASE_VALUES_THRESHOLD */

- #ifndef THOR_CASE_VALUES_SPARSENESS
-       /* 10 is the original default value. A smaller value can
-      be appropriate if size is more important than speed. */
```

```
- #define THOR_CASE_VALUES_SPARSENESS 10
- #endif
-
        else if (TREE_INT_CST_HIGH (range) != 0
              || count < CASE_VALUES_THRESHOLD
              || ((unsigned HOST_WIDE_INT) (TREE_INT_CST_LOW (range))
!              > THOR_CASE_VALUES_SPARSENESS * count)
              || TREE_CODE (index_expr) == INTEGER_CST
              /* These will reduce to a constant.  */
              || (TREE_CODE (index_expr) == CALL_EXPR
--- 4784,4793 ----
  #endif /* HAVE_casesi */
  #endif /* CASE_VALUES_THRESHOLD */

        else if (TREE_INT_CST_HIGH (range) != 0
              || count < CASE_VALUES_THRESHOLD
              || ((unsigned HOST_WIDE_INT) (TREE_INT_CST_LOW (range))
!              > 10 * count)
              || TREE_CODE (index_expr) == INTEGER_CST
              /* These will reduce to a constant.  */
              || (TREE_CODE (index_expr) == CALL_EXPR
***************
*** 4891,4901 ****
  #ifdef HAVE_casesi
        if (HAVE_casesi)
          {
- #ifdef THOR_CASESI_QIMODE
-           enum machine_mode index_mode = QImode;
- #else
            enum machine_mode index_mode = SImode;
- #endif
            int index_bits = GET_MODE_BITSIZE (index_mode);
            rtx op1, op2;
            enum machine_mode op_mode;
--- 4885,4891 ----
```

B.5 jump.c versus jump.c.org

```
*** jump.cTue Aug  1 15:39:09 1995
--- jump.c.orgWed Oct 18 12:56:36 1995
***************
*** 437,449 ****
               rtx tem = find_equiv_reg (NULL_RTX, insn, 0,
                                sreg, NULL_PTR, dreg,
                                GET_MODE (SET_SRC (body)));
! /* PRESERVE_DEATH_INFO_REGNO_P doesn't work any more. That's why
!    we have introduced this new macro */
! #ifdef THOR_PRESERVE_DEATH_INFO_REGNO_P
               /* Deleting insn could lose a death-note for SREG or DREG
                  so don't do it if final needs accurate death-notes.  */
!              if (! THOR_PRESERVE_DEATH_INFO_REGNO_P (sreg)
!               && ! THOR_PRESERVE_DEATH_INFO_REGNO_P (dreg))
```

```
  #endif
                  {
                /* DREG may have been the target of a REG_DEAD note in
--- 437,447 ----
                rtx tem = find_equiv_reg (NULL_RTX, insn, 0,
                                          sreg, NULL_PTR, dreg,
                                          GET_MODE (SET_SRC (body)));
! #ifdef PRESERVE_DEATH_INFO_REGNO_P
                /* Deleting insn could lose a death-note for SREG or DREG
                   so don't do it if final needs accurate death-notes.  */
!               if (! PRESERVE_DEATH_INFO_REGNO_P (sreg)
!                   && ! PRESERVE_DEATH_INFO_REGNO_P (dreg))
  #endif
                  {
                /* DREG may have been the target of a REG_DEAD note in
```

B.6 function.c versus function.c.org

```
*** ../../gcc-2.7.0/function.cTue Oct 31 13:37:27 1995
--- ../../gcc-2.7.0/function.c.orgThu Jun 15 23:50:37 1995
***************
*** 1937,1947 ****
          && (GET_CODE (SET_DEST (x)) == REG
           || (GET_CODE (SET_DEST (x)) == SUBREG
               && GET_CODE (SUBREG_REG (SET_DEST (x))) == REG))
- #ifdef THOR_FIXUP_VAR_REFS_1
-         && x == single_set (insn))
- #else
          && x == single_set (PATTERN (insn)))
- #endif
        {
          rtx pat;

--- 1937,1943 ----
***************
*** 1986,1996 ****
          && (GET_CODE (SET_SRC (x)) == REG
           || (GET_CODE (SET_SRC (x)) == SUBREG
               && GET_CODE (SUBREG_REG (SET_SRC (x))) == REG))
- #ifdef THOR_FIXUP_VAR_REFS_1
-         && x == single_set (insn))
- #else
          && x == single_set (PATTERN (insn)))
- #endif
        {
          rtx pat;

--- 1982,1988 ----
***************
*** 3583,3595 ****
             /* If -ffloat-store specified, don't put explicit
                float variables into registers.  */
```

```
             || (flag_float_store
!                && TREE_CODE (TREE_TYPE (parm)) == REAL_TYPE)
! #ifdef THOR_NO_INITIAL_REG
!            || THOR_NO_INITIAL_REG
!                /* On Thor we do not want to copy the parameters
!                   to registers when entering a function.*/
! #endif
!            )
             /* Always assign pseudo to structure return or item passed
             by invisible reference.  */
             || passed_pointer || parm == function_result_decl)
--- 3575,3581 ----
             /* If -ffloat-store specified, don't put explicit
                float variables into registers.  */
             || (flag_float_store
!                && TREE_CODE (TREE_TYPE (parm)) == REAL_TYPE))
             /* Always assign pseudo to structure return or item passed
             by invisible reference.  */
             || passed_pointer || parm == function_result_decl)
```

B.7 config.sub versus config.sub.org

```
*** config.subThu Jul  6 08:30:14 1995
--- config.sub.orgWed Oct 18 12:38:29 1995
***************
*** 135,141 ****
           | alpha | we32k | ns16k | clipper | sparclite | i370 | sh \
           | powerpc | powerpcle | sparc64 | 1750a | dsp16xx | mips64 | mipsel
\
           | pdp11 | mips64el | mips64orion | mips64orionel \
!          | sparc | thor)
           basic_machine=$basic_machine-unknown
           ;;
      # Object if more than one company name word.
--- 135,141 ----
           | alpha | we32k | ns16k | clipper | sparclite | i370 | sh \
           | powerpc | powerpcle | sparc64 | 1750a | dsp16xx | mips64 | mipsel
\
           | pdp11 | mips64el | mips64orion | mips64orionel \
!          | sparc)
           basic_machine=$basic_machine-unknown
           ;;
      # Object if more than one company name word.
```

B.8 configure versus configure.org

```
*** configureThu Jul  6 08:29:27 1995
--- configure.orgWed Oct 18 12:36:00 1995
***************
*** 2274,2281 ****
           tm_file=sparc/sp64-elf.h
           extra_parts="crtbegin.o crtend.o"
```

```
          ;;
-      thor-*-*) #Generic version of Thor
-           ;;
  # This hasn't been upgraded to GCC 2.
  #    tahoe-harris-*)# Harris tahoe, using COFF.
  #         tm_file=tahoe/harris.h
--- 2274,2279 ----
```

APPENDIX C - Instruction set for Thor

All instructions found in the Thor microprocessor, with assembler mnemonic, format, hex code, instruction execution clock cycles, flags affected and possible exceptions are defined in *Table 10*.

Table 10 Thor's instructions set

Mnemonic	Instruction	Format	Hex Code	Cycles	Flags	Exception	Remark
ABS	Absolute Value	2	73	1	Z,N	10,11	
ADD	Add Integers	2a 2b 4a 4b	01	1	Z,N,C,U	2,10,11	
ADDF	Add Float	2a 2b 4a 4b	10	3	Z,N,C,U	2,10-12,14	
ADDI	Add Immediate	2a 2b 4a 4b	0C	1	Z,N	2,10,11	
ADDU	Add Unsigned	2a 2b 4a 4b	02	1	Z,N,C,U	2,10,11	
AND	Logic And	2a 2b 4a 4b	03	1	Z,N	2,10	
ANDI	Logic And Immediate	2a 2b 4a 4b	1F	1	Z,N	2,10	
CALL	Call Subprogram	2a 2b 4a 4b	27	2		2,5,10	Push stack
CALLP	Call Protected	2a 2b 4a 4b	34	2	UM	2,5,10	Push stack
CLL	Compare Lower Limit	2a 2b 4a 4b	0A	1		2,8,10	
CLRF	Clear Flags	2b 4b	30	1	All		
CMP	Compare Integers	2a 2b 4a 4b	09	1	Z,N,C,U	2,10	
CMPF	Compare Float	2a 2b 4a 4b	15	2	Z,N,C,U	2,10,14	
CMPI	Compare Integers Immediate	2a 4a	75	1	Z,N		
CMPU	Compare Integers Unsigned	2a 2b 4a 4b	0F	1	Z,N,C,U	2,10	
CUL	Compare Upper Limit	2a 2b 4a 4b	0B	1		2,8,10	
DIV	Divide Integers	2a 2b 4a 4b	18	5-20	Z,N	2,10,13	
DIVF	Divide Float	2a 2b 4a 4b	11	15	Z,N,C,U	2,10-14	
FBC	First Bit Changed	2b 4b	3D	1	Z,N,C,U	10	Push stack
FLT	Float Of Integer Value	2	72	2	Z,N,C,U	10	
FLUSH	Flush Cache	2	40	32-64		4	Privileged
HLT	Halt	2b 4a	00,FF	1		4	Privileged
INT	Integer Of Float Value	2	71	3	Z,N	10,11	
JR	Jump Relative	2a 2b 4a 4b	20	1		5	
JREQ	Jump Equal	2a 2b 4a 4b	21	1		5	
JRGE	Jump Greater Or Equal	2a 2b 4a 4b	22	1		5	
JRGT	Jump Greater Than	2a 2b 4a 4b	23	1		5	
JRLE	Jump Less Or Equal	2a 2b 4a 4b	24	1		5	
JRLT	Jump Less Than	2a 2b 4a 4b	25	1		5	
JRNE	Jump Not Equal	2a 2b 4a 4b	26	1		5	
JRX	Jump Relative Indirect	2a 4a	70	1		2,5	
LDX	Load Indirect	2a 2b 4a 4b	28	1		2,10	
MOD	Modulus	2a 2b 4a 4b	17	1	Z,N	2,10	
MTOS	Move Top Of Stack	2a 2b 4a 4b	2B	1		2,10	Move stack
MUL	Multiply Integers	2a 2b 4a 4b	1B	4	Z,N	2,10,11	
MULF	Multiply Float	2a 2b 4a 4b	12	3	Z,N,C,U	2,10-12,14	
MULI	Multiply Immediate	2a 2b 4a 4b	0D	4	Z,N	2,10,11	
MULL	Multiply Integers Long	2a 2b 4a 4b	19	4	Z,N	2,10	
MULU	Multiply Integers Unsigned	2a 2b 4a 4b	1A	4	Z,N	2,10	
NOP	No operation	2b 4b	3C	1			
NOT	Not	2a 2b 4a 4b	2A	1	Z,N	2,10	
OR	Logic Inclusive Or	2a 2b 4a 4b	04	1	Z,N	2,10	
ORI	Logic Or Immediate	2a 2b 4a 4b	16	1	Z,N	2,10	
POP	Pop	2a 2b 4a 4b	2C	1		2,10	Pop stack
POPR	Pop Register	2b 4b	37	1		4,10	Pop stack, priv. for EOS-TP
POPX	Pop Indirect	2b 4b	3B	1		2,9,10	Pop stack
PSH	Push	2a 2b 4a 4b	2D	1	Z,N	2,10	Push stack

Table 10 Thor's instructions set

Mnemonic	Instruction	Format	Hex Code	Cycles	Flags	Exception	Remark
PSHI	Push Immediate	2a 2b 4a 4b	2E	1	Z,N	2,10	Push stack
PSHR	Push Register	2b 4b	36	1		10	Push stack
PSHX	Push Indirect	2b 4b	3A	1	Z,N	2,9,10	Push stack
RAISE	Raise Exception	2	3E	1		Any	
RET	Return	2b 4b	35	1		2,5,10	Pop stack
RETU	Return To Usermode	2b 4b	38	1	UM	2,5,10	Pop stack
SBR	Subtract Reversed Integers	2a 2b 4a 4b	06	1	Z,N,C,U	2,10,11	
SBRF	Subtract Reversed Float	2a 2b 4a 4b	13	3	Z,N,C,U	2,10-12,14	
SBRU	Subtract Reversed Unsigned	2a 2b 4a 4b	0E	1	Z,N,C,U	2,10	
SETF	Set Flags	2b 4b	2F	1	All		
SL	Shift Left	2b	31	1	Z,N	10	
SLD	Shift Left Dynamic	2a 2b 4a 4b	1C	1	Z,N	2,10	
SR	Shift Right	2b	33	1	Z,N	10	
SRA	Shift Right Arithmetic	2b	32	1	Z,N	10	
SRAD	Shift Right Arithmetic Dynamic	2a 2b 4a 4b	1D	1	Z,N	2,10	
SRD	Shift Right Dynamic	2a 2b 4a 4b	1E	1	Z,N	2,10	
STX	Store Indirect	2a 2b 4a 4b	29	1		2,10	
SUB	Subtract Integers	2a 2b 4a 4b	07	1	Z,N,C,U	2,10,11	
SUBF	Subtract Float	2a 2b 4a 4b	14	3	Z,N,C,U	2,10-12,14	
SUBU	Subtract Unsigned	2a 2b 4a 4b	08	1	Z,N,C,U	2,10	
TA	Task Accept	2	76	2-9	TSI		
TAE	Task Accept End	2	79	2		2,10	Pop stack
TAS	Task Accept Start	2	78	1	Z,N		
TCA	Task Conditional Accept	2	77	2-9	TSI		
TCE	Task Conditional Entrycall	2a	7B	2	TSI	2	
TDLY	Task Delay	2	7D	1	TSI		Pop stack
TE	Task Entrycall	2a	7A	2-9	TSI	2	
TEE	Task Entrycall End	2	7C	1	Z	5	
TEST	Test	2b 4b	3F	1	Z,N	10	
TPTR	Task Pointer	2	39	1	Z,N	2,10	
TREG	Task Register	2	7E	1			Privileged
TSCH	Task Schedule	2	6F	1	TSI		
XOR	Logic Exclusive Or	2a 2b 4a 4b	05	1	Z,N	2,10	

APPENDIX D - List of C validation suites

By requesting at several newsgroups at internet we have obtained a list of companies offering C validation suites. These could be of obvious interest if one decide to officially qualify the compiler for use in space-borne computer systems.

1) In GNU CC version 2.3.3 a testsuite is included. It can be found at *prep.ai.mit.edu* (in the '`pub/gnu/`' directory and the file to be fetched is called '`gcc-2.3.3-testsuite.tar.gz`'). This is probably the most interesting testsuite, since it follows GNU CC's interpretation of ANSI C and supports the GNU CC's extensions to this standard.

2) SVVS - System V Verification Suite also tests the C compiler, especially the libraries. (From AT&T).

3) ACE C VALIDATION SUITE - Checks the compliance of a C compiler to the ANSI X3J11 draft standard. Also tests library functions for conformance to the X/OPEN standard or the SVID where these differ from their draft standard counterparts. The suite consists of over 50000 lines of source in about 600 different programs together performing over 2000 tests. The ACE C Validation suite is priced at 15500 dutch guilders (currently approximately $7500).
Contact Ge Gaal at *info@ace.nl* or *..!uunet!mcvax!ace!info*

ACE Associated Computer Experts bv
Van Eeghenstraat 100
1071 GL Amsterdam
The Netherlands

Phone: +31 20 646416
Telex: 11702 (ace nl)
Fax: +31 20 750389

4) ACE Associated Computer Experts bv provide SuperTest, the ANSI-C test and validation suite. Information available at address as below:

M.P. Roodzant
ACE Associated Computer Experts bv
van Eeghenstraat 100
1071 GL Amsterdam
The Netherlands

email: <*marco@ace.nl*>
tel: +31 20 6646416
fax: +31 20 6750389

5) MetaWare sells a small C validation suite. $2,000

903 Pacific Ave, Suite 201
Santa Cruz, CA 95060
(408) 429-6382

6) The Plum Hall Validation Suite for C $10,000
Plum Hall
1 Spruce Ave.
Cardiff, NJ 08232
(609) 927-3770

Plum Hall
PO Box 44610
Kamuela, HI 96743
808-882-1255
plum@plumhall.com (Dr. Thomas Plum)

Plum Hall also has a subset of the test suite called ’sampler’ which is labelled as ’freeware’.

7) The PERENNIAL Validation Suite for C Compiler Validation
PERENNIAL
4677 Old Ironsides Drive, Suite 450
Santa Clara, CA 95054
(408) 727-2255

Perennial
4699 Old Ironsides Dr.
Suite 210
Santa Clara, CA 95054
408-748-2900
support@peren.com (customer support)

Perennial sells the Perennial C++ Validation Suite.

Try contacting Perennial at: *info@peren.com*

8) C Compiler Torture Test - Checks a compiler against K&R. $20
The Austin Code Works
11100 Leafwood Lane
Austin, TX 78750-3409
(512) 258-0885

9) HCR offers a C Test Suite in various forms (50,000 tests to 350,000 tests)

HCR Corporation	Phone:	(416) 922-1937
130 Bloor Street West	Telex:	06-218072 HCR TOR
Suite 1001	Fax:	(416) 922-8397

Toronto, Ontario
Canada
M5S 1N5

10) RG Consulting, RoadTest

RG Consulting
396 Ano Nuevo #216
Sunnyvale, CA 94086
408-732-7839
rfg@netcom.com (Ron Guilmette)

Complete information on RG Consulting's ANSI C and C++ compiler test suites is available for anonymous FTP from *ftp.netcom.com* in the '`pub/rfg/roadtest`' directory. (If you cannot do FTP, let me know and I'll be glad to E-mail the file to you.)

Free samples!

11) Modena Software at: *modena@netcom.com*

Modena Software sells the Test++ Validation Suite.

APPENDIX E - Installation of GNU CC for Thor

Here follows a step by step scheme describing what kind of actions you must perform in order to install the compiler. During our project we did not have root access to the UNIX system we were working on, and therefore we have only tried to install the compiler in our own home directory.

- First, you must obtain a full version of GNU CC, including the source files. The version number must be 2.7.0, since this is the version our work is based on. The source code is freely accessible and can be fetched at numerous ftp[1] sites around the world. In Sweden you can preferably fetch GNU CC at the site: *ftp.sunet.se*. The official site for GNU CC is: *prep.ai.mit.edu*.
- All the files obtained from the ftp site are merged (with the UNIX command '`tar`') and packed (with another GNU program '`gzip`') into a single file, and by convention the file's suffix is '`.tar.gz`'. For example, if you want to fetch version 2.7.0 of GNU CC, you should ftp after file '`gcc-2.7.0.tar.gz`'. First, you should unpack the file with the command '`gunzip`', and then run the command '`tar -xvf`' to restore all files included in the '`.tar`'-file. Preferably, you place the files in the directory '`~/gcc-2.7.0`' (where the '`~`' in the beginning stands for your home directory).
- In all the source files in which we have altered, according to *APPENDIX B - 'Diff files of the changes in GNU CC source'*, you must now perform the changes. The source files are found in the directory '`~/gcc-2.7.0`'.
- Go to the directory '`~/gcc-2.7.0/config`'. Make a new directory '`thor`'. Place all files listed in *APPENDIX A - 'Listing of machine dependent files'* in this directory ('`~/gcc-2.7.0/config/thor`')

One possibility now is to build the compiler in the same directory as the source code (in our case: '`~/gcc-2.7.0`'), but we thought this approach to be kind of messy, even though this is the easiest way. We wanted the object files to be separated from the source code, and therefore we created the directory '`~/gcc-2.7.0/thor`' where we built our compiler. One drawback with this approach is that you must create links to some files, otherwise the building script will not be able to find them.

- First create a directory '`/thor`' under the directory '`~/gcc-2.7.0`'.
- Create a directory '`/cp`' under the directory '`~/gcc-2.7.0/thor`'.
- Go to the newly made directory ('`~/gcc-2.7.0/thor/cp`') and create the following links with the commands: '`ln -s ../../cp/lang-specs.h`' and '`ln -s ../../cp/lang-options.h`'.
- Create the directory '`~/thor`', then go to the directory '`~/gcc-2.7.0/thor`'.

Now we can finally start building the compiler. There are essentially three commands to run

1. Ftp means File Transfer Protocol, which essentially means a standardized format for fetching and sending files. If one's computer system is connected to internet is usually means that one can fetch files through this protocol.

which takes care of everything, one command to run the configuring script, which will create the makefile you are going to use, and two commands to run the makefile.

- Run the command '`../configure --target=thor --prefix=/userpath/thor`', where '`userpath`' is the complete path to your home directory. This will create a makefile that will install the final compiler under '`~/thor`'. If the prefix option is omitted, it will be installed under '`/usr/local`'.
- Run the command '`make LANGUAGES=c`'. After you give this command you could preferably take a cup of coffee, since this will take a while.
- Run the command '`make LANGUAGES=c install`'.

Now the building process is completed. If you want to reinstall the compiler you must run the command '`make distclean`' before you run the '`configure`' script. The building actions have now created various files (object-, executable files etc.).

- The object files of the GNU CC's source code are to be found in the directory '`~/gcc-2.7.0/thor`'.
- The executable file is called '`thor-gcc`' and is found in the directory '`~/thor/bin`'. It can be seen as a driver program which in turn calls the other executable files; '`cc1`' and '`cpp`' (found in the directory '`~/thor/lib/gcc-lib/thor/2.7.0`'). The file '`cpp`' is the preprocessor of the compiler, and in '`cc1`' the bulk of the compilation work is done.

One last thing worth mentioning is that the whole building process according to our '`t-thor`'-file and the '`Makefile.in`' uses the UNIX C compiler ('`cc`') together with a debugging flag ('`-g`'). This means that the compiler is built with debugging information in the object files, i.e. larger files and the execution may be a bit slow. If your system is equipped with a normal installation of GNU CC you can use that one instead, together with optimization if you wish. Here is the procedure to accomplish that:

- Edit the '`t-thor`'-file and add the following lines; '`CC = gcc`' and '`CFLAGS = -O`'.
- Alternatively you can add the lines above to the '`make`' command line explained above.

When the building process starts you can see that the files are compiled with '`gcc -O`' instead of '`cc -g`'.