A translator from RISC code into MC68020 code.

by

Peter K. S. Tan

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Bachelor of Science in Electrical Engineering

at the

Massachusetts Institute of Technology

June 1989

© Peter K. S. Tan 1989

The author hereby grants to MIT permission to reproduce and to
distribute copies of this thesis document in whole or in part.

Signature of Author

______________________________

Department of Electrical Engineering and Computer Science
May 22, 1989

Certified by

______________________________

Prof. Rishiyur S. Nikhil
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by

______________________________

L. A. Gould
Chairman, Department Committee on Undergraduate Theses

JUN 16 1989
A translator from RISC code into MC68020 code.

by

Peter K. S. Tan

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 1989 in partial fulfillment of the requirements for the degree of Bachelor of Science in Electrical Engineering

Abstract

The parallel machine language of a dataflow machine is a dataflow graph (similar to a precedence graph). A node in the dataflow graph is an instruction which executes only when the operands it requires are available. The edges connecting the nodes in the graph represent the essential data dependences among instructions. The instructions can be represented by the Explicit Token Store (ETS) instruction set. The work that this thesis addresses is in the design of the back end of a translator that has the objective of translating Explicit Token Store (ETS) code into Motorola MC68020 machine code. The front end of the translator translates ETS code into intermediate Reduced Instruction Set Computing (RISC) code and the back end further translates the RISC code into MC68020 code. The back end is also responsible for producing the bootstrap code and the runtime system code of the dataflow machine. The runtime system code provides the frames (workspace) management code and the token queue management code that are needed to simulate the runtime of the dataflow machine. The target hardware for the MC68020 code is the SUN-III workstations.

Thesis Supervisor: Prof. Rishiyur S. Nikhil

Title: Assistant Professor of Electrical Engineering and Computer Science
Acknowledgements

I truly thank God very much for helping me through all these years.

I would like to thank Professor Nikhil for being a most patient thesis supervisor and a great educator. Without his guidance and advice, this project would have been too difficult.

I would like to thank Lina, my thesis partner who implemented the front end. Without her brilliant work, this project would never have got off the ground. Truly, I would like to thank her even more for her emotional support and patience during the long, long debugging stages. Funny though, "Are you sleeping?" is one of her favourite comments.

I also thank members of CSG, Jamey Hicks, Dr. Jonathan Young and Dr. Andy Boughton, for their help and advice.

I would like to thank my friends namely;

Joon Hin, my buddy for his 4.23 inch-wide smile and valuable friendship.

Joon Aun, the best and craziest roommate in the world and also for providing those wonderful noises in our room.

Sam Ong Hock Soo, who inspired me greatly when I was young and who still inspires me today,

George "Hammer" Lyden, my teammate on the MIT tennis team, for being a rare good friend and a good listener in MIT,

Datuk Chong Eng Kuan, Billy Liew, Fiona Tan, Tan Poh Beng, Mr. Khoo Kay Tatt, Frankie Lim, Francis Su, Harry Wong, Fadzil Daud, Fong Loong Keh, Kenny Khoo, Tammy Kang,

and the rest of my friends from all around the world.

Also, I would like to thank the coffee maker in the CSG kitchen.
I dedicate this to my family, specially to Papa and Mummy,

and

Pat, David, Paul and Ah Ngai.

I'll always be grateful to each and every one of you for being my family
## Contents

1 Introduction
   1.1 The Role of the Translator ............................................. 10
   1.2 Making it all work ....................................................... 13

2 Explicit Token Store .......................................................... 14

3 The Front end of the Translator ............................................. 19
   3.1 RISC world Memory Map ................................................ 20
   3.2 RISC code Generation ................................................... 21

4 RISC world Runtime System .................................................. 25
   4.1 Directives that reserves memory ...................................... 25
   4.2 Declaration of Registers ................................................ 26
   4.3 The initialization subroutine that is called from the MC68020 bootstrap code .................................................. 29
   4.4 Queue and activation frame subroutines that are called by the RISC world ETS program .................................................. 30
   4.4.1 Store-in-frame. ......................................................... 30
   4.4.2 Token-q-load. ......................................................... 31
   4.4.3 Token-q-store. ......................................................... 32
   4.4.4 Get-context. ......................................................... 33
   4.4.5 return-context ....................................................... 36

5 The Back end of the Translator ............................................. 43
   5.1 Translating RISC code into MC68020 code. ............................ 43
      5.1.1 Translation of RISC directives: .................................. 43
      5.1.2 Translation of RISC registers .................................... 46
      5.1.3 Translation of RISC instructions .................................. 47
   5.2 Producing MC68020 bootstrap and exit code. .......................... 50

6 Conclusion ............................................................................. 52
   6.1 Optimizations and Improvements. ...................................... 52

A Interactive C program, bootstrap and exit code .......................... 54
   A.1 Interactive C program .................................................... 54
   A.2 MC68020 Bootstrap code ................................................ 54
A.3 MC68020 Exit code ........................................... 55
A.4 Calling the MC68020 bootstrap code from the interactive C program ... 56
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Block Diagram</td>
<td>11</td>
</tr>
<tr>
<td>1.2</td>
<td>Translation from ETS to MC68K code</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>A section of a dataflow graph</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Instruction and activation frame</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>An example ETS Pipeline</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Memory Map</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>match location and token in queue</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>ETS instruction +</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Translation of ETS into RISC code</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>queue and frame buffer</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Calls from bootstrap code and jumps to exit code</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Direction of travel for queue</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>codeblock and its descriptor</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Frame buffer during get-context</td>
<td>35</td>
</tr>
<tr>
<td>4.6</td>
<td>return context</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Cleanup</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>bootstrap (entry) and exit subroutine</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Tables

3.1 RISC instruction set used by Front end. ......................... 20
5.1 MC68020 instruction set. ......................................... 49
5.2 continuation: MC68020 instruction set. ......................... 50
Chapter 1

Introduction

A dataflow architecture, the Explicit Token Store (ETS) architecture, is presently being developed in the Laboratory of Computer Science in the Massachusetts Institute of Technology (MIT). Concurrently, a separate project (the VonMon project) was undertaken with the objective of executing an ETS program on a von Neumann machine. The VonMon project is a translator from ETS code to von Neumann code and the von Neumann target hardware is a SUN-III workstation. For modularity, the translator is divided into two modules; the front end and the back end. The purpose of this thesis is to describe the implementation of the back end.

Chapter 2 briefly introduces the execution characteristics of the ETS machine via a simple example. Chapter 3 describes the front end [3] of the translator that produces Reduced Instruction Set Computing (RISC) code from ETS code. The RISC code is the intermediate code representation for the translator. Chapter 4 describes the runtime system. The runtime system is used for runtime system calls from the RISC code generated by the front end. The runtime system is hand coded in RISC code. Chapter 5 describes the back end of the translator. The back end produces MC68020 code for execution on the SUN-III workstation. Discussion focuses on 1) the translation from RISC code into MC68020 code and 2) the bootstrap and exit code. The final chapter provides a conclusion and suggestions for future development of the VonMon project.

We now present an overview of the translator.
1.1 The Role of the Translator

The compiler of high level ID language programs has the capability of producing TTDA machine code and ETS machine code. As shown in figure 1.1, the TTDA code is executed by the GITA simulator whereas the ETS code is intended for execution on its target hardware, the Monsoon machine which is presently under development. The role of the translator is to translate ETS code into von Neumann code (MC68020 code). The MC68020 code is executed on the von Neumann target hardware, a SUN-III workstation. An important note should be taken here that presently, the capability of the translator is restricted to ETS code that does not involve I-structure operations [5].

To encourage modularity, the translator is divided into the front end and the back end. The front end produces RISC code and the back end produces MC68020 code. The RISC code is the intermediate code representation for the translator and the MC68020 code is the von Neumann target hardware’s machine level code. The entire translator is written in Common LISP.

At this point it would be wise to define the intermediate code as the RISC world and the target hardware’s code as the MC68K world, since both codes are Von Neumann codes.

The translation from ETS code to MC68020 code is further described in detail as shown in figure 1.2.

The front end translates the ETS program into a RISC program. The RISC program is the RISC world ETS program.

Before passing the RISC program to the back end, the code is concatenated with the hand coded RISC runtime code. The concatenation can either be done by hand or by the front end. The concatenated RISC code is then passed to the back end. The back end translates the RISC code and produces MC68020 code (MC68K world ETS program plus the MC68K runtime code) into an assembly file. The back end also appends hand coded MC68020 bootstrap code (entry code) to the beginning of the assembly file and hand coded MC68020 exit code to the end of the assembly file.
Figure 1.1: Block Diagram
Figure 1.2: Translation from ETS to MC68K code
1.2 Making it all work

In order to create an executable object file, the MC68020 assembly file has to be hand linked with a C program. The C program is hand coded and at runtime, it interacts with the user, i.e., prompts the user for the argument to the ETS program. The C program calls an entry subroutine (the bootstrap code) in the assembly code and passes the argument to it. After saving the argument, the entry subroutine makes an instruction jump to a $MC68K\ world$ runtime subroutine that performs initialization operations (we will refer to this as the initialization subroutine in the runtime system). Thereafter, the execution of the $MC68K\ world$ ETS program begins. Upon completion of the program, control is passed to the exit code. The exit code prints out the results and returns to the C program.
Chapter 2

Explicit Token Store

The purpose of this chapter is to introduce the basic concept of a dataflow machine by stepping through an example and providing explanations along the way. This is not intended to be a thorough inspection of a dataflow machine and hence, the interested reader should refer to Dr. Greg Papadopoulos's doctoral thesis [5] for more information. We give here a simple example of a program executing on the ETS machine. Initially, a higher level program coded in ID [4] has been compiled into ETS code. The ETS code resides in local memory of a dataflow processor and the scheduling of its instructions is determined by a dataflow graph. A section of a dataflow graph is given in figure 2.1.

Assume here a single dataflow processor. The nodes are the machine instructions and the edges represent the data dependences between the nodes. The data on the edges are the operands to the instructions (nodes) and they are represented as tokens. A token contains a tag and the operand itself (the operand is 64-bits). The tag further contains a pointer to the instruction to be executed and the address of an activation frame. The concept of an activation frame will be explained shortly.

In our example, instruction * can only be executed after the result from instruction + is computed. This scheduling principle also applies to instruction + since it requires two operands in edges a and b in order to execute. In order to implement this principle, the ETS machine uses a waiting-matching rule. The waiting-matching rule uses the following algorithm:
IF one edge has an operand,
THEN {
    check if other edge has an operand in the activation frame
    IF yes (matches), execute the instruction and send the result to the
token queue
    ELSE { store the present operand in an activation frame,
        set the presence bits and
        fetch the next token (operand) from top of queue }
}

Here, we have just introduced two new terms; queue and activation frame. The
queue uses a FIFO rule to store the result token and send a new token into the processor
everytime an instruction is executed. The activation frame is a block of contiguous local
memory used to store operands of instructions that do not have their respective matching
operands. This storing operation puts the instructions into a waiting state. Tokens that
belong to the same invocation of a function (or, more generally, an ETS code block)
have the same activation frame. It is important to realize that a function can be called
recursively. In this situation, tokens of the same instruction will have different contexts
and as a result, different activation frames.
Physically, the instructions and a corresponding activation frame for a context are shown in figure 2.2. The relevant dataflow graph for figure 2.2 is shown in figure 2.1.

Please note that an instruction location supplies the opcode, the offset (r) within the activation frame for the match, and the destinations of the result token. A location in the activation frame will contain present/empty bits and the value of the operand waiting to be matched.

In our example, the + operator at statement s has two output destinations: the instruction * at statement s+1 and the instruction - at statement s+2. Thus the destinations for the + operator are encoded as +1 and +2, respectively. The instruction must also encode the port number (left or right) of a destination instruction. We have omitted it in this diagram for simplicity. The + operator also specifies the offset in the activation frame, in this case offset +0, where the match will take place. Notice that the - operator can reuse this location, as neither input to the instruction - can arrive until after the instruction + has executed.

Figure 2.3 shows an example pipelined implementation of an ETS processor. The instruction fetch is performed before the operand matching. This is necessary because the instruction encodes the offset, r, of the rendezvous location in the activation frame pointed to by the incoming token's tag. The operand matching stage queries the presence bits on location c+r. If the slot is empty, the token's value is written into the slot
Figure 2.3: An example ETS Pipeline
and a "bubble" or no-operation is submitted to the ALU. On the next step the bubble propagates to the form token stage which, in turn, does not insert any tokens into the token queue.

If the slot is full, the operand from the slot is extracted, and this operand along with the value on the token being processed are submitted to the ALU. The destination tags are computed in the form tag stage, in parallel with the ALU operation. Finally, new token(s) are constructed in the form token stage, which concatenates the destinations from the form tag unit and the result of the ALU. The new tokens are inserted into the token queue.
Chapter 3

The Front end of the Translator

The main purpose of the front end [3] is to translate each ETS instruction into its equivalent von Neumann RISC code representation. Please note that the RISC code representation of a single ETS instruction will consist of a block of RISC instructions. We chose RISC to be the intermediate code representation because its instruction set is simple. The simplicity of the instruction set will enable the back end to easily translate the instructions into any target hardware’s machine code. For the VonMon project, the instruction set similar to the R2000 [2], a RISC chip, is chosen.

Please note that the intermediate code is a list of the RISC instructions and each instruction, in turn, is a list consisting of the opcode and its operands. An example of the intermediate code is shown below:

(  
  (LI RESULT_PRESENT_REG PRESENT)  
  :  
  :  
  :  
  (ADDI RESULT_LO_REG RV1LO ZERO)  
  (J START)  
)  

The syntax of a RISC instruction [2] is:

( <opcode> <destination-register> <source-operand> <source-operand> )
( <opcode> <destination-register> <source-operand> )

depending on the instruction opcode.

The RISC instruction set that is used by the front end to produce RISC code is shown in table 3.1.

<table>
<thead>
<tr>
<th>Instruction Description</th>
<th>Opcode</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Address</td>
<td>LA</td>
<td>LA rd relocatable-address</td>
</tr>
<tr>
<td>Load Byte</td>
<td>LB</td>
<td>LB rd offset base-register</td>
</tr>
<tr>
<td>Load Word</td>
<td>LW</td>
<td>LW rd offset base-register</td>
</tr>
<tr>
<td>Store Byte</td>
<td>SB</td>
<td>SB rs offset base-register</td>
</tr>
<tr>
<td>Store Word</td>
<td>SW</td>
<td>SW rs offset base-register</td>
</tr>
<tr>
<td>Load Immediate</td>
<td>LI</td>
<td>LI rd immediate-value</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>AND</td>
<td>AND rd rs1 rs2</td>
</tr>
<tr>
<td>Bitwise ANDI</td>
<td>ANDI</td>
<td>ANDI rd rs immediate-value</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>SUB rd rs1 rs2</td>
</tr>
<tr>
<td>Addition</td>
<td>ADD</td>
<td>ADD rd rs1 rs2</td>
</tr>
<tr>
<td>Add Immediate</td>
<td>ADDI</td>
<td>ADDI rd rs immediate-value</td>
</tr>
<tr>
<td>Multiply</td>
<td>MULT</td>
<td>MULT rs1 rs2</td>
</tr>
<tr>
<td>Branch on Equal</td>
<td>BEQ</td>
<td>BEQ rs1 rs2 relocatable-address</td>
</tr>
<tr>
<td>Branch on Not Equal</td>
<td>BNE</td>
<td>BNE rs1 rs2 relocatable-address</td>
</tr>
<tr>
<td>Branch on Greater/Equal Zero</td>
<td>BGEZ</td>
<td>BGEZ rs relocatable-address</td>
</tr>
<tr>
<td>Branch on Greater Than Zero</td>
<td>BGTZ</td>
<td>BGTZ rs relocatable-address</td>
</tr>
<tr>
<td>Branch on Less/Equal Zero</td>
<td>BLEZ</td>
<td>BLEZ rs relocatable-address</td>
</tr>
<tr>
<td>Branch on Less Than Zero</td>
<td>BLTZ</td>
<td>BLTZ rs relocatable-address</td>
</tr>
<tr>
<td>Jump</td>
<td>J</td>
<td>J relocatable-address</td>
</tr>
<tr>
<td>Jump Register</td>
<td>JR</td>
<td>JR rs</td>
</tr>
<tr>
<td>Move From HI Register</td>
<td>MFHI</td>
<td>MFHI rd</td>
</tr>
<tr>
<td>Move From LO Register</td>
<td>MFLO</td>
<td>MFLO rd</td>
</tr>
</tbody>
</table>

Table 3.1: RISC instruction set used by Front end.

3.1 RISC world Memory Map

In the RISC world, we need memory for the code, activation frames and the token queue. The memory map for the RISC world is as shown in figure 3.1.

Please refer to figure 3.2. As mentioned in chapter 2, the ETS token has 64-bit values as operands. Therefore, a match location in an ETS activation frame has presence bits followed by 64 bits of the operand value. A corresponding match location in the RISC
word activation frame will have one byte to represent the presence bits, 3 bytes to encode the type of the operand and 8 bytes to store the operand value. We actually need only 1 byte to encode the type but to maintain 4-byte boundaries we use 3 bytes to produce a total of 12 bytes for a match location in the RISC world.

In the ETS machine, the token queue is maintained in the hardware. In the RISC world we will simulate the ETS queue (a "wrap around" queue) in memory. Since each ETS token has a 32-bit frame pointer, a 32-bit instruction pointer and a 64-bit operand value, we will need at least 16 bytes in the RISC world to simulate a single ETS token. In our simulation we add another 4 bytes to represent the type of the operand value (once again, we preserve the 4-byte boundary) giving a total of 20 bytes for a simulated ETS token in the RISC world queue.

There is a runtime system to maintain the token queue and the activation frames. The runtime system is used for runtime system calls by the RISC world ETS program, and it will be presented in the next chapter.

3.2 RISC code Generation

The RISC code generation is the main task of the front end. For an example, let’s look at translating the simple ETS instruction +.

Please refer to figure 3.3 and 3.4. In the RISC world, in order to synchronize the two input operands so that both operands are present to execute ("fire") the instruction, Von Neumann synchronizing code is needed. Assume that there are three registers, FP, IP and Rv in the RISC world, and that they have been loaded with the values in a token,
Figure 3.2: match location and token in queue

Figure 3.3: ETS instruction +
Figure 3.4: Translation of ETS into RISC code

i.e., FP is loaded with the address of an activation frame, IP is loaded with the address of the instruction to be executed and Rv is loaded with the value of the operand itself.

The synchronizing code checks if the specified match location in the activation frame pointed to by the FP has a present operand. The match location is specified because the offset r in the ETS activation frame is available to the translator (from the ETS instruction) at translation time. The front end maps the offset r onto the RISC world offset d (figure 3.4). If the operand is absent in the RISC world match location, the synchronizing code stores the value in Rv into the match location, sets the presence byte and fetches the values of the next token from the top of the queue into the FP, IP and Rv registers. A jump is then executed to the instruction that is pointed to by the IP.

Otherwise, if the operand in the RISC world match location is present, the synchronizing code branches off to perform the RISC world add operation. Following that, control is passed to the fork code.

In our example, the result of the add operation needs to be passed to two destinations. From the destination field of the ETS instruction, (during translation time) the
front end can determine the corresponding RISC world destination instruction. The fork code produces a token that has as its destination the first RISC instruction of the synchronizing code for the ETS instruction Y. This token is pushed into the bottom of the queue. Instead of producing another token that has the destination instruction X (the left instruction), the jump code automatically loads the FP register with the present frame address, the IP register with the address of the first RISC instruction of the synchronizing code for the ETS instruction X and the Rv register with the result of the add operation. A jump is then executed to the instruction that is pointed to by the IP. This mode of forking will ensure that control is always passed down the left branches of the dataflow graph.

Our presentation of the front end is not complete since there will always be different RISC code for different ETS instructions. The interested reader should refer to the front end documentation found in Lina Muryanto’s thesis [3].
Chapter 4

RISC world Runtime System

The RISC world runtime system is used for system calls from the RISC world ETS program. It maintains the queue and the activation frames in memory. The RISC world runtime system code consists of

1. directives that reserves memory for the token queue and activation frames,

2. declaration of registers,

3. the initialization subroutine that is called from the MC68020 bootstrap code,

4. queue and activation frame subroutines that are called by RISC world ETS program.

4.1 Directives that reserves memory

Memory is reserved for the token queue and the activation frames.

For the token queue,

(\texttt{.DATA})

(\texttt{LABEL STARTQBUFFER .SKIP QBUFSIZEINBYTES})

reserves QBUFSIZEINBYTES bytes in data memory for the queue buffer. The syntax "LABEL" is used to declare STARTQBUFFER as a label and ".skip" is used to reserve memory starting from the present location counter.

For the activation frames,
(DATA)

(LABEL STARTFRAMEBUFFER .SKIP NMAX+2*MLSIZE)

reserves ((NMAX + 2) x MSIZE) bytes in data memory for the frame buffer.

### 4.2 Declaration of Registers

A register is declared by the following directive in RISC:

(\textit{DATA})

(LABEL FF_POINTER .WORD STARTFRAMEBUFFER)

FF_POINTER is loaded initially with a word (4 bytes) value of STARTFRAMEBUFFER.

At this point, an illustration (figure 4.1) of the queue and frame buffers in memory would be helpful for the reader to understand the role of the RISC registers.

The size of the registers are 32 bits and they can be categorized below:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>frame pointer</td>
</tr>
<tr>
<td>IP</td>
<td>instruction pointer</td>
</tr>
<tr>
<td>TYPE</td>
<td>type of token value</td>
</tr>
<tr>
<td>RV1HI</td>
<td>high value of first token</td>
</tr>
<tr>
<td>RV1LO</td>
<td>low value of first token</td>
</tr>
<tr>
<td>RV2HI</td>
<td>high value of second token</td>
</tr>
<tr>
<td>RV2LO</td>
<td>low value of second token</td>
</tr>
<tr>
<td>PORT</td>
<td>port specifier</td>
</tr>
<tr>
<td>RRETURN</td>
<td>contains the address of the return instruction</td>
</tr>
</tbody>
</table>

-----------------------

TQHD | token queue head |
TQTL | token queue tail |
QBOT | the physical end of the queue |
a single free frame at initialization containing n 12-bytes of memory

Figure 4.1: queue and frame buffer
QTOP the physical top of the queue

FRAMEBUFFERPOINTER contains the physical starting address of the header of the buffer containing the frames
ENDOFFRAMEBUFFERPOINTER contains the physical starting address of the header that marks the end of the buffer containing the frames
FF_POINTER points to a free frame
OLD_FF_POINTER points to previous free frame
ENDOF_FF_POINTER points to the end of the free frame pointed to by the FF_POINTER
F_POINTER points to an activation frame

<comment: the following registers are used as temporary auxiliary registers by GET_CONTEXT and RETURN_CONTEXT>

FRAMESIZEARGREG contains the size (in 12 bytes multiples) of the frame to be allocated by GET_CONTEXT
FRAMESIZEARGINBYTES contains the size in bytes of the frame to be allocated by GET_CONTEXT

FF_SIZEREG1 the size (in 12 bytes multiples) of a free frame; this register is used to compare between the size of a free frame and value of FRAMESIZEARGREG

FF_SIZEINBYTESREG the size in bytes of the free frame

FFNHEADER_SIZEINBYTESREG the size of the free frame plus its header in bytes
4.3 The initialization subroutine that is called from the MC68020 bootstrap code

The bootstrap code is MC68020 code that stores the argument passed from the C program (the C program prompts the user for the argument at runtime) into a register. The bootstrap code then jumps to the initialization subroutine in the runtime code as shown in figure 4.2.

The purpose of the initialization subroutine is to:

1. Initialize the headers of the frame buffer:
There are two headers that needs to be initialized. The initialization of these headers will create the initial free frame list as shown in figure 4.1. A header is located at the top of a free frame and contains the following information: i) the size (in 12 byte multiple) of the free frame and ii) pointer to the next free frame. The size of the header is 12 bytes so that a header exactly occupies one match location.

2. Jump to the RISC world ETS program:

The RISC world ETS program will create the first tokens [3] and store the tokens into the queue. Thereafter, the top token in the queue is fetched and a jump is executed to the instruction pointed by the IP of the token. The execution of the entire RISC world ETS program will follow.

4.4 Queue and activation frame subroutines that are called by the RISC world ETS program

These are the subroutines that uses the token queue and the activation frames.

4.4.1 Store-in-frame.

The specifications of the subroutine is shown in below:

- **STORE_IN_FRAME**:
  
  - **Function**:
    
    * Store current token < c.s_p, v > into the activation frame.
    * Set presence byte
  
  - **Arguments**:
    
    * RV1HI = Hi-value of v
    * RV1LO = Lo-value of v
    * FP = Frame-pointer c
    * IP = Instruction-pointer + Port, i.e. s + p
* RRETURN = Return address upon return from STORE_IN_FRAME

- Output:
  * None

### 4.4.2 Token-q-load.

The specifications of the subroutine is shown below:

- **TOKEN_Q_LOAD**

  - Function:
    * Load a token from the queue into the processor registers.

  - Arguments:
    * None

  - Output:
    * FP ← Frame-pointer
    * IP ← Instruction-pointer
    * PORT ← Port to current instruction
    * RV1HI ← Hi-value
    * RV1LO ← Lo-value

The token queue is implemented as a wrap around queue. The direction of travel for the tqhd and tqtl is shown in figure 4.3.

The algorithm for popping a token from the top of the queue is:

```plaintext
if queue is full
then
  { unset the qfullflag }

load in the values of the top token into registers

tqhd ← tqhd + sizeof(one token)
```
if tqhδ hits the bottom
then
{ set tqhδ to top of queue}
return

The qfullflag is used to differentiate between the queue at its full state and the queue being at its initial state (empty state). At both states, the tqhδ is equal to the tql.  

4.4.3 Token-q-store.  

- **TOKEN_Q_STORE**  
  
  - Function:
    
    * Store current token < c.sₚ, v > into the queue.
  
  - Arguments:
    
    * RV1HI = Hi-value of v  
    * RV1LO = Lo-value of v  
    * R1 = Instruction-pointer + Port, i.e. s + p  
    * R2 = Frame-pointer c
* RRETURN = Return address upon return from TOKEN_Q_STORE

- Output:

  * None

The algorithm for storing a token to the bottom of the queue is:

```c
if queue is full
then
{ indicate that queue is full by printing
  an error message }
else
{
  push the values of the registers into the
  bottom of the queue

  tqttl <-- tqttl + sizeof(one token)

  if tqttl hits the bottom
  then
  { set tqttl to top of queue}

  if tqttl equal to tqttd
  then
  { set the qfullflag }

  return
}
```

4.4.4 Get-context.

Shown in figure 4.4 is an example of a code block and its descriptor. The interested reader should refer to Lina Muryanto's thesis [3] for an explanation of the code block and its descriptor. The codeblock descriptor provides information pertaining to the size

33
of the frame associated with the code block and this information is used when allocating a frame for the code block.

- **GET\_CONTEXT**

  - **Function:**
    * Allocate a frame of a given size, and return a tag.

  - **Arguments:**
    * R1 = Address of code-block descriptor
    * PORT = Port of the result tag
    * RRETURN = Return address upon return from GET\_CONTEXT

  - **Output:**
    * RV1HI ← Frame-pointer which points to the allocated frame
    * RV1LO ← Instruction-pointer + port, where the instruction-pointer points to the indirection table at the entry of the called code-block.

The frame buffer is shown in figure 4.5.
GET-CONTEXT

Figure 4.5: Frame buffer during get-context
Referring to figure 4.5, the algorithm of the get-context is as follows:

```
set FF_POINTER to point to
the first header of the frame buffer

loop:
  if size of present free frame pointed
to by FF_POINTER is greater
  or equal to the frame size argument,
  then jump to free frame found

else {
  set FF_POINTER to point to
  next free frame

  go back to loop}

free frame found: allocate the frame starting from the
end of the free frame

set F_POINTER to point to the
header of the frame

return F_POINTER

return
```

4.4.5 return-context

- *RETURN_CONTEXT*

  - Function:

    * Deallocate a frame of a given pointer.

  - Arguments:
DURING RETURN-CONTEXT

AFTER FRAME HAS BEEN RETURNED

Figure 4.6: return context

* R2 = Frame-pointer which points to the frame to be deallocated.

* RRETURN = Return address upon return from RETURN_CONTEXT

- Output:

  * None

The algorithm of the return-context is more involved because it requires cleaning up free frames that are exactly adjacent to each other. Cleaning up requires combining any two adjacent free frames into a single free frame.

We first explain the return-context subroutine.

For example, as shown in figure 4.6, the algorithm for return-context is as follows:
set FF_POINTER to point to
the first header
of the frame buffer

set F_POINTER to point to
frame that is to be returned

loop:

if FF_POINTER points to the
next free frame
located after the frame
pointed by F_POINTER
then
go to next free frame found

else {
  set OLD_FF_POINTER to
  value of FF_POINTER

  set FF_POINTER to point
to next free frame in
the list

  go to loop
}

next free frame found: change the frame pointed to
by the F_POINTER
into a free frame by
changing its header

comment: The new header will
point to the
next free frame
(which is pointed
to by the FF_POINTER)
set the free frame pointed
by the OLD_FF_POINTER to
point to the "return" frame.

comment: Now the "return" frame
is a free frame in the
list of free frames.

goto cleanup

We use FF_POINTER to find the next free frame located after the return frame. Therefore, we define the free frame pointed by the FF_POINTER as the next free frame. We need to locate the next free frame because we want it (the next free frame) to be pointed by the return frame.

We use the OLD_FF_POINTER to point to the free frame located before the return frame. We need to locate this free frame because we want it (the free frame located before the return frame) to point to the return frame.

Cleanup is called because the frame that was returned may be adjacent to another free frame. Therefore, it is called whenever a frame has been returned to the free frame list. Looking at the last snapshot in figure 4.6 and first snapshot in figure 4.7, FF_POINTER is now pointing to the next free frame after the return frame. Going down the list of free frames, the address of the second free frame located after the "return" frame is used as the top limit (refer to figure 4.7). The top limit is used to determine the end of the cleanup procedure.

The algorithm of cleanup is:

cleanup: set toplimit by saving the address
of the second free frame (down the
list of free frames)
located after the "return" frame

use the OLD_FF_POINTER to point to
the free frame
DURING CLEANUP

Figure 4.7: Cleanup
located before the "return" frame

reset FF_POINTER to point to the next free frame located after the free frame pointed by the OLD_FF_POINTER

loop1: if FF_POINTER points to toplimit then cleanup is finished else {

if the next free frame (pointed by FF_POINTER) is adjacent to the present free frame (pointed by OLD_FF_POINTER)

then jump to combine

else {

set OLD_FF_POINTER to point to next free frame located down the its list

set FF_POINTER to point to next free frame down its list

go to loop1

}

}

combine: combine the two free frames into one
set FF_POINTER to point to next
free frame down its list

preserve the OLD_FF_POINTER

go to loop1
Chapter 5

The Back end of the Translator

The back end of the translator is written in Common LISP. The back end has two objectives:

1. translating the RISC code into MC68020 code and

2. producing MC68020 bootstrap code (MC68020 entry code) and MC68020 exit code.

5.1 Translating RISC code into MC68020 code.

The source RISC code is the RISC world ETS program which is concatenated with the RISC runtime system. The result of the translation is the MC68020 code, which consists of the MC68K ETS program and MC68K runtime system, and the code is written into an assembly file.

Translation of RISC code involves

1. translation of RISC directives

2. translation of RISC registers and

3. translation of RISC instructions.

5.1.1 Translation of RISC directives:

1. LABEL directive
In order to identify a label in the RISC code, the identifier \textit{LABEL} must always precede the label itself. For example:

\begin{verbatim}
in the RISC world,

(LABEL FOO J BAR)
\end{verbatim}

is the RISC \textit{J} instruction with the label \textit{FOO}

The back end would translate that instruction into

\begin{verbatim}
in the MC68K world,

.even
FOO:    jmp BAR
\end{verbatim}

The \textit{.even} directive in the \textit{MC68K world} is added to ensure that the FOO is located on an even-byte boundary because:

i) the lowest order bit of the IP register is used to encode the port and it will be masked out when used to jump to an instruction. Therefore, the IP points to even addresses only.

ii) all branches and jumps must be made to an instruction on an even boundary. A branch or jump to an odd boundary will result in an error. Please refer to the SUN-III Assembly Language Reference manual [1].

2. \textit{.EQU} directive

The \textit{.EQU} directive is used to declare constants.

\begin{verbatim}
in the RISC world,

(.EQU ONE 1)
\end{verbatim}

is translated into

\begin{verbatim}
in the MC68K world,

ONE = 1
\end{verbatim}
3. **.WORD directive and .SKIP directive**

The `.WORD` directive is used to initialize 4 bytes of the present data memory location. The `.SKIP` directive is used to reserve a block of specified length starting at the present data memory location.

```plaintext
in the RISC world,

(LABEL XVAR .WORD 1)
(LABEL TABLE .SKIP 1000)
```

is translated into

```plaintext
in the MC68K world,
  .even
XVAR:  .long 1

  .even
TABLE:  .skip 1000
```

Note that the `.even` is added in the MC68K world to ensure that all accesses to XVAR and TABLE are legal (4 byte boundary).

4. **.DATA and .TEXT directives**

These are used to specify the type of the instructions that follows after the directive.

```plaintext
in the RISC world,
  (.DATA)
  (LABEL AUXR1 .WORD 0)

  (.TEXT)
  (NOP)
```

is translated into

```plaintext
in the MC68K world,
  .data
  .even
```
AUXR1: .long 0

.text
nop

5.1.2 Translation of RISC registers

It is characteristic of RISC instructions to mainly work with registers. The Motorola 68020 micro processor has only a limited amount of registers, as a result, the back end simulates a RISC 32-bit register by assigning 4 bytes of the MC68K world data memory to represent a RISC register. Everytime a register is used in the RISC world, the corresponding 4 bytes in the MC68K world data memory is cached into a MC68020 register. In order for the back end to recognize a RISC register, we also created the rule that all RISC registers must be declared in the intermediate code. For example;

in the RISC world,
(DATA)
(LABEL FP .WORD 0)

is a register that is translated into

in the MC68K world,
.data
.even
FP: .long 0

The declaration of all RISC registers is done in the RISC runtime code. The runtime is presented shortly.

Let’s now look at an example that actually uses RISC registers:

in the RISC world,
(DATA)
(LABEL FP .WORD 0)
(LABEL AUXR1 .WORD 0)
(TEXT)
(LW FP 1 AUXR1) ;; [[[AUXR1] + 1] --> FP
would be translated by the back end into:

in the MC68K world,

.data

.even

FP: .long 0

.even

AUXR1: .long 0

.text
movl AUXR1,a0 ; [AUXR1] ---> a0
movl a0@$(1),FP ; [(a0) + 1] ---> FP

Note how the AUXR1 is cached into the MC68020 address register a0.

5.1.3 Translation of RISC instructions

Each RISC instruction is translated into one or more MC68020 instructions. An example:

in the RISC world,

(ADD RD RS RT)

is translated into

in the MC68K world,

;; RS, RT and RD
;; are MC68K registers
movl RS,d0 ; [RS] ---> d0
addl RT,d0 ; [RT] + [d0] ---> d0
movl d0,RD ; [d0] ---> RD

Addressing mode used by the back end translator are:

- An operand field of an means an Address-register.
• An operand field of \textit{dn} means a Data-register.

• An operand field of \textit{ru} means an Address- or Data-register.

• An operand field of \textit{ea} means an effective address designated by one of the above permissible addressing modes.

• An operand field of \textit{\#data} means an immediate operand.

The MC68020 instruction set used by the back end is shown in table 5.1. and 5.2. Table 5.1 and 5.2 may not be sufficient for future development of the VonMon project and hence, for further information on the instruction set and addressing modes, please refer to the SUN-III Assembly Reference Manual [1].

The translation operation of all the RISC instructions in the list (the intermediate code) is done iteratively by the back end. The basic algorithm of the operation is as follows:

```
open assembly file

START:   end of list? --yes-- finish
else
{  
   process label identifier (if any) of present instruction

   translate present instruction into the assembly code

   cdr down the list

   go to START
}

FINISH:  close assembly file
```

The translation of the present instruction in the above algorithm uses an opcode lookup table to identify the RISC instruction. Once identified, the appropriate MC68020 instructions are written into the assembly file.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation Name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>add binary</td>
<td>addl ea,dn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addl dn,ea</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addl ea,an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addl #data,ea</td>
</tr>
<tr>
<td>addql</td>
<td>add quick</td>
<td>addql #data,ea</td>
</tr>
<tr>
<td>andl</td>
<td>logical and</td>
<td>andl ea,dn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>andl dn,ea</td>
</tr>
<tr>
<td></td>
<td></td>
<td>andl #data,ea</td>
</tr>
<tr>
<td>beql</td>
<td>branch on equal</td>
<td>beql ea</td>
</tr>
<tr>
<td>bgel</td>
<td>branch greater or equal</td>
<td>bgel ea</td>
</tr>
<tr>
<td>bgtl</td>
<td>branch greater than</td>
<td>bgtl ea</td>
</tr>
<tr>
<td>bnel</td>
<td>branch not equal</td>
<td>bnel ea</td>
</tr>
<tr>
<td>bral</td>
<td>branch always</td>
<td>bral ea</td>
</tr>
<tr>
<td>clr1</td>
<td>clear an operand</td>
<td>clr1 ea</td>
</tr>
<tr>
<td>cmpl</td>
<td>arithmetic compare</td>
<td>cmpl ea,dn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cmpl #data,ea</td>
</tr>
<tr>
<td>extw</td>
<td>sign extend</td>
<td>ext dn</td>
</tr>
<tr>
<td>extl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>jump</td>
<td>jmp ea</td>
</tr>
<tr>
<td>jge</td>
<td>jump greater or equal</td>
<td>jge ea</td>
</tr>
<tr>
<td>jgt</td>
<td>jump greater than</td>
<td>jgt ea</td>
</tr>
<tr>
<td>jlt</td>
<td>jump less than</td>
<td>jlt ea</td>
</tr>
<tr>
<td>jbsr</td>
<td>jump to subroutine</td>
<td>jbsr ea</td>
</tr>
<tr>
<td>lea</td>
<td>load effective address</td>
<td>lea ea,an</td>
</tr>
<tr>
<td>movl</td>
<td>move data</td>
<td>movl ea,ea</td>
</tr>
<tr>
<td>muls</td>
<td>signed multiply</td>
<td>muls ea,dn</td>
</tr>
<tr>
<td>negl</td>
<td>negate binary</td>
<td>negl ea</td>
</tr>
<tr>
<td>orl</td>
<td>inclusive or</td>
<td>orl ea,dn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>orl dn,ea</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or #data,ea</td>
</tr>
</tbody>
</table>

Table 5.1: MC68020 instruction set.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation Name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>pea</td>
<td>push effective address</td>
<td>pea ea</td>
</tr>
<tr>
<td>rts</td>
<td>return from subroutine</td>
<td>rts #n</td>
</tr>
<tr>
<td>subl</td>
<td>arithmetic subtract</td>
<td>subl ea,dn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subl dn,ea</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subl ea,an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subl #data,ea</td>
</tr>
<tr>
<td>link</td>
<td>link and allocate</td>
<td>link an,#disp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>linkl an,#disp</td>
</tr>
<tr>
<td>unlk</td>
<td>unlink</td>
<td>unlk an</td>
</tr>
</tbody>
</table>

Table 5.2: continuation: MC68020 instruction set.

5.2 Producing MC68020 bootstrap and exit code.

In order to execute the assembly file produced by the back end, a C program (Appendix A) is compiled and linked with the assembly file. At run time, the C program will prompt the user for the argument. The program then passes the argument to an MC68020 bootstrap (entry) subroutine in the assembly file.

A complete picture is shown in figure 5.1 to illustrate how the bootstrap begins and the program terminates.

The bootstrap (entry) subroutine (Appendix A) is hand coded MC68020 code and is appended to beginning of the assembly file by the back end. This subroutine saves the argument off the stack into a MC68K register and then jumps to the initialization subroutine in the runtime code. After initialization, the execution of the MC68K world ETS program is performed. Upon completion of the program, control is passed to the exit code.

The exit subroutine (Appendix A), similar to the entry subroutine, is hand coded MC68020 code and is appended to the end of the assembly file by the back end. The exit subroutine prints out the high signed 32-bits and low signed 32-bits of the result and returns to the C program.
Figure 5.1: bootstrap (entry) and exit subroutine
Chapter 6

Conclusion

The VonMon project has explored the possibility of creating a dataflow machine on a Von Neumann machine. However, the development of the VonMon project at this point limits the translation of high level ID programs to code that does not involve I-structures. In order to execute ID programs that uses I-structures on the SUN-III workstation;

1) ETS instructions that involve I-structures have to be translated by the front end into RISC code. The instructions are I-structure related instructions.

2) more runtime subroutines that involve the I-structures need to be written. These subroutines perform allocation of I-structure locations, I-fetches, I-stores and garbage collection.

3) memory needs to be allocated for the heap memory. Once the heap memory is available, the frames and the token queue can be allocated in the heap.

6.1 Optimizations and Improvements.

The present version of the translator is the first and hence, much optimizations and improvements can be done to run the translated ETS program faster. The suggestions here are aimed at the back end.

i) the .even directive in the MC68K world can be eliminated for labels that are never jumped to. This can be achieved by rewriting the back end to keep track of jump instructions and their destinations. If the .even is eliminated, the instruction before the label need not be a jump instruction to that label.
ii) assign MC68020 microprocessor's registers to the IP, FP, RV1HI, RV1LO, RV2HI, RV2LO, PORT or any other RISC world registers that are often used. In order to do this, the SUN-III reference manual [1] or other pertinent literature should be researched to locate unreserved and reserved registers.

iii) implement a multiple-argument version of the translator. Presently, the C program prompts the user for only one argument. In order for the translated ETS program to receive more than an argument, an argument chain needs to be created. Further research on this should be done by consulting members of the Computation Structures Group.
Appendix A

Interactive C program, bootstrap and exit code

A.1 Interactive C program

main()
{
    int y;

    printf("please give an argument to your program -->");
    scanf("%d", &y);

    ets(y); /* this is the call to the mc68020 bootstrap code */

    printf("program executed\n");
}

A.2 MC68020 Bootstrap code

.data
.text
.globl _ets
.even

_ets: link a6,#0

movl a6@8, ARGUMENT_FROM_OUTSIDE_WORLD

jmp CREATE_FRAMEBUFFERHEADERS

A.3 MC68020 Exit code

.data1

.even

MSG: .ascii "the high result is %d and the lo result is %d\12\0"

.text

.even

PRINT_RESULT:

movl RESULT_LO_REG, sp@

movl RESULT_HI_REG, sp@

pea MSG

jbsr _printf

addl #12, sp

unlink a6

rts

.data1

.even

ERRMSG:

.ascii "QUEUE is full \12\0"

.text

.even
PRINT_QFULL_ERR:
pea ERRMSG
jbsr _printf
addql #4,sp
unlk a6
rts

A.4 Calling the MC68020 bootstrap code from the interactive C program

In order to call a MC68020 subroutine, we need to understand the C calling convention used on the SUN-III workstation.

In the C program,

ets(y); /* this is the call to the mc68020 bootstrap code */

calls the MC68020 _ets subroutine.

The ets(1) call is compiled into:

```
movl #0x00000001,sp0- ;put argument on stack
 ; ; [sp] - 4 --> sp
 ; ; 1 --> [sp]

jbsr _ets ;;put return address on stack
 ; ; and jump to _ets
 ; ; [sp] - 4 --> sp
 ; ; ret --> [sp]
 ; ; _ets --> pc

ret: addqw #0x4,sp
```
The _ets subroutine is shown below:

_ets: link a6,#0 ;; [sp] - 4 --> sp
      ;; a6 --> [sp]
      ;; [sp] --> a6

movl a60(8),ARGUMENT_FROM_OUTSIDE_WORLD

jmp CREATE_FRAMEBUFFERHEADERS
Bibliography


