PROGRAM DEVELOPMENT ENVIRONMENT FOR TWINE-RISC

by

SHYAM SUNDAR PALREDDY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March 1994

PROGRAM DEVELOPMENT ENVIRONMENT FOR TWINE-RISC

A thesis submitted
in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Shyam Sundar Palreddy

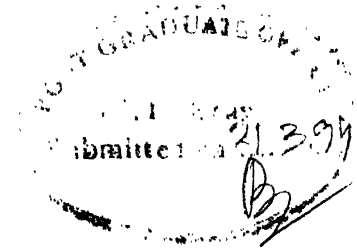
to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

March, 1994

CSE-1994-M-PAL-PRO



CERTIFICATE

It is certified that the work contained in the thesis entitled *Program Development Environment for Twine-RISC*, by *Shyam Sundar P* has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sanjeev Kumar Aggarwal,
Assistant Professor,
Department of Computer Science
and Engineering,
IIT Kanpur.

Dr. Rajat Moona,
Assistant Professor,
Department of Computer Science
and Engineering,
IIT Kanpur.

March, 1994

To My Parents.

ACKNOWLEDGEMENTS

I am very grateful to my guides, Dr RAJAT MOONA and Dr. SANJEEV KUMAR AGGARWAL, for offering constant guidance throughout my thesis work.

I thank bikka, krishna, malhal, sai, reddy, srenu, anand, vaidya, venkat, chaitanya, PL and all other friends and classmates who made my stay at IITK enjoyable.

I thank my wingmates murali (with narsim and gcp), partha, bairagi, sitharam who were very helpful in time of need.

I would like to thank my friends vijji, srenu, kittireddy and pottireddy who kept up my spirits, by constantly being in touch.

Lastly but not the least I thank all the swayamsevaks (especially madhusudhanji, pradeep, santoshji, satyamji, rajeshji, girishji) who made my stay in kanpur memorable. The impressions created by madhusudhanji and pradeepji are going to stay for my lifetime. The baudhiks of dupadji with his astonishing insight were very knowledgeable.

27-3-94 Shyam Sundar P.

Abstract

Twine-RISC is a novel single-chip, low-cost processor architecture which exploits the instruction-level temporal parallelism by its well engineered RISC pipeline, and spatial parallelism by allowing multiple threads of computation to co-exist and execute in parallel. This architecture is a hybrid of Von Neumann and Dataflow architectures. The aim of this thesis was to develop software support for Twine-RISC.

In this thesis we have developed a Macro-assembler, Linker and a Simulator for this architecture. The Twine-RISC processor allows execution of multiple instructions per clock cycle. The number of instructions that can be executed per cycle is equal to the number of Twine-RISC Streams in that processor. This architecture has been simulated on a Von Neumann machine, which does not allow simultaneous execution of instructions.

Contents

List of Figures

Chapter 1

Introduction

Twine-RISC is a low cost single chip processor architecture which exploits instruction level parallelism by its well engineered RISC pipeline and spatial parallelism by allowing multiple threads of computation to co-exist and execute in parallel. It (Twine-RISC) is a novel design which captures the concept of dataflow and Von Neumann architectures. The concept of Twine-RISC is still in research stage and hopefully after a through performance study may become the processor of the future.

1.1 Evolution of Twine-RISC

RISC architectures [PS82] have been derived from the conventional von-Neumann architecture. These architectures are widely used in many of the present day commercial computers. RISC instructions are simple, regular and are usually based on three operands. However, in RISC, the inter-dependence of instructions due to the stored program concept of von Neumann computers has been a major bottleneck in the parallel execution of programs.

Dataflow architectures [AC86] offer a possible solution for efficiently exploiting concurrency of computation on a large scale. The computing nodes are fired when data arrives and the execution of instructions may not be in the sequence in which they are stored in the memory of a computer [AN89]. However, no existing architecture supports efficient execution of dataflow programs.

Nikhil and Arvind [NA89] proposed *P-RISC*, which combines the ideas of both von-Neumann and dataflow computing. In P-RISC, the program counter(PC), found in von-

Neumann computers is eliminated and multiple threads of computation is achieved through the execution of tokens, a concept borrowed from dataflow computing [AC86, AN89]. The RISC feature of pipelined instruction execution is also effectively utilized. However, in a single processor, multiple threads cannot be simultaneously executed.

Moona, Nandy and Rajaraman [MNR] have proposed a novel architecture called **Twine-RISC** which supports execution of multiple threads in a single processor. *Twine-RISC* has eliminated many drawbacks which are existing in P-RISC and efficiently exploits the fine-grain parallelism which is inherent in most programs.

Dhiren Patel [DH92] had developed a simulator from which he was able to propose some modifications to the original architecture [MNR]. He has also concluded that Twine-RISC is able to fulfill its goals of executing dataflow graphs efficiently with economical architectural frame work.

Dinesh Rao [DIN93] had proposed a simple hardware design for Twine-RISC. In the design proposed by him he had shown two Twine-RISC Streams and had justified it by mentioning the complexities involved as number of Twine-RISC Streams increased. He had also suggested as future work to develop software support like Compiler, Assembler, Loader etc. for the Twine-RISC.

1.2 Twine-RISC architecture

1.2.1 Introduction

In this section, we briefly discuss the processor architecture of *Twine-RISC*. A detailed description of Twine-RISC is available in [DIN93, DH92]

Twine-RISC is a processor which combines the advantages of von-Neumann and dataflow architectures. Multiple threads can be efficiently executed in *Twine-RISC*. The multiple RISC pipelines in *Twine-RISC* facilitate simultaneous execution of multiple threads, thus effectively exploiting the instruction level parallelism. *Twine-RISC* supports *split-phase* transactions between the global memory and the processor through the message processor. All the units of the TRS (described later) operate asynchronously and a handshaking unit is present between each pair of interfaced blocks of a TRS.

Fig. 1.1 illustrates the processor architecture of *Twine-RISC*. In this figure we have shown one Twine-RISC Stream (TRS) completely and the block of second one. There can

be any number of such TRSs as permitted by state of art VLSI Technology.

Now we will describe the functionality of different units of Twine-RISC.

1.2.2 Operand Memory

The Operand Memory (OM) concept in *Twine-RISC* is similar to the register file of conventional RISC processors. It consists of 64 registers of 32-bits each. The OM is shared by all TRSs. The OM is a multi-port memory structure to cater to the demand of multiple TRSs. The read ports are utilized by the OFUs of the TRSs and the write port is used by the RSUs.

1.2.3 Code Memory

The Coce Memory (CM) is common to all TRSs and is positioned outside the chip. It holds the instructions and is read-only for the TRSs. A separate host processor is used to initialize the CM.

1.2.4 Token Queue

The continuation tokens for the TRSs are stored in the Token Queue (TQ). A continuation token consists of two pointers, namely the frame pointer (FP) and the instruction pointer (IP). The IP points to the position of the instruction to be executed in the CM and the FP is a base pointer to the data in the OM for a code block. Multiple active invocations of the same code block are possible by the use of frame-relative addressing. A continuation token can be utilized by any TRS. The TQ is initially loaded by the *host processor* through the Sequencer and subsequently, the continuation tokens are supplied by the TRSs. Host can also insert tokens later during the program run to make *Twine-RISC* execute threads asynchronously and thereby handle asynchronous events.

1.2.5 Sequencer

The sequencer serializes the continuation tokens generated by TRSs, MP and Host processor. It sends them one after the other to the TQ. All the tokens present in the TQ are independent of one another and the sequence in which these tokens are stored in the TQ is irrelevant.

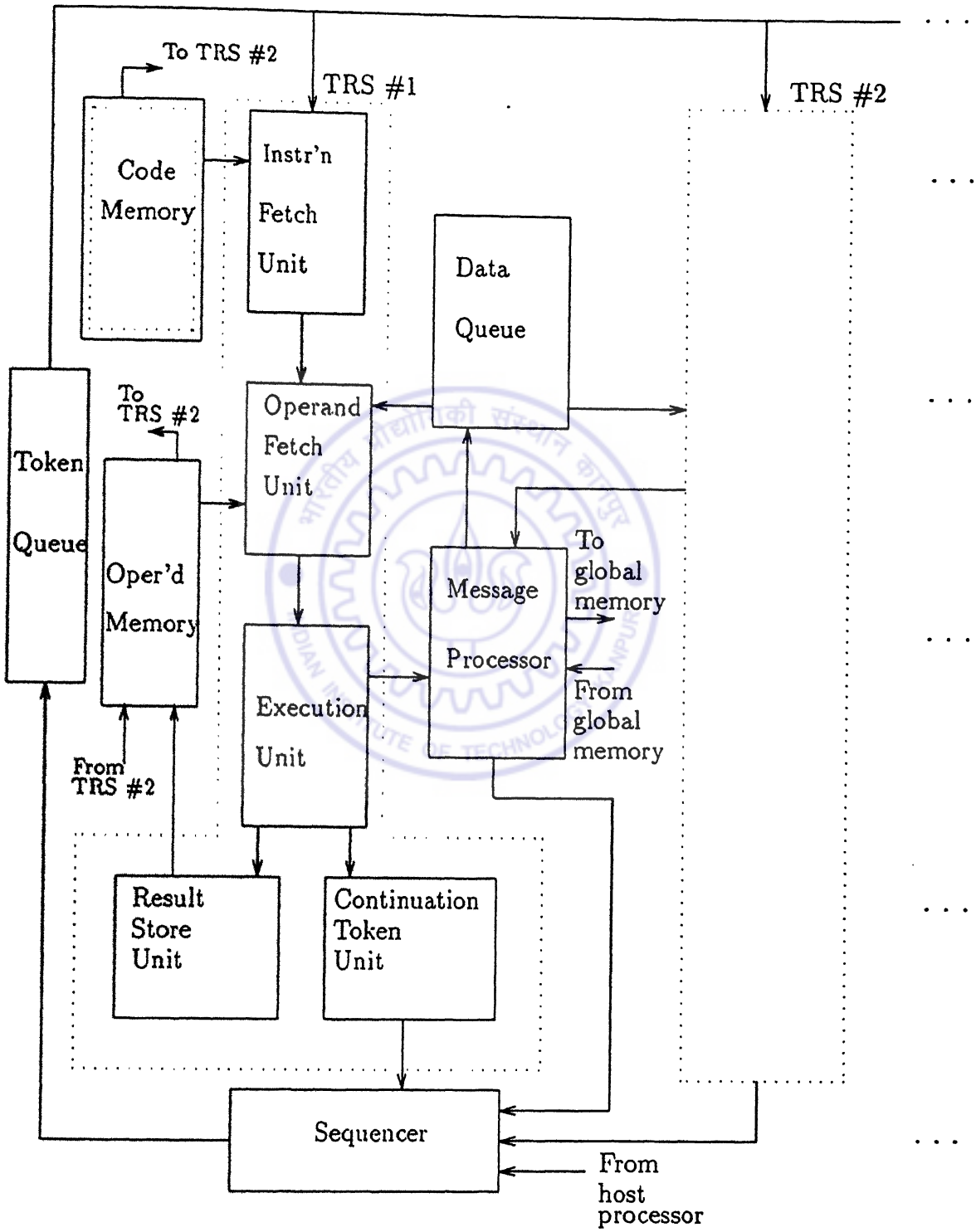


Figure 1.1: Architecture of Twine-RISC

1.2.6 Data Queue

The Data queue is similar to TQ and is used to store data coming from the global memory in response to LOAD/LOADX through the Message Processor. These data are written into the DQ and a thread to execute the instruction RESM is added to the TQ. When RESM is executed, data is finally moved from the DQ to the OM and the thread is reintiated.

1.2.7 Message Processor

MP takes care of the movement of data between the TRSs and the global memory. In case of a read request, the MP receives a response from the global memory controller containing a value, Operand Memory address and continuation token. The MP writes data into DQ and generates a continuation token (0,0) to be stored in the TQ. The MP also directs the LOAD/STORE requests to the global memory. The EXU of the RISC pipeline sends 78 bits data to the MP consisting of 1-bit Read, 32-bit address, 32-bit IP, 6-bit Destination Register (DR) and 1-bit request. The MP receives a similar message (77-bits) from the global memory controller without the request bit and sends it to the DQ.

1.2.8 Instruction Fetch Unit

The Instruction Fetch Unit (IFU) fetches continuation token from TQ and fetches the next instruction from CM (using IP). It also partially decodes the fetched instruction to determine weather the next instruction is to be fetched from the next CM location or not. It also detects the MJOIN instruction and sets the mjoin-lock to enable the atomic execution of MJOIN instruction.

1.2.9 Operand Fetch Unit

This TRS block decodes instructions partially by using three bits of opcode and decides the number of operands to be fetched from the OM. This unit also detects the RESM instruction and if so, fetches operands from the DQ. It then routes the instruction, operands, FP and IP to the EXU.

1.2.10 Execution Unit

It is similar to the ALU of a conventional processor. It also prepares continuation tokens (FP,IP) for branch and other special instructions for thread initiation and forwards it to the Sequencer through the buffer B3. After executing the arithmetic and logical instructions, it sends the result and the address of the destination register to the RSU. Requests for memory read/write are sent to the MP.

1.2.11 Result Store Unit

This is the only stage that can write to the Operand Memory(OM). It writes the value of the result generated by the EXU in the destination register . It also releases the MJOIN *lock* line set by the IFU.

1.3 Motivation

An ideal way to evaluate Twine-RISC would be to develop a compiler that identifies the parallelism in programs and code it using *mfork* and *mjoin* instructions. The machine code thus obtained should thus obtained should be made to run on Twine-RISC and thus obtain the performance metrics. Another way could be to develop programs using dataflow language which generates machine code for Twine-RISC. In either case we can think of the compilers generating an assembly code rather than the machine code directly.

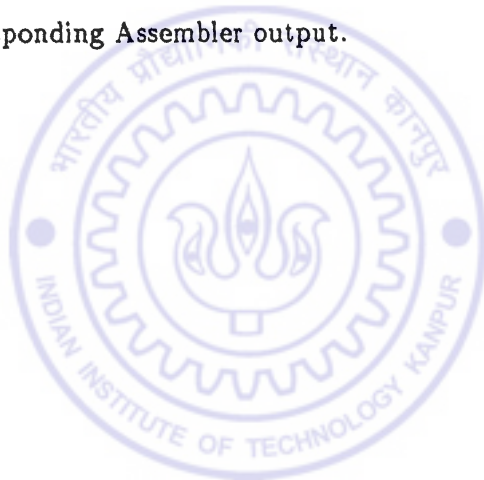
Now is the need of an assembler which generates the object code for Twine-RISC, a linker which links the object code modules generated by assembler into an executable code. In order to run the executable code we need a simulator which not only runs the program but also provides some debugging facilities for program development.

In this thesis we have developed a one-pass macro assembler and a linker using which the assembly level programs of Twine-RISC can be assembled & linked to obtain an executable code. We have also developed a simulator which takes the executable code for Twine-RISC and simulates its execution on the proposed architecture. Apart from this the simulator also provides debugging tools like step execution, breakpoints, tracing etc. It also provides some performance metrics related to Twine-RISC architecture.

1.4 Conclusion

In the previous section we have described *Twine RISC*, a novel processor architecture. It has a simple pipeline structure and a modular design, which helps designing each of the blocks in a systematic way with minimum overheads.

The rest of the thesis is organized as follows: In chapters 2, 3 and 4, we will discuss implementation details of *Assembler*, *Linker* and *Simulator* respectively. In chapter 5, we make concluding remarks. Appendix A gives the user manual for *Assembler*. Appendix B gives the detailed instruction set of *Twine RISC*. Appendix C gives the user manual for *Linker*. Appendix D gives the user manual for *Simulator*. In appendix E has an example program and the corresponding Assembler output.



Chapter 2

Implementation of the Assembler

2.1 Introduction

Assembler takes input from an ASCII file containing the assembly language program for *Twine RISC*, the syntax of which is given in Appendix A. The input is parsed and if it is found to be free of syntax errors then an object file is created (which contains object code suitable for linking). The format of the output file is "Common Object File Format" (COFF) [FFM88].

We used *lex* and *yacc* [UUT88] tools for parsing of input file. Ours is a single pass assembler, and we resolve forward references by backpatching the generated code [DHM84]. The assembler is developed in C and tested on Sun machines.

In this chapter we discuss the implementation of assembler. Our assembler is a macro assembler and does macro expansion with resolution of local variables. To simplify the assembly programming, some macros are predefined and discussed in this chapter.

2.2 Macro processing

We could not use standard macro processors like *cpp* because they do not allow the local labels in macro, which we needed. Our macro processor allows local labels in macros which are mapped to a unique label, each time the macro is expanded. We do not permit hierarchical macro definition wherein, a macro can be defined within another macro definition.

2.2.1 Macro table

Macro table is used to store the information pertaining to a macro definition. It is organized as a tree. Its definition is given below.

Macro table node definition

```
typedef struct macro_table
    {
        struct macro_table  *lptr,*rptr;
        char name[MAX_VAR_LEN];
        int no_of_parms;
        int count;
        char  *macro_str;
    } macro_table_node;
```

"lptr" and "rptr" are two pointers to the sub-trees of the node. "name" is used to store the name of the macro, its length should be less than 50 (MAX_VAR_LEN). "no_of_parms" is the number of parameters this macro has. "count" is used to store a unique number for the macro which will be useful for generating unique names for local labels (explained later). "macro_str" is to store the processed body (explained later) of the macro. During processing the body of the macro is written in an array (of size 4000), and at the end it is copied into "macro_str" by allocating the required memory. So the limit on the maximum length of the macro definition is 4000 bytes.

2.2.2 Local labels in macro

Local labels are declared by using local directive in the variable declaration (see appendix A). All other labels are assumed to be global. Whenever a macro is expanded global labels are reproduced without any modifications. However the local names are prefixed and suffixed to generate unique names every time. This way, the name clash is avoided across macro expansions.

The method we chose to generate unique label names is as follows. During macro definition all instances (usages and definitions), of local labels are prefixed with a '#' character, followed by the name of the macro and '_' character. If a label starts with '#' (which will be the case if a macro is called in another macro) then an extra '#' character is not prefixed and only the name of the current macro followed by '_' character is added.

During macro expansion all the local labels (i.e. all the words which start with '#' character) are suffixed with '_' followed by a unique count in the macro table entry of that macro. This count is incremented after each macro expansion.

Thus if a local label is redefined in a macro, its name after the expansion will also be same and will lead to error.

2.2.3 Macro definition

If the assembler sees `MACRO` or `macro` (see appendix A), then it assumes that as start of the Macro definition. The information about the macro is stored in the macro table and the body of the macro is processed as follows. Formal parameters in the body of the macro are replaced by '&' character followed by number of the parameter (for example third parameter will be replaced with "&3"). Local labels are processed as explained earlier. The body processed this way is stored in the macro table to be used at the macro expansion time.

2.2.4 Macro call

When a macro call is detected the assembler needs to get the actual parameters of the call before its expansion. These parameters should be replaced verbatim for the formal parameters. However, the specifications of the assembler cause reduction of the expressions. To disallow this we take the input directly and store the actual parameters. The actual parameters thus obtained are stored in an array and they will be used for expansion of the macro. After the actual parameters have been stored in an array the control is passed to macro expansion routine.

2.2.5 Macro expansion

While expanding the macro we take the body of the macro from the macro table, and the actual parameters which have been stored in an array (when a macro call is found) as said earlier. The body of the macro is scanned for parameter replacement. As said earlier, the formal parameters of the macro are replaced by their number prefixed by '&', it is easy to find these strings and replace with corresponding actual parameters. For example, if "&3" is found then it is replaced with the third actual parameter. The local variables are suffixed

by a unique integer as described earlier. The macro expansion takes place in a buffer which is passed to the assembler.

2.3 Symbol table

Symbol table is used to store identifiers and labels. It is organized as a hash table [ASU86].

The Hash function [ASU86] which we used is as follows (in C language),

```
hashpjw(s)
char *s;
{
    char *p;
    unsigned int  h=0,g;

    for( p = s; *p != '\0'; p = p+1 )
        {
        h = (h << 4) + (*p);
        if ( g = h & 0xf0000000)
            {
            h = h ^ (g >> 24 );
            h = h ^ g;
            }
        }

    return( h % HASH_TABLE_LENGTH);

}/* end hashpjw() */
```

In the above function the HASH_TABLE_LENGTH is the size of the hash table. It is defined as constant (211). If it is prime number then the performance of this algorithm will be good. The data structure for symbol table is shown in Fig. 2.1 and explained below.

- Hash table is a fixed size array of *buckets* [ASU86]. Each bucket is a doubly linked list. Elements in the bucket are symbol table entries. Some of the buckets may be empty.

- Each entry in the symbol table appears on exactly one of these buckets. Storage for the entries is drawn from an array of structures of type *Symbol table node* (whose declaration is given below).

Symbol table node declaration

```
typedef struct symbol_table
    {
        struct syment  sym_ent;
        struct symbol_table  *lptr,*rptr;
        int sym_index;
    } symbol_table_node;
```

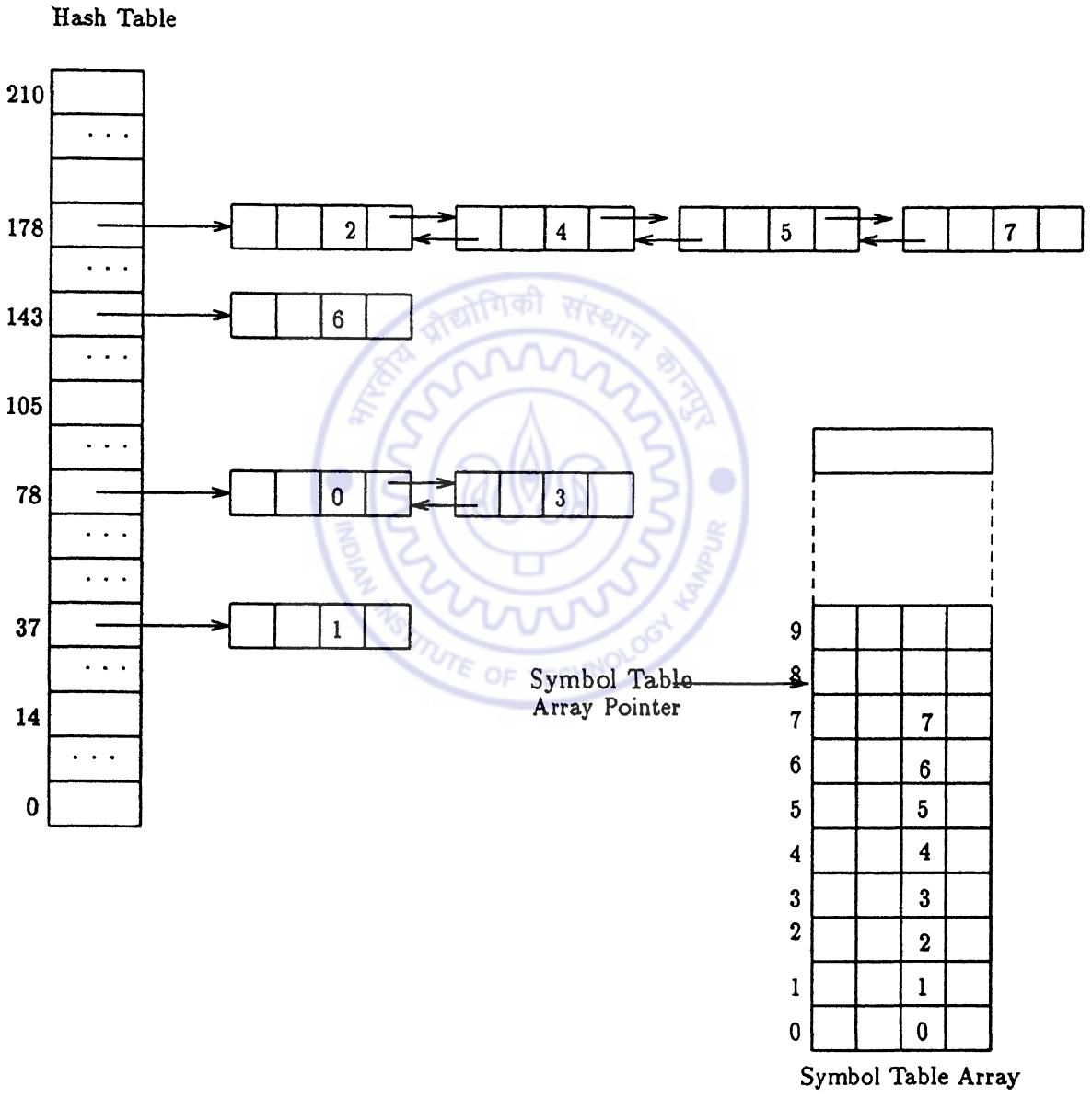


Figure 2.1: Data structure of symbol table

The declaration for a symbol entry *struct syment* is the one given in *syms.h*, the symbol entry declaration file of COFF[FFM88]. "lptr" and "rptr" are used to store the pointers to nodes to the left and right of this node in the doubly linked list. "sym_index" is used to store the symbol table index of the node.

2.4 Mnemonic table

Mnemonic table is used to store mnemonic, its opcode and the number of operands required (definition of its node is given below). It is organized as a tree. Whenever an instruction mnemonic is found in the input, this table is accessed to get the associated information needed for generating machine code.

Mnemonic Table Node Definition

```
typedef struct  mnemonic_table
    {
     struct mnemonic_table  *lptr,*rptr;
     char name[MAX_MNE_LEN];
     int code;
     int max_ops;
    } mnemonic_table_node;
```

"lptr" and "rptr" are used to store the sub-trees of the node. "name" is to store the name of the instruction mnemonic. "code" is to store the opcode of the instruction. "max_ops" is the number of operands expected for this instruction.

2.5 Input

As said earlier we used *lex* and *yacc* for input parsing. *lex* automatically generates a procedure called *input()*, for getting input. Controlling the source of input is necessary when a macro call is given. It has to be changed from the input file to internal buffer (containing the expanded macro). For this we have to define our own *input* function.

2.6 Generating code

During parsing, the operands of the instruction are stored in an array. On reaching the new line character the assembler will generate code for the instruction using information in operand array and mnemonic table. A relocation entry will be added if the instruction has

a forward reference (for backpaching), or an external reference (for linker processing), or an absolute address reference (for example, unconditional jumps, to be processed by linker). If -l option (for listing) is given to the assembler then listing of the instruction is written (see Appendix A). If -g option (for debugger information) is given then line number entry is made to the instruction.

2.7 Relocation and line number entries

2.7.1 Relocation entries

Relocation entries are stored in an array. Each element of this array is a structure of type reloc entry as defined in COFF [FFM88] format and given in "reloc.h".

The definition of structure is reproduced here.

```
struct reloc {
        long    r_vaddr;    /* (virtual) address of reference */
        long    r_symndx;   /* index into symbol table */
        unsigned short  r_type;     /* relocation type */
    };
```

"r_vaddr" is to store the address of the instruction for which that relocation entry is written. "r_symndx" is to store the symbol Table index of the identifier which has to be used for relocation. "r_type" is to store the type of the relocation that needs to be done.

2.7.2 Line number entries

Line number entries are added to the object file if -g option is given to the assembler. These entries are used by debuggers. Line number entries are stored in an array. Each element of this array is of the following type (also given in "linenum.h" of COFF [FFM88]).

```
struct lineno
{
    union
    {
        long    l_symndx ;  /* sym. table index of function name
                        iff l_lnno == 0      */
        long    l_paddr ;   /* (physical) address of line number */
    }       l_addr ;
    unsigned short  l_lnno ;    /* line number */
} ;
```

If the value of "l_lnno" is zero then the field in union "l_addr" is taken to be "l_symndx" which contains the Symbol Table index of the ".file" entry [FFM88] of the file whose line number entries follow. If "l_lnno" is not zero, the value in it is taken to be line number of the instruction which is located at address "l_paddr".

2.8 Backpatching

In a single pass assembler if forward references are made then relocation entries are made for these, and zeros are filled for addresses/offsets in the code. At the end of parsing these holes are filled and corresponding relocation entries are removed. This process is called Backpatching [DHM84].

Since ours is a single pass assembler we need to do backpatching to resolve forward references. All the relocation entries for conditional jumps instructions for which backpatching is done (those for which jump address is defined in the same file), are removed. After backpatching relocation entries are removed except those which have external references.

2.9 Predefined macro MFORK

Twine RISC provides an instruction called *mfork* [DIN93] which can generate multiple threads (maximum five). This instruction takes two register operands one of which is be used for generating threads, and other is used to return the number of threads generated. The first register contains offsets (relative to mfork instruction) of the start of the threads which have to be generated. For loading this value into register the programmer has to use several assembler instructions. To help such a coding we provide a predefined macro MFORK to do the job of calculating offsets and initializing registers.

The MFORK macro takes two register operands and label (maximum four) operands. The label operands are the ones from which the new threads are to be started. When assembler comes across an MFORK macro it stores all these operands, current location counter value and current line number in an array. Now the location counter is incremented, to leave space for six instructions, which are going to be generated for each MFORK. After all the input has been parsed and no errors has been found then assembler expands the MFORK. Assembler calculates the offsets (relative to mfork instruction) of all the labels. The offset of each one is put into one byte, and if any of them overflows then an error

message is generated. The four offset bytes are packed into one four byte word. If less than four labels are given then most significant bytes of the word will be zeros.

For example, `MFORK r6,r7,start` where offset between *mfork* instruction and label "start" is 75 (0x4b) the following instructions are generated. be produced are as follows.

```
1   mvi  r6 ,  0x0
2   sftl   12 , r6 ,r6
3   mvi  r6 ,  0x0
4   sftl   12 , r6 ,r6
5   mvi  r6 ,  0x4b
6   mfork  r6 ,  r7
```

2.10 Conclusion

Our assembler supports all the standard features of an assembler. It is a single pass back-patching macro assembler and supports some predefined macros to simplify coding.

The user manual for the assembler is given in Appendix A. The instruction set of the Twine RISC is given in Appendix B.

Chapter 3

Implementation of the Linker

3.1 Introduction

Linker accepts object files and prepares an executable file for the Twine-RISC architecture or another object file suitable for further *Linker* processing (with -r option). The object modules on which link operates are specified on the command line. The Linker input files (object files) are expected to be in "Common Object File Format" (COFF) [FFM88], and the output of the linker (executable file) is also in COFF format.

In this chapter, we discuss the implementation of the *Linker*. The processing that needs to be done to combine multiple object files (each of which is in COFF format) is also discussed.

3.2 Processing input

All the input files of the linker are processed in the order they are given. Information is extracted from each file, and concatenated to the respective sections, i.e text sections of all the files are concatenated, and similarly data sections, relocation entries, symbol tables etc are concatenated. As a result of this concatenation the absolute addresses of instructions change. Therefore, we have to do relocate addresses in the locations where these absolute values have been changed. This we will discuss in section 3.4. We also have to update information in symbol table entries, relocation entries, etc. These are discussed in the following section.

3.2.1 Relocation entries

Relocation entries contain the information about those locations in the program, whose contents depend on the address at which the program is placed, or those which use symbols defined in other files. Relocation information is provided by the Assembler or Compiler. The Linker uses relocation information to correct these locations. This process is called *Relocation.* For each relocation that needs to be done in the object code, one Relocation entry is required.

In the previous chapter we have discussed the declaration of the relocation entry, and the information stored by fields in it. As said earlier the *r_vaddr* field is to store the address of the instruction for which that relocation entry is written, Since the address changes (after concatenation of files) we have to update it. The text offset (size of the text section before loading the current object file) is to be added to *r_vaddr*. Similarly the symbol table offset (number of entries in the symbol table before loading the current object file) is added to *r_symndx* field, which stores the symbol on basis of which this location has to be relocated.

3.2.2 Line number entries

Line Number Entries contain the information about the line number of the source code and the location of the corresponding instruction. This information is required for debuggers.

In the previous chapter we have seen the declaration of the Line Number Entry. The value in *l_lnno* is used as a flag for the union of the line number entry. If *l_lnno* field is zero then *l_symndx* field (in union *l_addr*) will contain the symbol table index of the ".file" entry of the file to which the following line number entries belong. If *l_lnno* is not zero then *l_paddr* field (in union *l_addr*) will contain the location (address) of the instruction generated from the source code line whose line number is contained in *l_lnno*.

3.2.3 Symbol table entries

The Symbol Table is used to store the information about the variables, constants and labels used in the assembly program. In the previous chapter we discussed the declaration of the Symbol Table Entry. In COFF [FFM88] format the symbol table is situated after the line number entries. Processing that needs to be done to the Symbol Table Entry due to concatenation is as follows.

If name of the symbol is stored in String Table (which is contained at the end of the file in COFF [FFM88] format), then the String Table offset (size of the String Table before loading the current object file) is added to the *n_offset* field of the entry. Now the storage class of the entry is tested, if it found to be External Declaration then it is stored in an array (later used for resolving cross references). If it is label then text offset is added to its *n_value* field. If it is any of the data types then data offset is added to that field. If the symbol is "_start" entry or the one given at *-e* option (and if it is of type label) then the value in its *n_value* field is taken as entry point for (the execution of) the resulting file. The ".file" entries in the Symbol Table are arranged as a linked list. All the symbols of an input file (object file) are stored in a separate subtree. This helps in resolving the cross references.

3.3 Resolving cross references

As said earlier while reading the Symbol Table all the symbols which are declared externally are stored in an array to be used for resolving cross references. For each of these symbols we search the symbol subtrees (as said earlier) to find if that symbol is defined in any of the subtrees other than the one to which that symbol belongs. if it is found we store a pointer to the defined symbol in symbol entry of the external symbol. If it is not found in any file then an error message is given, if -r option is not given.

3.4 Relocation

If all the cross references have been resolved successfully then the linker performs relocation addresses. For relocation entries corresponding to unconditional jumps the location of the instruction may get changed. Therefore, this will be relocated using the value of the symbol (whose index is stored in *r_symndx*). These relocation entries will be added to the output file because the loader might need them later. The relocation entries of conditional jumps will be of those which use labels defined in other files. These will be relocated and their relocation entries will be removed.

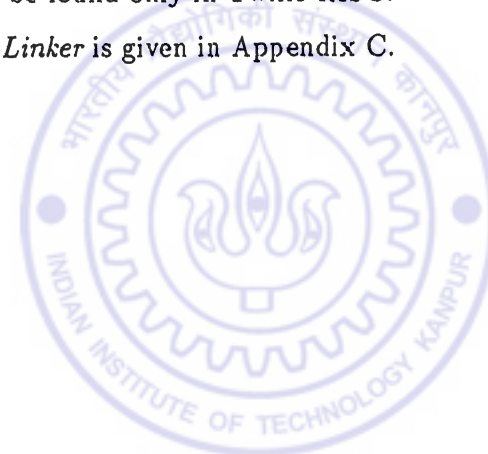
3.5 Output

After doing relocation the complete information is written to the specified (with -o option) file. This file is the executable file of the Twine RISC (if -r option not given). The output file will be in COFF [FFM88] format and can be executed on a Twine-RISC machine.

3.6 Conclusion

The linker for the Twine RISC is similar to the other linkers. It does not have any special features like in the assembler where we had to handle some special features (like special instructions) which can be found only in Twine RISC.

User manual of the *Linker* is given in Appendix C.



Chapter 4

Implementation of the Simulator

4.1 Introduction

Simulator takes input from an executable file of Twine-RISC machine, produced by the linker. It expects the file in COFF [FFM88] format and simulates the Twine-RISC [DIN93] architecture. The Simulator is written in C language, and runs under Sun OS.

Twine-RISC can execute more than one instruction in every cycle. The number of instructions it can execute is equal to the number of Twine RISC Streams (TRS) on the machine. Since the simulator itself runs on a Von Neumann machine we can not perform thhe simultaneous execution. However it can be simulated.

The number of TRSs present in the architecture being simulated can be specified with -t option of the simulator. The default number of streams is two.

4.2 Execution in each TRS

All Twine RISC Streams (TRS) fetch instructions from a common Code Memory (CM). Each TRS has its own Instruction Pointer (IP) and Frame Pointer (FP). Initially the IP of all the TRS is made to -10 and Frame Pointer to zero.

4.3 Queues in Twine RISC

There are two queues in the Twine RISC architecture [DIN93, DH92] Token Queue and Data Queue.

4.3.1 Token Queue

The continuation tokens for the TRSs are stored in Token Queue. A continuation token consists of two pointers, namely the instruction pointer (which gives the start address) and the frame pointer, of the threads which are waiting for execution. If any of the TRSs becomes free, the first token in the token queue is taken and execution of thread started. The token queue is implemented as a circular queue.

4.3.2 Data Queue

The data queue is used to store the data values. When a memory request is made to the global memory with LOAD/LOADX instructions the memory controller returns the data through the Message Processor. The Message Processor prepares data token (which contains the data value received) and stores it in the data queue. A continuation token with IP address zero (corresponding to RESM instruction) is pushed to the token queue. Data token contains the data value returned by memory controller, the register in which it has to be stored, the IP and the FP of the thread to be executed.

When a RESM instruction is executed, it takes first data token in the queue, stores the data in the register specified in the token, and adds a continuation token (with IP and FP as those given in data token) to token queue.

4.4 Message processor

When a memory based instruction (LOAD, LOADX, STORE & STOREX) is executed, the request (read or write) is sent to the message processor. Message Processor forwards this request to the memory controller. The memory controller can take multiple cycles to respond, so a thread which executes a memory read instruction (LOAD & LOADX) has to be stopped as it can't continue without that data. When the memory controller responds, the Message Processor adds a data token (containing the data just received) and a continuation token for RESM instruction (explained earlier).

The memory requests influence the performance of the system because the response time of the global memory could be more than one cycle and the thread will have to be stopped. As a result threads may some times be waiting for data and some of the TRSs may be free. This leads to under utilization of total computing power in the Twine-RISC.

The response time (for read request) of global memory of the machine to be simulated can be specified using -m option. The response time is specified in terms of the number of cycles. The number of cycles need not be the same (constant) for different (memory) requests. If the response time is same (constant) for all requests then *-mconstant <cycles>* is to be given where <cycles> is the number of cycles taken for each request. If time taken for different requests is not same then *-mvariable <min_cycles> <max_cycles>* is to be given, where the response time is between <min_cycles> and <max_cycles>. Simulator generates a random number for memory response time for each request which falls between <min_cycles> and <max_cycles>. If -m option is not given to the simulator then, the response time is taken to be constant and is five cycles.

For simulating the Message Processor we used a queue. Whenever a request is made to the memory we add a token to this queue with the number of cycles required to respond. After each cycle we decrement the cycles required for all entries of this queue. If the cycles required becomes zero for any of the entries then (as said earlier) the message processor pushes the data in this entry into data queue and places a continuation token (for RESM instruction) on the token queue.

4.5 Execution unit

As we have said earlier the Twine-RISC machine executes one instruction per cycle in each TRS. But as we are simulating it on a Von Neumann machine we can not execute these instructions simultaneously. The order of execution is from the first TRS to the last TRS. However this is transparent to the user.

Initially the execution starts with the continuation token which will be placed on the token queue by the system (possibly host machine). The IP (Instruction Pointer) of this continuation token will be the value given in the "entry" field of the optional header of the input file in COFF [FFM88] format.

Before execution of every cycle, we check to find if there are any continuation tokens on the token queue. If any of the TRS is free, and there are some tokens on token queue then a new thread (of the first continuation token on token queue) is loaded on the free TRS. The thread waiting in the queue can be loaded into any of the free TRS, it does not depend on the TRS on which it was executed before stopping. If one of the TRS is executing an

mjoin instruction (the instruction which is used for synchronization of multiple threads) then the Execution Unit puts an *mjoin-lock* and so no other TRS can execute the *mjoin* instruction in the same cycle. Thus ensuring the atomicity of *mjoin* instruction.

If the user specifies the number of steps to be executed (step execution), then after each cycle we check to find out if that many have been executed. Before executing each cycle the IPs of all the TRSs are checked to find if a breakpoint is set on any of these. If so then the execution of all TRSs is stopped, and control is returned to the user.

The execution ends if all the queues are empty and the IP address of any of the TRS crosses text section, this will be a successful execution. If one of the thread crosses the text section while some other thread is alive (is running in one of the TRS) or while any of the queue is non-empty, it means there is an error in thread management. Error message will be given in this case.

4.6 Measuring the performance

For the given input program the simulator gives a comparative performance of the simulated Twine-RISC (having more than one TRSs) with the Twine-RISC having only one TRS. Output also gives the percentage utilization of the Twine-RISC machine it is simulating. This gives how well the machine (which is being simulated) has been used. This value will be close to 100 if for most of the time all the TRSs were busy (i.e at any given point of time the number of threads is more than or equal to the number of TRS).

4.7 Important data structures

The three queues token queue, data queue, and message processor queue are organized as circular queues. The register file is implemented using an integer array of size 64. For storing IPs and FPs for each of the TRS we used two integer arrays.

4.8 Source line display

The 'S' command at Simulator prints the source program lines of the instructions which are going to be executed next in different TRSs. This is possible only if the source file is compiled with '-g' option. The information about the source lines is stored in line number

entries of the COFF [FFM88] format if '-g' option is given to the assembler. We use this information to print the source lines.

The line number entries are arranged in an increasing order of the address of the instruction. When the command to print source lines is given, for all the TRS which are alive (all TRSs which have a running thread in it) we take its IP address and search for the line number entry containing that address (using binary search algorithm). If we find an entry for this address then we search for line number entry which contains the symbol table index of the ".file" entry of the source code file (from which we can get the name of the source program file). We can find this entry by going backwards in the line number entries from the point where we found the entry of this address. As said in previous chapter the line number field ("l_lnno") of the ".file" entry will be zero, it can be identified by this.

In the above explained way we get the line number and the name of the file in which it is given. Now we open that file, extract the required line and display it.

4.9 Printing and loading of variables

Printing and Loading of variables (Printing the current value of an identifier or assigning a value to it) is very similar to that of printing source program line, described above. This also needs the line number entries.

For all the TRSs which have a thread running we search for the line number entry (as explained in previous section) and from there we get the symbol table index of the ".file" entry (in symbol table) of the file in which this variable may have been declared.

After getting the symbol table index of the ".file" entry, we search forwards in the symbol table to find the given variable. If it is found, print its value or assign a value to it, depending on what is requested.

For Printing the value of the variable we have to check its type to get it from the global memory. As we said in the previous chapter the "n_value" field of the symbol table entry stores a pointer to the data (in data section) if it is a variable and it contains the value itself if it is a constant (declared with '='). If it is a variable of type integer, then we have to print the four bytes from the pointer, and if it is of type byte then we have to print only one byte from the pointer and similarly for other types.

4.10 Code display

The Simulator displays the code it is executing, in the form of instruction and its operands. For this -c option has to be given to the simulator or "cs" command at the Simulator. It can be toggled in the Simulator by typing "c" command. it actually displays the code just before executing.

4.11 Breakpoint

The Simulator allows setting of a single breakpoint. Before executing every cycle all the TRS are checked to see if they are at breakpoint. If any of them is at breakpoint then the execution of that cycle is stopped and message displayed.

The breakpoint can be set at any point of execution in debug mode by giving "bs" command. It can be cleared by giving a "bc" command, and displayed with "bp" command.

4.12 Conclusion

Our Simulator does an evaluation of comparative performance of the simulated Twine RISC with the Twine-RISC (with single TRS). It takes all aspects of the Twine-RISC into consideration including the memory access delays. We have also provided a debugging interface for the Simulator even through it is not very sophisticated. This debugger can be improved by providing features like code viewing.

Simulation of the Twine RISC architecture is different from that of a Von Neumann architecture because it does parallel execution of instructions. User manual for Simulator is given in Appendix D.

Chapter 5

Conclusion

Twine-RISC is an architecture which uses the ideas of Von Neumann and Dataflow architectures. It allows multiple threads of computation to co-exist and execute in parallel.

Our main aim in this thesis has been to develop software support for Twine-RISC architecture. We have developed an Assembler, a Linker and a Simulator.

Our Assembler supports all the standard features and some extra features like MFORK to suite the needs of the Twine RISC architecture.

Our Linker does the relocation of the Twine-RISC object code to produces the executable code of Twine-RISC.

Our Simulator simulates the Twine RISC architecture on the machine on which it is run. It takes Twine-RISC machine executable files and executes them. Its interface can be used for writing software applications on TWINE-RISC. It provides some basic debugging facilities like step execution, breakpoint setting, displaying and loading of registers and variables, printing source code line, etc.

Our simulator also does a comparative performance evaluation of Twine-RISC having multiple TRS with that of the Twine-RISC with single TRS. The assumption that the Twine-RISC architecture with multiple streams can exploit the fine-grained parallelism of the application programs has been found to be true.

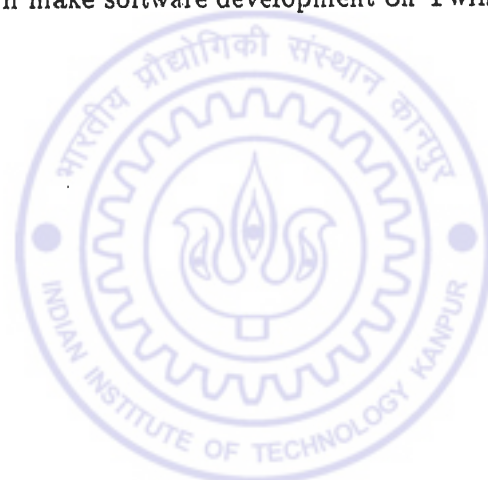
5.1 Scope for extensions

The following extensions will make the software development on Twine-RISC easier.

- The assembler can be improved by providing features like include files which can help making a library of common macros for different applications. Allowing macro definitions within a macro definition will make it more sophisticated.
- The interface of the Simulator can be improved by adding features like viewing of code and conditional breakpoints, displaying of variables, etc.

5.2 Scope for future work

A compiler for a language which can exploit parallelism in the programs has to be written for Twine-RISC. This will make software development on Twine-RISC easier.



Bibliography

[AC86] Arvind, D. Culler. *Dataflow Architectures*, Annual Reviews in Computer Science, Vol 1, Annual Reviews Inc., Palo Alto, CA, 1986. pp 225-253.

[AN89] Arvind, Rishiyur S. Nikhil. *Executing a Program on the MIT Tagged-Token Dataflow Architecture*, IEEE *Trans. Comp.*. 1989.

[DIN93] Dinesh Rao, B. *Design of Twine-RISC*, Master's Thesis, Department of Computer Science, IIT Kanpur, March 1993.

[DH92] Dhiren, Patel. *TWINE-RISC : ARCHITECTURE and PERFORMANCE EVALUATION STUDY*, Master's Thesis, Department of Computer Science, IIT Kanpur, March 1992.

[FFM88] Programers Manual SUN i386 : File Formats , SUN Inc. 1988.

[UUT88] Unix Utilities Manual , SUN Inc. 1988.

[ASU86] Aho, V. A, Ravi Sethi and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesely Publishing Company.

[DHM84] Dhamhadre, D. H. (1984). *Systems Programming*. Addison-Wesely Publishing Company.

[MNR] R.Moona, S.Nandy, V.Rajaraman. *Twine RISC: An Architecture for Simultaneous Execution of Multiple Threads*. [Personal Communication].

[NA89] R. Nikhil, Arvind. *Can Dataflow Subsume Von Neumann computing?*, Proc. 16th Int. Symp. On Computer Architecture, Jerusalem, Israel, June 1989. pp 262-272.

[PS82] David A. Patterson and Carlo H. Sequin, *A VLSI RISC*, IEEE Computer, Sept. 1982, pp.8-21.

Appendix A

Assembler User Manual

A.1 Description

Assembler takes input from an ASCII file containing the assembly language program for *Twine RISC*, and if the input is syntactically correct produces an object file suitable for linking.

A.2 Synopsis

```
asm  <input_file_name> ...   [ -o <output_file_name> ]  [ -g ]  [ -d ]
    [ -l <list_file_name> ]  [ -c <code_list_file_name> ]
```

NOTATION :

BNF.

A.3 Elements of assembly language

A.3.1 Character set

Assembler recognizes the following character set:

- The letters `A` through `Z` and `a` through `z`.
- The digits `0` through `9`.

- The ASCII graphic characters - the printing characters other than letters and digits.
- The ASCII non-graphic characters - space, tab, carriage return and newline.

A.3.2 Identifiers

Identifiers are used to tag assembler statements. For example labels or symbolic names of constants.

An identifier in an assembler program is a sequence of characters from the following set (with some restrictions).

- Upper case letters A through Z.
- Lower case letters a through z.
- Digits 0 through 9.
- The character underscore (_).

The restrictions on the Identifiers are as follows:

1. The first character of an identifier must be an alphabet.
2. All characters of an identifier are significant and are checked in comparisons with other identifiers.
3. Upper and lower case letters are distinct, so that no_of_items and NO_OF_ITEMS are two different identifiers.
4. The maximum number of characters allowed in an identifier is 50.

A.3.3 Labels

A label is an identifier followed by semi-colon. It represents the address of the location at which it has been defined. This can be used later for jump instructions etc.

A.3.4 Constants

The *Assembler* provides two kinds of constants. They are *numeric constants* and *string* constants.

Numeric constants

Assembler assumes that any token that starts with a digit is a numeric constant. Assembler accepts numeric quantities in decimal (base 10), hexadecimal (base 16), or octal (base 8) radices. The range of Numeric constants that is allowed is -2^{31} to $(2^{31} - 1)$.

All numbers that start with zero (0) are taken as octal constants and those which start with zero-ex (0x or 0X) are taken to be hexa-decimal constants. So decimal numbers can't be written with leading zeros. The hexadecimal digits consist of the decimal digits 0 through 9 and the hexadecimal digits `a` through `f` or `A` through `F`. Octal digits consist of decimal digits 0 through 7.

String constants

A string is a sequence of ASCII characters, enclosed in double quote signs ("). Within a string the double quote sign can not be used.

A.4 Expressions

This section gives operators which assembler provides and then gives the rules for forming expressions.

The Unary operators that are provided are:

Unary Operators in Expressions		
Operator	Function	Description
-	unary minus	Two's complement of its argument
~	logical	One's complement of its argument negation

The binary operators that are provided are:

Binary Operators in Expressions		
Operator	Function	Description
+	addition	Arithmetic addition of its arguments
-	subtraction	Arithmetic subtraction of its arguments
*	multiplication	Arithmetic multiplication of its arguments
/	division	Arithmetic division of its arguments (Integer division)

A.4.1 Syntax of expression

The syntax for Expression is :

```
expr :-    numeric_constant
         | identifier
         | expr op expr
         | u_op expr
         | ( expr )
```

where *op* is

op :- + | - | * | / | << | >> | & | % | '|'

and *u_op* is

u_op :- - | ~

In the above grammer *numerical_constant* is a decimal, hexa-decimal, or octal number. *identifier* is a variable defined in the program.

The operand precedence is (in descending order):

1. ~ , - (Unary Minus)
2. * , / , % , &
3. << , >>
4. + , - , |

A.5 Assembly language program layout

An assembler program consists of a series of lines. A line is a statement optionally followed by a comment, ending with a <newline> character. Blank lines are allowed. The form of the line is:

```
[ <statement> ] [ ; <comment> ]
```

One statement is to be written per line. The format of a statement is :

```
[ <labelfield> ] [ <opcode> [ <operandfield> ] ]
```

It is possible to have a statement which consists of only a label field. The fields of a statement can be separated by spaces or tabs. There must be at least one space or tab separating the opcode field from the operand field, but spaces are unnecessary elsewhere.

A.5.1 Label field

Labels are identifiers (explained earlier) which are used to tag the locations of program and data objects. The format of a `<label field>` is :

```
<identifier> : [ <identifier> : ] . . .
```

If present, a label always occurs first in a statement and must be terminated by a colon:

```
final:                    ; This is a label definition.
```

When a label definition is encountered in the program, the assembler assigns that label the value of the current location counter. The value of label is relocatable. The symbols absolute value is assigned when the program is linked with the Twine-RISC linker.

A.5.2 Operation code field

The operation code field of an assembly language statement identifies the statement as one of the following:

1. Machine instruction,
2. Macro call
3. Pre-defined macro
4. Assembler directive.

A machine instruction is indicated by an instruction mnemonic. The assembly language statement is intended to produce a single executable machine instruction. The operation of each instruction is described in appendix B. Conventions used in assembler for instruction mnemonics are also described in it.

A macro call expands the macro and places the expanded body of the macro at the current location.

A pre-defined macro is expanded at the current location. It produces many executable instructions. For example, MFORK (explained later) produces six instructions.

An assembler directive/pseudo-op, performs some function during the assembly process. It does not produce any executable code.

Assembler expects all instruction mnemonics in the op-code field to be in *lower case only.* The names of register operands must also be in *lower case only.* This behaviour differs from the case of identifiers, where both upper and lower case letters may be used and are considered distinct. Assembler directives can also be given in both the cases and taken as equivalent.

A.5.3 Operand field

The *operand field* of an assembly language statement supplies the arguments to the machine instruction, macro call, pre-defined macro or assembler directive. In general an operand field consists of zero or more operands, and in all cases, operands are separated by commas. In other words, the format of an <operand field> is :

```
<operand>   [  ,   <operand>   ]  . . .
```

More details about the operands of the instructions is given in Appendix B and for directives, macro calls & predefined macros is given later in the chapter.

The kind of objects which can form an operand are :

- Register Operand
- Identifiers (Labels)
- Expressions

A.5.4 Comment field

Comments can be placed in the program. Any field that follows a semi-colon (;) other than in a string constant definition is a comment. When the assembler encounters a semicolon (other than in a string constant definition) it stops parsing of that line.

for example :

```
;This is a comment.
        ; So is this.
```

The comment can be in the same line as the statement or in a separate line.

A.5.5 Direct assignment statements

A direct assignment statement assigns the value of an arbitrary expression to a specified identifier. The direct assingment statements can be given only in data definition section (explained later). The format of a direct assignment statement is:

```
<identifier>  =  <expression>
```

Examples of the direct assignments are:

```
total    =  100
a_count  =  total/3
b_count  =  total - a_count}
```

In addition an identifier which has been defined in a direct assignment statement cannot be used as label later. Both situations give rise to assembler error messages.

A.6 Sections in assembly program

The Assembly program can have three kinds of sections. They are

1. **Data Declaration Section** This section is used to define data. A data declaration section can be had any where in the assembly program, except inside a macro definition. There can be any number of data declaration sections. A data declaration section cannot encapsulate another data declaration section.

2. **Macro Definition Section** This section is used for defining macros. Each macro definition needs a separate macro definition section. A macro definition section can be any where in the assembly program execept in a data declaration section. Nested macro definitions are not supported (i.e. a macro can't be defined within a macro), only a macro call can be done within a macro.

3. **Code Section** This section contains the instruction lines, macro calls and pre-defined macros. Only the program lines in this section generate machine code.

The Assembler directives needed to define the sections are explained in the next section. The Code Section is taken to be the default section if no other section is defined (i.e. There are no special directives to specify this section). An example assembly program with corresponding Assembler output are given in appendix E.

A.6.1 States of assembler

At any stage the assembler will be in one of the following states:

1. DATA DEFINITION : This is the state of the assembler in data declaration section. At the end of data declaration section the state of assembler changes to CODE.

2. MACRO DEFINITION : This is the state of the assembler in macro definition section. At the end of macro definition section the state of assembler changes to CODE.

3. MACRO EXPANSION : This will be the state of the assembler after a macro call. At the end of macro expansion the state of the assembler changes to MACRO DEFINITION (if the macro call was in macro definition) or CODE (if the macro call was in code section).

4. CODE : This will be the state of assembler at the start. The program has to end in this state (i.e. in code section).

A.7 Assembler directives

All the assembler directives have to be given at the start of a new line. They can be preceded by space characters but not any other characters. They can be given in both cases (upper and lower) and are equivalent.

A.7.1 DATA

DATA (or *data*) signals the commencement of data declaration section. This has to be followed by one or more declarations of identifiers, constants, etc. Data declaration should be terminated by *dend* directive (explained later). It is not necessary that atleast one declaration is done between *data* and *dend*, They can be empty also. Between *data* and *dend* no other assembler directive is allowed except *extern* directive (explained later). On seeing this directive the state of the assembler changes from CODE to DATA DEFINITION.

syntax :

```
DATA
 OR
data
```

Between *data* and *dend* no other assembler directive is allowed except *extern* directive (explained later).

Data definitions

An identifier can be defined to be of type byte (unsigned) using `DB` or `db`, of type word (integer, four bytes) using `DW` or `dw`, of type double (eight bytes) using `DD` or `dd`. Constants can be defined in data section (syntax has been given in previous section). Identifiers can used only after declaration.

An example data declaration section.

```
DATA                    ; Start of Data Declaration Section
flag  db 10             ; Defines flag as  byte type with initial value 10
count  dw 1             ; Defines count as word type, with initial value 1
str=  "Example"         ; Defines str as string constant.
j =  1<<2               ; Defines j as numeric constant.
extern ex,ext           ; Defines ex & ext as external variables.
x  db  (12,34,45,67,89)     ; Defines an array of five integers with
                            ; 12, 34, 45, 67 & 89 as initial values and
                            ; x refers to the first byte.
DEND                        ; End of Data Declaration Section
```

A.7.2 DEND

DEND or *dend* signals the end of the data declaration section. On seeing this directive the state of the assembler changes from DATA DEFINITION to CODE. This directive is compulsory at the end of every data declaration section.

syntax :

```
DEND
 OR
dend
```

A.7.3 MACRO

MACRO (or *macro*) signals the start of the macro definition section. On seeing this directive the state of the assembler changes from CODE to MACRO DEFINITION. This directive is to be followed by name of the macro (an identifier) and an optional list of dummy parameters. All the statements after this directive until the occurrence of *MEND* directive (explained later in this section) are taken to be those of the macro which is being defined. Between *MACRO* and *MEND* no other assembler directive is allowed except *LOCAL*.

syntax :

```
MACRO  <macro_name>   <list_of_parameters>
                OR
macro  <macro_name>   <list_of_parameters>
```

where <macro_name> is any permitted identifier (explained earlier) and <list_of_parameters> is

```
[ <identifier> [ , <identifier> ] . . . ]
```

A.7.4 MEND

MEND (or *mend*) signals the end of the macro definition section. On seeing this directive the state of the assembler changes from MACRO DEFINITION to CODE. This directive has to be compulsoryly given at the end of the macro definition section.

syntax :

```
MEND
 OR
mend
```

A.7.5 LOCAL

LOCAL or *local* can be used to declare some labels to be local to a macro. If any labels are used in a macro without defining them as local to a macro then they are assumed to be global labels. This directive can be given at any point in the macro definition, but it is considered to be a local label only after it has been declared, all the occurrences before declaration are treated as global label definitions

syntax :

```
LOCAL  <list_of_local_labels>
                          OR
       local  <list_of_local_labels>
```

where <list_of_local_labels> is

```
<identifier> [ , <identifier> ] . . .
```

A.7.6 EXTERN

EXTERN or *extern* can be used to declare the identifiers as external (those which have not been declared but used in that particular file). This directive has to be given only in the data declaration section. The labels which have been declared to be external have to be declared in some other file and linked using the *Linker* (explained in appendix C).

syntax :

```
EXTERN  <list_of_extern_symbols>
                              OR
extern  <list_of_extern_symbols>

where <list_of_extern_symbols> is

            <identifier>  [  ,  <identifier>  ]  . . .
```

A.7.7 ENTRY

ENTRY or *entry* is a pre-defined label. The address of the location where this label is declared is taken as starting point for the execution of the file. A special symbol "_start" is added to the symbol table to store the current location counter value. This should be given only in code section.

syntax :

```
ENTRY :
  OR
entry :
```

A.8 Pre-defined macros

A.8.1 MFORK

MFORK can be used for generating the *mfork* instruction. It takes two register parameters and labels (maximum four and minimum one) from which the threads have to be generated using *mfork*. *MFORK* generates six *Twine-RISC* instructions. The first five instructions are to load a 32 bit value into the register (the first operand) to be used by the *mfork* instruction

to generate threads. The sixth is the *mfork* instruction itself. If any of the labels are not defined or are defined as external symbols then assembler gives error messages.

syntax :

```
MFORK  REG_1 ,  REG_2  <label_list>
```

where REG_1 , REG_2 are two register operands
and <label_list> is

```
[ <identifier> [ , <identifier> ] . . . ]
```

A.9 Error messages of assembler

In this section we give the error messages from the Assembler. We explain the meaning of the message and possible causes of the error.

Every message starts with the words *<filename> line :<lineno>* <filename> is the name of the input file in which error has occurred. <lineno> is the line number in which error has occurred.

1. *Unknown mnemonic or undefined macro call* : Operation code field is Unrecognizable. A macro which is not defined might have been called. There may be a spelling mistake in instruction mnemonic or macro call.

2. *Illegal character '#' in <identifier>* : Character '#' is not an element of assembler and cannot be used.

3. *Undefined Identifier <id_name>* : <id_name> has been used without defining.

4. *Register number too high* : Register number used is more than 63.

5. *Illegal character < char >* : Character (<char>) which is not an element of Assembler is used.

6. *Data definition not allowed in macro definition* : Assembler directive DATA given in a macro definition. As said earlier data definition is not allowed in macro definition.

7. *LOCAL allowed only in macro definition* : Assembler directive LOCAL is given outside a macro definition. Defining of local variables is allowed only in macro definition.

8. *Identifier redefined* : An Identifier is defined twice.

9. *Program ended abruptly* : Program ended in a macro definition or data definition section.

10. *Macro redefined* : A macro has been defined more than once.

11. *Integer (between 0 and 63) expected as first operand* : The first operand should be integer between 0 and 63 for sftl, sftr storex and loadx instructions.

12. *Label operand expected* : Non label operand is given to one of the Jump instructions.

13. *Integer expected as second operand* : Integer between 0 and 4023 expected for mvi instruction.

14. *Incorrect number of operands* : Number of operands given is not equal to that expected.

15. *Register operand expected* : Non register operand is given at a place where register operand is expected.

16. *Conditional jump offset more than 0x7ff* : Length of the jump is more than what is permitted in conditional jump.

17. *Too many label operands for MFORK macro* : Maximum number of label operands allowed for MFORK is four.

18. *Non-label operands given to MFORK macro* : MFORK expects labels to the location from which threads have to be generated.

19. *Label <label_name> not defined* : label <label_name> has been used but not defined.

Appendix B

Instruction Set of Twine-RISC

Twine-RISC has 22 instructions. Each of these instructions can be executed in single clock cycle except one (*MFORK* instruction). These instructions are classified into five major categories viz. Arithmetic and Logic, Copy, Branch, Memory reference and Generation and synchronization of multiple threads. In this appendix, we explain the function of the instructions of *Twine-RISC*. Here we often refer to FP (Frame Pointer) and IP (Instruction Pointer) of TRS [DIN93, DH92]. We have explained these in chapter 4.

A detailed description of how bits are encoded and the actions done by the Twine-RISC is given [DIN93, DH92]. Any register access is starting from FP (Frame Pointer).

B.1 Arithmetic and logic group

This group of instructions perform the arithmetic and logic operations.

B.1.1 ADD, SUB, AND, OR, XOR instructions

These instructions need three register operands. The operation is done on contents of first two registers and the result is stored in the third register.

The syntax of these instructions is

opcode rs1, rs2, rd where

rs1 - left operand source register.

rs2 - right operand source register.

rd - destination register.

The operation is performed on [FP + rs1] and [FP + rs2]. The resultant value is stored in [FP + rd]. Execution continues from the next location (IP + 1) and no continuation token is generated.

For example: ADD r1, r2, r3

Adds the contents of r1 and r2 and stores the result in r3.

B.1.2 SFTL, SFTR instructions

Shift the operand *left* or *right*. These instructions take one integer (between 0 and 63) and two register operands.

The syntax is

opcode x, rs, rd.

x - number of bits to be shifted.

rs - operand source register.

rd - destination register.

[FP + rs] is shifted to left or right x-bits and the result is stored in [FP + rd]. The execution continues from (IP + 1) and no continuation token is generated.

For example : SFTR 12,r3,r1

Shifts the contents of r3 12-bits to right and stores the result in r1.

B.2 MVI instruction

This instruction is to move immediate data (from instruction template) to register.

The syntax is

MVI rd,x

rd - destination register.

x - Integer operand.

Moves x to 12 least significant bits of [FP + rd]. This will not change the 20-most significant bits of [FP + rd]. Absolute value of x should be less than 2047 (0x7ff). The execution continues from (IP + 1) and no continuation token is generated.

For example : MVI r6, 44

Moves 44 into 12-least significant bits of r6.

B.3 Branch instructions

These are the jump group of instructions.

B.3.1 JMP instruction

JMP is an unconditional jump instruction (supports direct jump up to a address range of (2^{18}-1).

The syntax is

JMP <label_name>

<label_name> - Label to which the jump has to be made.

The thread will be stopped and a continuation token with IP as address of the location at which label <label_name> is defined and FP as that of parent is generated.

B.3.2 JUMP instruction

JUMP is an unconditional jump instruction (supports jump to a address range of (2^{32}-1)).

The syntax is

JUMP rs.

rs - register specifying the jump offset.

The thread is stopped and a continuation token (FP.IP + [rs]) is generated.

B.3.3 JZ, JP, JPZ, JNZ instruction

These instructions support conditional jump up to 12-bit offset from the current IP. A register operand which contains the value on which the condition is to be tested has to be given.

The syntax for these instructions is

JCND rs, <label_name>

rs - condition operand source register.

<label_name> - Label to which the jump is to be made if the condition is true.

If the condition is true on the value in register rs the current thread is stopped and a continuation token with IP as address of the location at which label <label_name> is defined) and FP as that of parent is generated. If the condition is false, continuation token is not generated and execution continues from (IP + 1).

B.4 Special instructions

These instructions are extensions to the conventional RISC instruction set.

B.4.1 MFORK instruction

MFORK spawns parallel threads of computation from an executing thread. Four possible new threads are organized as 8-bit offsets n1, n2, n3, n4 and grouped into a 32-bit word. This is stored in a register.

The syntax is

MFORK rs, rd

rs - operand source register from which new thread offsets are derived.

rd - destination register.

[FP + rs] is interpreted as four bytes. For each byte, if the value is non-zero, a new thread is generated. A continuation token of (FP,IP + offset value) generated for each non-zero offset value. One continuation token (FP, IP + 1) is always generated. Finally the number of threads generated (including "IP + 1") is stored in [FP + rd]. This value can be used by the MJOIN instruction to synchronize the threads.

B.4.2 MJOIN instruction

The MJOIN instruction helps in the synchronization of multiple threads. This instruction decrements the specified register content by 1 and writes the result back in the same location.

The syntax of MJOIN is

MJOIN rs, rs.

rs - operand source register which contains the number of threads to be synchronized.

MJOIN decrements [FP + r2] by 1 and tests it. If the resultant value is zero, the continuation (FP, IP + 1) is generated else the thread dies.

B.4.3 CHFP instruction

This instruction changes the FP with the first 6-bits of the specified register.

The syntax is CHFP rs.

rs - operand source register.

The 6-least significant bits of rs are loaded to FP. the number in rs should be between 0 and 63 (both inclusive).

B.5 Memory based instructions

These instructions are used to move data to and from the global memory.

B.5.1 LOAD instruction

This instruction is used to load data from the global memory to the registers.

The syntax of this instruction is

LOAD rs, rd.

rs - operand source register containing the global memory location.

rd - destination register.

Request is sent to the memory controller for data at address [FP + rs]. The thread dies and is resumed after the memory controller responds. The data returned by the memory is stored in [FP + rd].

B.5.2 LOADX instruction

The syntax is

LOADX a, rd.

a - 6-bit value specifying the address of the global memory location in the instruction.

rd - destination register.

Request is sent to the memory controller for data at address a. The thread dies and is resumed after the memory controller responds. The data returned by the memory is stored in [FP + rd].

B.5.3 RESM instruction

This instruction is executed to move the data from the data queue to the register.

The syntax is

RESM

This instruction stores the data returned by the memory (on load request) in the registers. It reinitiates the thread which stopped on memory request. This instruction should not to be given in assembly program. The system generates it automatically, when necessary.

B.5.4 STORE instruction

This instruction is used to store data into the global memory from the registers.

The syntax is

STORE rd, rs.

rd - operand source register containing the address of the global memory location. (to which the data is to be moved).

rs - operand source register from which data is to be moved to the global memory.

[rs] is sent to memory for storing in [FP + rd]. The thread continues from the next instruction.

B.5.5 STOREX instruction

The syntax is

STOREX x, rs

x - 6-bits specifying the address of the global memory location in the instruction.

rs - operand source register from which data is to be moved to the global memory.

[rs] is sent to memory for storing in x. The thread continues from the next instruction.

B.6 Mnemonic table

#	Instruction	Description of the Instruction
1	add rs1, rs2, rd	Adds contents of rs1 & rs2 and stores the result in rd.
2	sub rs1, rs2, rd	Subtracts contents of rs2 from contents of rs1 and stores the result in rd.
3	and rs1, rs2, rd	Logical AND of contents of rs1 & rs2 is stored in rd.

Cont'd ...

#	Instruction	Description of the Instruction
4	or rs1, rs2, rd	Logical OR of contents of rs1 & rs2 is stored in rd.
5	xor rs1, rs2, rd	Logical XOR of contents of rs1 & rs2 is stored in rd.
6	sftl x, rs, rd	Shifts the contents of rs to the left by x-bits and stores the result in rd.
7	sftr x, rs, rd	Shifts the contents of rs to the right by x-bits and stores the result in rd.
8	mvi rd, x	Moves x to register rd.
9	jmp <label>	Unconditional jump to <label>.
10	jump rs	Unconditional jump to a location whose offset is in rs.
11	jz rs, <label>	Jump to <label> if contents of rs is zero (equal to).
12	jp rs, <label>	Jump to <label> if contents of rs is positive (greater than).
13	jpz rs, <label>	Jump to <label> if contents of rs is positive or zero (greater than or equal to).
14	jnz rs, <label>	Jump to <label> if contents of rs is negative or zero (less than or equal to).
15	mfork rs, rd	generate threads using contents of rs and store no. of threads generated in rd.
16	mjoin rs, rs	Decrement contents of rs and test it, if it is zero thread continues else thread dies.
17	chfp rs	Loads the first six bits of rs to FP.
18	load rs, rd	Load the contents of memory address [rs] into rd.
19	loadx a, rd	Load the contents memory address a into rd.
20	resm	Move data from data queue to register.
21	store rd, rs	Store the contents of rs into memory address [rd].
22	storex x, rs	Store the contents of rs into memory address x.

Appendix C

Linker User Manual

C.1 Description

Linker takes as input the object file generated by the Assembler. Linker generates an executable file for the Twine-RISC machine. The object modules on which linker operates are specified on the command line.

C.2 Synopsis

The usage of the Linker is

```
link  [ -e entry ] [ -o name ] [ -r ] [ -s ] [ -ysym ] [ -s ] [ -t ]
filename ...
```

C.3 Linker options

The options and file names can be given in any order.

-g

is used for linking debugging objects.

-o <filename>

is used for specifying the output file name. The user is adviced to use this option. if no output file name is given then linker appends ".out" to the start of the first input

file name and writes output into it. For example, if the name of first input file is "ip" then the output will be stored in "ip.out".

-e <id_name>

is used for specifying the entry point of the executable file which will be output of the Linker. The *id_name* that will be specified with this option should be defined as a label. If there is more than one symbol with this name then the first symbol which has been defined as label will be taken as entry point.

-r

Generate relocation entries in the output file so that it can be the subject of another Linker run. This flag also prevents error messages, if some of the external variables are not declared.

-t

Trace. Display the name of each file as it is processed.

-s

Strip. Used to remove the symbol table, string table line number entries and relocation entries to save space (but impair the usefulness of the debuggers).

-ysym

Display each file in which *sym* appears, its type and whether the file defines or references it. Only one symbol can be given with this option.

C.4 Object file processing

The files specified on the command line are processed in the order listed. Information is extracted from each file, and concatenated to form the output.

After linker has processed all input files and command line options, the form of the output it produces is based on the information provided in both.

Linker gives an error if it could not resolve a cross reference. If it could resolve all the cross references, it produces a completely linked executable file (for Twine RISC).

C.5 Error messages of linker

In this section we give the error messages from Linker. We explain the meaning of the message and possible causes of the error.

1. *Entry : no label with name* < *label_name* > : An identifier with the name given in -e option (< *label_name* >) does not exist.

2. *Duplicate symbol, error in input file* < *file_name* > : Two symbol are there with same name in input file (object file) < *file_name* >.

3. *External symbol* < *symbol_name* > *is not defined* : A symbol which has been declared to be external has not been defined in any the files which are being linked.

4. *Illegal input file* < *file_name* > : Input file < *file_name* > is not an object file (of Twine-RISC).

Appendix D

Simulator User Manual

D.1 Introduction

Simulator takes the executable files of the TWINE-RISC machine and simulates the execution on Twine-RISC processor. The Simulator accepts the input executable file in COFF format (Common Object File Format).

The execution starts from the location specified by the *entry* field of the optional header in input file. This *entry* contains the program entry location which can be specified using a linker option (See Appendix C), or the *entry* label of assembler source (See Appendix A). If none of this is defined then the start of the user code in text section is taken as the starting point.

Simulator also provides some primitive debugger options like executing in single steps or specified no of steps, printing or loading values of registers and data locations at any stage of execution and printing of the source code line for any instruction (If the source file is compiled with -g option) etc. These will be explained in more detail later.

D.2 Synopsis

```
sim [ -d ] [ -i ] [ -o ] [ -c ] input_filename
{ [-mconstant <cycles> ] | [ -mvariable  <min_cycles>  <max_cycles> ]  }
```

D.3 Options

The filename and options can be given in any order. The options that are available in the simulator are :

-d

This option takes the simulator into debugging mode and need not be given if only the execution (and not trace) of the input file is desired. More details about the debugger are given later in the manual.

-i

This option is used for loading required values into the registers before execution of the input file. Using -d option, however one can change the register values at any time during the execution. At the start, simulator expects the register number and its value. Any number of such tuples can be specified. The register initialization terminates when a negative number is used for a register.

-o

This option is used for printing values of the registers and data locations after the execution of the input file. Using -d option one can see the register contents at any time during the execution.

-c

This option is used for printing the instruction and its operands while executing an instruction. The output is similar to the one that is printed with -c option in the assembler.

-t<NO_OF_TRS>

This option is used for giving the number of Twine RISC Streams in the architecture being simulated. The maximum limit of NO_OF_TRS is currently 8 (eight). If -t option is not given, the default number of streams is taken as two.

-mconstant <cycles>

This option is used to specify the response time of the global memory (i.e. no. of cycles from the memory request and the corresponding response). If this option is given then the response time is taken to be constant for all accesses.

-mvariable <min_cycles> <max_cycles>

Like the above option this option is also to specify the response time of the global memory, But with this option the response time is taken to be a random variable which can take value between <min_cycles> and <max_cycles> (both inclusive). The simulator uses a different random number for each memory request to model variable memory latency.

D.4 Debugger commands

The total length of the debugger command should not exceed 200 characters. If the first parameter of a debugger command is an integer then space between this parameter and the command is not necessary.

- **h**

  This command gives the on-line help available.

- **q**

  This command is used to quit the debugger.

- **s**

  This command is used for executing program in single step.

- **s <n>**

  This command is used for executing n steps of the program. n is expected to be positive. n steps may mean a maximum of $n \times$ *number of streams* instructions to be executed. The minimum number of instructions executed is 0(zero, when there are no instructions pending).

- **S**

  This command is used to print the source line of the next instruction to be executed (only for those streams which have threads). This can be printed only if the source file has been compiled with -g option.

- **P**

  This command is used to print the Performance of the simulated Twine RISC architecture compared with that of Twine RISC with single TRS, using the current program. The output also gives the percent utilization of the Twine RISC architecture with the given number of streams. This value will be closer to 100 if for most of the time all the Twine RISC Streams were busy. It will be 100 if through out the execution the number of threads is more than or equal to the number of Twine RISC streams.

- **pr <reg_no>**

  This command is used to print the value in register *reg_no* .

- **pr <reg_1> <reg_2>**

  This command is used to print the values in the registers from *reg_1* to *reg_2* (both inclusive).

- pd <addr>

  This command is used to print the integer value at data location (address) n.

- pdi <addr>

  Same as above.

- pdb <addr>

  This command is used to print the byte value at data location (address) n.

- pds <addr>

  This command is used to print the short value (two bytes) at data location (address) n.

- pdd <addr>

This command is used to print the double value (eight bytes) at data location (address) n.

- lr <reg_no> <value>

  This command is used to load *value* into register *reg_no* .

- ld <addr> <value>

  This command is used to store <value> (integer, four bytes) into data location (address) <addr>.

- ldi <addr> <value>

  Same as above.

- ldb <addr> <value>

  This command is used to store <value> (byte) into data location (address) <addr>.

- lds <addr> <value>

  This command is used to store <value> (short, two bytes) into data location (address) <addr>.

- ldd <addr> <value>

  This command is used to store <value> (double, eight bytes) into data location (address) <addr>.

- n

  This command is used to print the next instruction to be executed in different streams (only for the streams which have a thread running). If there are free streams and tokens waiting, they can be loaded using *L* command (see later in the section).

- t

  The tokens waiting in the token queue.

- d

  The data values waiting in the data queue.

- m

  The messages waiting in the message processor queue.

- e

  This is to continue the execution from the present location, until the end of the execution. While executing this command the breakpoints are not honoured.

- D

  This command is to print the data about the current input file.

- C

  Similar to *e* command, except the breakpoints are honoured.

- bs <addr>

  This command is to set the breakpoint at location <*addr*>. <*addr*> has to be between 0 and maximum address (end of text section which can be found using *D* command). <*addr*> can be obtained from the code listed by the assembler if *-c* option is given.

- bp

  This command is to give the location where breakpoint is set (if it is set).

- bc

  This command is to clear the breakpoint.

- cs

  This command is to set the flag for displaying the code during execution. This is same as giving -c option at simulator. The instruction is not displayed if the flag is not set. It is displayed if the flag is set (see commands *cr, cp & c*)).

- cr

  This command is to reset the flag for printing the code during execution. If the flag is set by giving *-c* option at simulator or *cs* command in the debugger it can be reset using this command (see *cs, cp & c*).

- cp

  This command prints the status of the flag which controls the code display (see *cs, cr & c*).

- c

  This command is to toggle the value of the flag controlling code display (see *cs, cr & cp*).

- L

  This command is to load the threads which are waiting (if any) into empty Streams (if any). This loading of threads will be done automatically by the Twine-RISC before each cycle. This command will be useful to see the instructions which will be executed next (using *n* command) or their source code (using *S* command).

- a <loc> <FP>

  This command is to add a token with IP address as <*loc*> and <*FP*> as frame pointer.

D.5 Error messages of simulator

In this section we give the error messages from Simulator. We explain the meaning of the message and possible causes of the error.

1. *Illegal jump address* : The given jump address is going out of the bounds of text section.

2. *Illegal mfork offset address* : The offset address given to mfork for creating threads is going out of bounds of text section.

3. *More than one input file given* : The simulator takes only one executable file (of Twine-RISC) at a time as input.

4. *Illegal entry address* : The entry address in the input file is out of bounds of text section or not pointing to the start of an instruction.

5. *Illegal input file* : The input file given to the Simulator is not an executable file (of Twine-RISC).

6. *Illegal data address* : The data address is crossing the bounds of data section.

7. File not compiled with -g option : The source code file of the input file is not compiled with -g option and an attempt is made to access this information.

8. Illegal instruction code used : The given Instruction opcode does not belong to one of the 22 permitted instructions.

9. *Register out of bound* : Register number used is less than zero or more than 63, which is not allowed in the architecture.

10. *Error in thread synchronization* : One threads may be executing while another thread gives an end of program signal.

Appendix E

Example Program

In this appendix we give example program for the Assembler with the Listing and Code-Listing for that program. During parsing the Assembler writes what it has parsed into a file specified with -l option. This is called Listing. After successful assembly the Assembler gives a listing of the code it has generated. This is called Code Listing.

E.1 Assembly program

In the next page an example assembly program is given.

```
;Program to find the square of two numbers
; z = x**2 + y**2


DATA    ; this is a comment
x11  db 10
x12  db 20
x13  db 30
x14  db 40
x22  dw 1
j =  1<<2
x  db  (12,34,45,67,89)
dend

MACRO load_val value,n      ; Macro to load ''value''  into n.
xor n,n,n                   ; Load zero into n.
mvi n,value                 ; Move ''value'' into n.
MEND                        ; End of Macro


MACRO square x
local leb

load_val 0,r10
load_val 0,r11
load_val 1,r12

add x,r10,r10
add x,r11,r11
leb: add x,r10,x
sub r11,r12,r11
jp r11,leb
sub x,r10,x
MEND ; square macro definition ends


entry:
MFORK r6,r7,cal_y2
square r1 ; find square of x
jmp final
cal_y2:
load_val 20,r5
chfp r5
square r1 ; find square of y
load_val 0,r5
chfp r5
final:
mjoin r7,r7
add r1,r21,r2 ; store (x**2 + y**2) in r3
```

End of Assembler input

E.2 Assembly listing

In this section we give the Assembler listing of above program of previous section.

```
0000                          1       MFORK  r6  r7  <LABELS>
                        *** MACRO <square> ***
0012   60 a2 8a            2       xor   r10, r10, r10
0015   e8 a0 00            3       mvi   r10,0
0018   60 b2 cb            4       xor   r11, r11, r11
001b   e8 b0 00            5       mvi   r11,0
001e   60 c3 0c            6       xor   r12, r12, r12
0021   e8 c0 01            7       mvi   r12,1
0024   00 12 8a            8       add   r1, r10, r10
0027   00 12 cb            9       add   r1, r11, r11
002a                       10    #square_leb_0 :
002a   00 12 81           11       add   r1, r10, r1
002d   10 b3 0b           12       sub   r11, r12, r11
0030   9c 00 0b           13       jp    r11, #square_leb_0
0033   10 12 81           14       sub   r1, r10, r1
                        *** MEND <square> ***
0036   c8 00 00           15       jmp   final
0039                       16    cal_y2 :
                        *** MACRO <square> ***
0039   60 51 45           17       xor   r5, r5, r5
003c   e8 50 14           18       mvi   r5,20
                        *** MEND <square> ***
003f   dc 50 00           19       chfp   r5
                        *** MACRO <square> ***
0042   60 a2 8a           20       xor   r10, r10, r10
0045   e8 a0 00           21       mvi   r10,0
0048   60 b2 cb           22       xor   r11, r11, r11
004b   e8 b0 00           23       mvi   r11,0
004e   60 c3 0c           24       xor   r12, r12, r12
0051   e8 c0 01           25       mvi   r12,1
0054   00 12 8a           26       add   r1, r10, r10
0057   00 12 cb           27       add   r1, r11, r11
005a                       28    #square_leb_1 :
005a   00 12 81           29       add   r1, r10, r1
005d   10 b3 0b           30       sub   r11, r12, r11
0060   9c 00 0b           31       jp    r11, #square_leb_1
0063   10 12 81           32       sub   r1, r10, r1
                        *** MEND <square> ***
                        *** MACRO <square> ***
0066   60 51 45           33       xor   r5, r5, r5
0069   e8 50 00           34       mvi   r5,0
                        *** MEND <square> ***
006c   dc 50 00           35       chfp   r5
006f                       36    final :
006f   fc 71 c0           37       mjoin   r7, r7
0072   00 15 42           38       add   r1, r21, r2
```

End of Assembler listing.

E.3 Code listing

In this section we give the Code listing for the assembly program given in section 1.

```
0000    e8 60 00          0   mvi  r6 ,  0x0
0003    4c c1 86          1   sftl   12 , r6 ,r6
0006    e8 60 00          2   mvi  r6 ,  0x0
0009    4c c1 86          3   sftl   12 , r6 ,r6
000c    e8 60 2a          4   mvi  r6 ,  0x2a
000f    6c 61 c0          5   mfork  r6 ,  r7
0012    60 a2 8a          6   xor  r10 , r10 ,r10
0015    e8 a0 00          7   mvi  r10 ,  0x0
0018    60 b2 cb          8   xor  r11 , r11 ,r11
001b    e8 b0 00          9   mvi  r11 ,  0x0
001e    60 c3 0c         10   xor  r12 , r12 ,r12
0021    e8 c0 01         11   mvi  r12 ,  0x1
0024    00 12 8a         12   add  r1 , r10 ,r10
0027    00 12 cb         13   add  r1 , r11 ,r11
002a    00 12 81         14   add  r1 , r10 ,r1
002d    10 b3 0b         15   sub  r11 , r12 ,r11
0030    9f fe 8b         16   jp  r11 ,  0x2a
0033    10 12 81         17   sub  r1 , r10 ,r1
0036    c8 00 6f         18   jmp   0x006f
0039    60 51 45         19   xor  r5 , r5 ,r5
003c    e8 50 14         20   mvi  r5 ,  0x14
003f    dc 50 00         21   chfp  r5
0042    60 a2 8a         22   xor  r10 , r10 ,r10
0045    e8 a0 00         23   mvi  r10 ,  0x0
0048    60 b2 cb         24   xor  r11 , r11 ,r11
004b    e8 b0 00         25   mvi  r11 ,  0x0
004e    60 c3 0c         26   xor  r12 , r12 ,r12
0051    e8 c0 01         27   mvi  r12 ,  0x1
0054    00 12 8a         28   add  r1 , r10 ,r10
0057    00 12 cb         29   add  r1 , r11 ,r11
005a    00 12 81         30   add  r1 , r10 ,r1
005d    10 b3 0b         31   sub  r11 , r12 ,r11
0060    9f fe 8b         32   jp  r11 ,  0x5a
0063    10 12 81         33   sub  r1 , r10 ,r1
0066    60 51 45         34   xor  r5 , r5 ,r5
0069    e8 50 00         35   mvi  r5 ,  0x0
006c    dc 50 00         36   chfp  r5
006f    fc 71 c0         37   mjoin  r7 ,  r7
0072    00 15 42         38   add  r1 , r21 ,r2
```

End of code listing.

E.4 Linking and running on Simulator

The Assembler will produce an object code for the program given in the erlier section. This object code will be linked to produce an executable file for the Twine-RISC architecture.

This executable file was run on the simulator of Twine-RISC with two TRSs. The values given for x (r1) and y (r21) are 40 and 30 respectively. The performance metrics given by the simulator are:

Number of cycles taken by Twine-RISC with 2 TRSs is 138 cycles.

Number of cycles taken by Twine-RISC with one TRS is 244 cycles.

Ratio of number of cycles taken by Twine-RISC with one TRS/two TRS is 1.77

Percentage utilization of the Twine-RISC with two TRSs is 88.41 .

The above data gives the parallelism available in the example program for the given input values.