Design and Modeling of a Real-Time
RISC Processor in VHDL

by

Syed Asaf Maruf Ali

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

June, 1994
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Design and modeling of a real-time RISC processor in VHDL

Ali, Syed Asaf Maruf, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1994
Design and Modeling of a Real-Time
RISC Processor in VHDL

BY
Syed Asaf Maruf Ali

A Thesis Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAH, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
COMPUTER ENGINEERING

JUNE 1994
KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHARAN, SAUDI ARABIA

This thesis, written by

Asaf Maruf Ali

under the direction of his Thesis Advisor, and approved by his Thesis committee, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee:

Chairman (Dr. Khalid M. Elleithy)

Member (Dr. Sadiq M. Sait)

Member (Dr. M.S.T Benten)

Member (Dr. A. Amin)

3.1.99

Department Chairman

Dr. Ala H. Rabeh
Dean, College of Graduate Studies

Date: 4.7.94
Design and Modeling of a Real-Time RISC Processor in VHDL

Syed Asaf Maruf Ali

COMPUTER ENGINEERING

JUNE 1994
Dedicated to

my parents,
sisters and nieces
Acknowledgment

In the name of Allah, Most Gracious, Most Merciful. Read in the name of thy Lord and Cherisher, Who created. Created man from a \textit{lecch-like} clot. Read and thy Lord is Most Bountiful. He Who taught \{the use of\} the pen. Taught man that which he knew not. Nay, but man doth transgress all bounds. In that he looketh upon himself as self-sufficient. Verily, to thy Lord is the return \{of all\}.

(The Holy Quran, Surah 96)

First and foremost, all praise to the Almighty Allah Who gave me the courage and patience to carry out this work. I am happy to have had a chance to glorify His name in the sincerest way through this small accomplishment and ask Him to accept my efforts. May He guide us and the whole humanity to the right path \textit{(Aameen)}. Acknowledgement is due to King Fahd University of Petroleum and Minerals for providing support to this work.

My deep appreciation goes to my major thesis advisor Dr. Khalid M. Elleithy for his constant help, guidance and the countless hours of attention he devoted throughout
the course of this work. He was always kind, understanding and sympathetic towards me. Working with him was indeed a wonderful and learning experience which I thoroughly enjoyed.

Thanks are also due to my thesis committee members Dr. M.S.T. Benten, Dr. Sadiq M. Sait and Dr. A. Amin for their interest, co-operation, advice and constructive criticism. In addition, I would like to thank Dr. Shafei for his invaluable discussions and practical insight of real-time systems.

I am also indebted to the department chairman, Dr. Samir Abdul Jauwad and other faculty members for their support.

Lastly, but not the least, thanks are due to my family members for their understanding, throughout my academic career, and also to my fellow graduate students, and all my friends on the campus especially (in alphabatical order) Aqil, Azher, Haroon, Idrees, Iftikhar, Irfan, Khalid, Masud, Mohammad, Neaz bhai, Sami and Shahid from whom I learned a lot and who made the long work hours pleasant.
Contents

Acknowledgement i

List of Tables viii

List of Figures ix

Abstract (English) xi

Abstract (Arabic) xii

1 Introduction 1

1.1 Real-time systems 2

1.2 RISC processors 3

1.3 VHDL 5

1.4 Motivation 5

1.5 Tasks to be accomplished 6

1.6 Thesis outline 7
2 Literature review

2.1 RISC for real-time applications ........................................... 9
   2.1.1 RISC architecture .................................................. 9
   2.1.2 Compilers ............................................................ 14
   2.1.3 Interrupt handling in real-time ................................... 15
   2.1.4 Cache memory in real-time ....................................... 16
   2.1.5 Register file in real-time ....................................... 18
   2.1.6 Memory requirements ............................................. 18
   2.1.7 Real-time languages .............................................. 19
   2.1.8 Architectural support of real-time systems .................... 19

2.2 System architectures ..................................................... 20
   2.2.1 Direct Memory Access (DMA) .................................... 21
   2.2.2 Virtual addressing ............................................... 22
   2.2.3 Data transfer protocols ......................................... 22

2.3 Modeling in VHDL .......................................................... 23

3 Real-time systems .......................................................... 26

3.0.1 Communication delays ............................................... 28
3.0.2 Computation delays .................................................. 28
3.0.3 Synchronization delays ............................................. 28

3.1 Desirable features of real-time systems ............................. 29
3.1.1 Predictability ........................................ 31
3.1.2 Interrupt handling capability ....................... 33
3.1.3 High speed I/O ....................................... 34
3.1.4 Fault tolerance ....................................... 35
3.2 Real-time operating systems ........................... 38
3.3 Real-time features available in commercial processors .................. 39
    3.3.1 Motorola 68040 .................................. 40
    3.3.2 ARM VL86C010 .................................. 41
    3.3.3 Motorola 88100 .................................. 43
    3.3.4 Intel 80960 ...................................... 45
    3.3.5 Intel 80196 ...................................... 47
    3.3.6 AMD 29000 ...................................... 48
3.4 A graphical display of RT features in some RISC CPUs .............. 50
3.5 Future of real-time systems ................................ 55

4 Instruction set design ................................... 57
    4.1 Instruction set design ................................ 57
    4.2 Addressing modes ................................... 64
    4.3 Instruction formats .................................. 66
    4.4 Proposed instruction set ............................. 68
    4.5 Pipelining ........................................... 69
5 Data path design 74

5.1 Data path design ........................................... 74

5.2 Datapath cycle analysis ................................... 77

5.2.1 Arithmetic and logical operations ..................... 77

5.2.2 Branch operations ..................................... 78

5.2.3 Bit manipulation operations ......................... 78

5.2.4 LOAD/STORE operations ......................... 78

6 Modeling decisions 83

6.1 Modeling ............................................. 83

6.2 Instruction set model .................................. 84

6.3 RISC model ........................................... 85

6.3.1 A reduced instruction set .......................... 86

6.3.2 Fixed format instruction size ....................... 86

6.3.3 A three port 32 x 32 register file ................ 86

6.3.4 An uncached system ................................ 86

6.3.5 A LOAD/STORE architecture ...................... 89

6.3.6 Local variable storage ............................. 89

6.3.7 Support for predictability ......................... 90

6.3.8 Support for high MIPS ............................ 91

6.3.9 Support for fault tolerance ....................... 91
6.3.10 Support for data I/O ........................................ 91
6.3.11 Support for interrupt handling ......................... 91
6.4 VHDL ............................................................... 92

7 Simulation results and discussion .......................... 94
7.1 Simulation results .............................................. 94
7.2 Discussion of simulation results ............................ 95
7.3 Conclusions ...................................................... 108

Appendix A .................................................................. 110
Bibliography .............................................................. 129

Vita ........................................................................... 136
List of Tables

2.1 Design and layout effort for some CPUs. ......................... 12
2.2 A comparison of RISC and CISC CPUs. ......................... 13

3.1 Desirable features of real-time systems on some CPUs. ........ 38
3.2 R-T features support at architectural and instruction set level. 39

4.1 Static analysis of ED and ROFF, word processors under UNIX. 58
4.2 Instruction usage of six compilers on VAX-11. .................... 59
4.3 Addressing mode usage on three VAX compilers. ............... 64
4.4 Arithmetic Instructions ........................................... 70
4.5 Logical Instructions ................................................ 70
4.6 Special Instructions ................................................. 71
4.7 Bit manipulation Instructions ...................................... 71
4.8 Flow Control Instructions ......................................... 71
4.9 Data Movement Instructions ........................................ 71

viii
List of Figures

3.1 Kiviat graph of desirable RT features of Motorola 88100. ............ 51
3.2 Kiviat graph of desirable RT features of Intel 80960. ................. 52
3.3 Kiviat graph of desirable RT features of ACORN RISC machine
   VL86010. ....................................................... 53
3.4 Kiviat graph of desirable RT features of proposed processor. ........ 54
4.1 Instruction mix in a typical program. .................................. 61
4.2 The instruction formats. .............................................. 67
4.3 A typical five stage pipeline. ....................................... 73
5.1 Arithmetic and logical operations. .................................... 79
5.2 Branch instructions. .................................................. 80
5.3 Bit manipulation operations. .......................................... 81
5.4 LOAD/STORE operations. ............................................. 82
7.1 Simulation result of setb instruction. ................................ 97
7.2 Simulation result of fflb instruction. ............................... 98
7.3 Simulation result of \texttt{ff0b} instruction. \\
7.4 Simulation result of \texttt{asl} instruction. \\
7.5 Simulation result of \texttt{mul} instruction. \\
7.6 Simulation result of a program showing branch and loop part 1. \\
7.7 Simulation result showing write to memory part 2. \\
7.8 Simulation result showing output of a real-time example.
Abstract

Name: Syed Asaf Maruf Ali
Title: Design and Modeling of a Real-Time RISC Processor in VHDL
Major Field: Computer Engineering
Date of Degree: June 1994

Real-time systems are characterized by high speed computation and strict timing constraints. Nowadays, powerful processors are capable of executing millions of instructions per second. These high speed processors can be utilized in the design of real-time systems which can benefit from their high performance in order to meet their strict timing constraints. This thesis is concerned with the study of real-time systems and their properties and determining the extent to which these real-time features are supported in currently available RISC processors. Once these features are identified, a new instruction set is proposed which attempts to target these features at the instruction set level. The instruction set is optimized for real-time applications by executing instructions in a single cycle and supporting powerful bit manipulation. The instruction set and the datapath are modeled in VHDL (VHSIC Hardware Description Language) at the behavioral level.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
June 1994
خلاصة الرسالة

الاسم: عاصف مروف على
عنوان الرسالة: تصميم و تمثيل معا لجات RISC في الوقت الحقيقي
بواسطة VHDL
التخصص: هندسة الحاسب اللالي
تاريخ الشهادة: يونيو 1994م

تتميز أنظمة الوقت الحقيقي بسرعة حسابية عالية وصرامة في القيود التي تفرضها على الوقت. وفي الوقت الحاضر، تستطيع المعالجات القوية من إنجاز ملايين التعليمات في الثانية الواحدة وبالتالي فإنه يمكن استغلال هذه المعالجات ذات السرعة العالية لتصميم أنظمة الوقت الحقيقي و ذلك بالاستناد من أدائها العالي لمقاولات التعديلات التي تفرضها أنظمة الوقت الحقيقي.


درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران _ المملكة العربية السعودية
يونيو 1994م
Chapter 1

Introduction

In our daily lives we come across a number of systems that must respond very quickly to changes in their inputs. Examples of such systems abound in the fields of avionics, space craft control, nuclear power plant control and powerful communication systems that need to operate in "real-time". In fact, even an airline reservation system represents a real-time system since transactions must be carried out within a certain time period. Since real-time systems play such an important role in our daily lives, it is necessary that we develop a good understanding of the issues involved in designing a reliable, fast and economical real-time system. The goal of this thesis is to study and identify desirable features of real-time systems and to study how these "desirable features" are supported in commercially available RISC processors. Then an instruction set is proposed along with a RISC based CPU core.
1.1 Real-time systems

The distinguishing characteristic of real-time systems is that correctness is not just a measure of the logical results of program execution but of the time at which these results are produced. Real-time systems are usually designed for specific applications due to which their design is ad hoc, expensive and complex. A real-time system involves two constraints. Firstly, it must adhere to a strict timing constraint and secondly the system must have predictable behavior. A design engineer’s foremost goal is to achieve deterministic response times. Once a real-time system is designed to be predictable, it’s performance can be improved. The issues involved in designing a reliable real-time system are numerous. Foremost among these is the guarantee that the system will behave in a predictable manner, that it must be fault tolerant and be highly responsive to external stimuli. Currently, research in this field deals largely with issues like schedulability and achieving deterministic behavior. However, there is an acute shortage of optimized instruction sets for real-time applications. Most of the active research in this area involves the design of fully pre-emptive operating systems and other distributed UNIX based systems. Chapter 3 of this thesis extensively deals with the science of real-time systems, their design and important properties.
1.2 RISC processors

RISC (Reduced Instruction Set Computer) processors are a natural choice for designing a real-time system due to their inherent faster execution speed. RISC processors attain good performance with the help of a reduced instruction set, pipelining, a large register file and a LOAD/STORE architecture. These factors contribute to high computation speed and overall good performance which is required in real-time systems. With the advent of high speed, single cycle RISC processors, a number of real-time applications are increasingly switching over to RISC processors in favour of once popular CISC processors. These applications range from high-end laser printers to robotics, aircraft avionics, underwater exploration, self-guided vehicles etc. Some RISC processors that have been employed for designing real-time applications are:

- 88000 family from Motorola

- 29000 family from AMD

- ARM family from ACORN Computers

- 80960 processor from Intel

RISC processors have a small instruction set and orthogonal instruction formats due to which their control logic is simple and design time is much shorter than comparable CISC processors. RISC processors can also be driven at higher clock frequencies
due to their simpler control and decode structures. RISC processors require a large memory bandwidth since more work has to be done due to their simple instruction sets. A large memory bandwidth can be satisfied by designing a high speed cache memory subsystem, employing a superscalar architecture, multiple functional units, pipelining, a LOAD/STORE architecture and in most cases a combination of these. RISC processors employ a LOAD/STORE architecture and most computation is register based. RISCs, therefore, require large register sets in order to support their load/store architecture and as a consequence, improve performance. However, a large register file tends to increase interrupt latency which is undesirable in real-time systems. Real-time systems require certain features in order to perform correctly. During the course of this study these “desirable” features were identified and ranked in order of merit. The objective of this study is to propose an instruction set suitable for real-time applications. These features must be supported at the architectural as well as the instruction set level. The architectural drawbacks of RISC that is heavy dependence on cache memory and a large register file are solvable in real-time systems. With a good, fully pre-emptable real-time operating system, a powerful real-time RISC computer can be constructed. Moreover, a RISC processor with fewer registers would definitely perform better in a real-time application since there will be less context to save during interrupts.
1.3 VHDL

The modeling language chosen is VHDL (Very High Speed IC Hardware Description Language). VHDL is an IEEE standard for modeling complex digital systems. It allows the user to define a behavioral as well as a structural view of a circuit. Since VHDL is structured like a high level language, it is easy to describe an algorithm to its equivalent hardware correspondence. In addition, VHDL supports features like custom data types, concurrency, generic statements, technology independence, library sharing etc. VHDL is chosen for modeling as it offers a number of advantages over the other HDL's namely the sharing of designs, more than one level of circuit description and support of hierarchical description of large designs and most importantly the ability to include timing information in the design. This enables accurate simulation of HDL models. Thus, we chose VHDL for modeling and verifying the instruction set of the RISC processor core at the behavioral level.

1.4 Motivation

Real-time systems play an important role in our lives. As we increase our dependence on automation, we will require more powerful processors to handle applications whose proper operation will save human lives and large sums of money and time. In order to successfully design a real-time system that is highly responsive, deterministic and very reliable, it is important that proper guidelines are followed. Real-time
systems are difficult to design since they require strict timing constraints. Albeit, real-time systems play such an important role there has not been significant research activities in specifying a proper instruction set optimized for real-time applications. In order to tackle this problem it was necessary to outline desirable features of real-time systems and to support these at the instruction set and hardware levels. We chose to implement these features at the instruction set level and relegate other features like timers and interrupt handlers to the hardware.

1.5 Tasks to be accomplished

1. To study important features of real-time systems.

2. To study RISC processors designed for real-time systems.

3. To propose an instruction set suitable for real-time systems.

4. To propose an architecture suitable for real-time systems.

5. Modeling and verifying the instruction set, processor and memory at the behavioral level in VHDL.

6. Designing a RISC CPU core on which features like timers, I/O, Interrupt handling, etc., can later be added.
1.6 Thesis outline

A study is carried out to determine desirable features of real-time systems. Once these features are identified they are ranked in order of their importance in a real-time system. Some commercially available RISC processors are analyzed for their suitability for real-time systems. This analysis is carried out at the architectural as well as the instruction set level. All the processors studied support some of these desirable features. The main objective is to incorporate all of these features at the architectural or instruction set level and to propose a RISC core on which missing features can later be added. The instruction set is modeled completely at the behavioral level in VHDL. The instruction set is simulated for correctness using a VHDL simulator.

Chapter 1 briefly introduces the reader to the concept of real-time systems, RISC processors and a hardware description language, VHDL. The next chapter is on literature survey, the current state of real-time systems, RISC processors being used in real-time systems and modeling digital systems in VHDL. Chapter 3 describes in detail real-time systems, their characteristics and future application areas. An extensive survey of current CISC and RISC processors is carried out in order to determine the extent to which real-time features are supported in available processors. The following chapter deals with the proposed instruction set, the instruction formats and the addressing modes and the decisions governing their choice. The
datapath arising from the description of the instruction set is discussed in chapter 5 along with figures showing how each instruction group is supported. In chapter 6 we discuss the important aspects of the modeling decisions considered in the architecture. Finally in chapter 7 we present the simulation results and conclusions.
Chapter 2

Literature review

2.1 RISC for real-time applications

2.1.1 RISC architecture

Computer architecture has actually gone back to its roots by employing the design philosophy of the early computers. During the early stages of computer evolution, computers were simple and had a small number of instructions. Studies done on computer programs from various disciplines showed that even complex programs consist of simple statements and instructions that are executed very frequently. These studies demonstrated that most programs consist of simple assignments, if statements and procedure calls [37]. Around the late 1970's, work started on simple, theoretically fast computers. In 1975 under the supervision of John Cocke of IBM, work
started on a machine known as 801 (the name refers to the number of the building where this machine was being designed). The 801 can be termed the first RISC machine however nothing was published till 1982. This processor embodied principles which are commonly found in a class of processors known as RISC [31]. The term RISC was first coined by Patterson et al. at Berkeley in 1981 when they introduced a 44,500 transistor chip named as RISC-I [30]. At about the same time, a group of researchers at Stanford University designed and implemented a processor they called MIPS (Microprocessor without Interlocked Pipeline Stages) [13]. Since then there have been vast improvements in RISC architecture resulting in large performance gains over their counterparts, the CISC (Complex Instruction Set Computer) processors. CISC processors support a large number of instructions and complex addressing modes. This results in a complicated decode unit and slower execution of instructions.

A good example of CISC architecture is Digital Equipment Corporation’s VAX 11/780. Introduced in 1978 as a successor to the PDP-11, the VAX has a large instruction set of over 300 instructions and a large number of complex addressing modes. A VAX instruction can vary from a single byte to maximum of 57 bytes. These factors combine to make the VAX’s control unit very large and bulky. The complexity of the decode unit increases the size of the microcode to hundreds of Kilobytes. Execution of an instruction becomes slow as more time is spent decoding it and less actually executing it [29]. RISC was conceived in an effort to overcome
these problems. RISC ideology optimizes the instruction set in order to include only those instructions that are frequently executed and relegate the rest to software routines. In this way, RISCs get their name that is Reduced Instruction Set Computers. Following are some of the principles followed by RISC designers [11]:

1. A small number of instructions (hardwired).

2. Single cycle execution of most instructions.

3. A LOAD/STORE architecture.

4. Fixed format instructions.

5. A large number of registers.

6. No microcode.

7. Small number of addressing modes.

8. Triadic register addressing (non-destructive register operations).

9. Many levels of memory hierarchy (caching).


11. Highly pipelined.

Table 2.1: Design and layout effort for some CPUs. [33]

<table>
<thead>
<tr>
<th>CPU</th>
<th>Transistors (thousands)</th>
<th>Design (Person-Months)</th>
<th>Layout (Person-Months)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISCI</td>
<td>44</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>RISCII</td>
<td>41</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>M68000</td>
<td>68</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>Z8000</td>
<td>18</td>
<td>60</td>
<td>70</td>
</tr>
<tr>
<td>iAPx-432</td>
<td>110</td>
<td>170</td>
<td>90</td>
</tr>
</tbody>
</table>

RISC architecture represent a radical change from conventional architecture design as embodied by CISC processors. Due to their simple design and shorter clock cycle, RISC processors are able to take advantage of new technologies. Table 2.1 compares design time of some RISC and CISC processors.

Currently, a lot of research is going on in the field of RISC architecture in order to take advantage of their superior performance. This, however, involves a number of important issues. The compiler of a RISC machine is difficult to design since it must do a lot optimization. In fact, RISC technology came into being due to the advancement in optimizing compilers. Development of compiler techniques have made it possible to achieve high performance with RISC machines. Good RISC compilers usually perform loop unrolling, pipeline scheduling, copy propagation, strength reduction, branch offset optimization etc [42]. See Table 2.2 for a comparison of RISC and CISC design philosophy.

RISC processors may be used in the design of real-time systems due to their single cycle execution of instructions and high performance. The instruction sets of RISC
processors are designed in such a way that at least 75 percent of the instructions execute in a single cycle. RISC technology is well suited for real-time applications since increased computing performance linearly improves real-time performance. Further [9] has found that benchmarking RISC against CISC in single parameter real-time functions like context switch time, preemption time, total interrupt handling time, etc., resulted in RISC being twice as fast. A reference of performance is a product of number of instructions executed, average number of machine cycles needed to execute one instruction and the cycle time. In equation form:

\[
\frac{\text{time}_{\text{program}}}{\text{instructions}_{\text{program}}} \times \frac{\text{clockcycles}_{\text{instruction}}}{\text{clockcycles}_{\text{cycle}}} \times \frac{\text{time}_{\text{cycle}}}{\text{clockcycles}_{\text{cycle}}}
\]

Due to a reduced instruction set and simple architecture both terms (2) and (3) in the above expression can be improved. This resulting processor speed is usable in real-time systems. In fact, pipelining can further improve throughput and thus term (2).
2.1.2 Compilers

Optimizing compilers and RISC architecture share a close relationship. Trends in programming real-time applications show the growing urge of high-level language usage. The high-level language programmer is using variables without regard to their mapping onto registers. this is the compilers responsibility to optimize their use. This requires complex allocation and optimization algorithms. Compilers do their best optimization with RISC architectures while RISC processors rely on compilers to obtain their full performance capabilities. A compiler works best with instructions which perform simple, well-defined operations. This close relationship between RISC and optimizing compilers is evident from the following:

- Register allocation: The compiler allocates processor registers to hold frequently used data and reduce LOAD/STORE operations. Since RISC processors usually have a large register file (=32), a good compiler can perform reasonable allocation.

- Redundancy elimination: The compiler tries to reuse results and eliminate redundant operations.

- Loop optimization: Loops are optimized by having unchanging variables and expressions moved outside them.

- Loop unrolling: The compiler replaces the iterations of a loop with a non-iterated straight-line code. The basic idea is to unroll the loop in order to
uncover loop-carried dependencies which may allow several iterations to overlap in execution.

- Slow operations replace: The compiler searches for situations where slow operations like multiply and divide (which take multiple cycles to execute) can be replaced with faster and simpler operations such as shift and add instructions.

- Strength reduction: In this technique, resource expensive operations are replaced with cheaper ones.

- Pipeline scheduling: The compiler schedules and reorganizes instructions in such a manner that pipeline delay slots are filled with useful instructions or in the worst case the compiler can always insert NOP's (no operation instructions).

2.1.3 Interrupt handling in real-time

In real-time systems, response time to an interrupt is an important factor governing performance. Whenever an interrupt occurs, the processor finishes executing its current instruction, saves the processor state and starts responding to the interrupt. Since most RISC instructions execute in a single cycle, the RISC CPU can, on the average, respond to an interrupt within one clock cycle, thus leading to shorter interrupt latency times. Usually a context switch saves a large number of registers before a task switch. One possibility is by restricting the usage of registers within
interrupt service routines [35]. On the other hand, CISC processors have variable size instructions that take multiple cycles to execute and thus lead to longer interrupt latency.

The simpler architectures of RISC allow higher clock rates and thus higher performance. However, in applications that have poor data locality, frequent interrupts, branches, context switches, cache misses and pipeline flushes RISC processors will have small performance gain over their CISC counterparts [42].

2.1.4 Cache memory in real-time

A number of techniques to design high throughput real-time systems with deterministic behavior have been reported. Real-time systems require some important design considerations. Real-time environments impose tight timing constraints on system behavior. Usefulness of a result is a function of both it’s accuracy and the time at which it is produced. This time value of a result is expressed as a constraint which if missed can cause system failure. Applications of real-time systems or embedded systems are in high speed communications, early warning defense systems, avionics, space exploration etc. Important considerations of embedded systems are cost, size, code compatibility and interrupt handling among others. Due to these constraints, embedded systems have to do away with power hungry functions like floating point units and memory management units.

Currently, a lot of research is being done in order to improve real-time systems
that exhibit unpredictable behavior [2, 9, 20, 26, 35, 40, 42]. Unpredictability in the
system may be due to pipelining, caching, virtual memory, context switching etc.
However, since cached systems can on average provide a 50 percent reduction in cycle
time, it is desirable to take advantage of cache memory. A low cost cache memory
system is proposed by the name of SMART (Strategic Memory Allocation for Real-
Time). This approach is attractive since it allows the system to operate within
timing constraints and provides guaranteed task execution times, while enjoying the
benefits of a high speed cached system [20].

Research has shown that the full Harvard architecture is a good candidate for
real-time applications since it is fast and easily modeled but it is costly to implement
and thus is infeasible for real-time systems. It has been reported that the single
stage Harvard architecture offers nearly the same performance at a much lower
cost [33]. Another technique may be to lock time critical data in the cache at the
expense of normal data. Another solution for overcoming cache drawback consists
of incorporating high-speed local memories into a RISC multiprocessor system, in
which all critical tasks can be locked. MC88000 designers recommend use of a
set-associative cache memory in order to allow permanent loading of speed critical
sections of code to remain in main memory [9]. The cache drawback may affect the
worst-case response time. In real-time applications the most important requirement
is to meet the deadline for a task. The overall time consists of the sum of response
time and task execution time. The overall time has to be predictable. If the response
time is slow due to cache drawback, a high computation RISC processor will make up for the time and meet the deadline.

2.1.5 Register file in real-time

The size of the register file plays an important role in determining the performance of a real-time system. One drawback of using a RISC processor for designing real-time applications is a large register file. All RISC machines employ a large number of registers in order to support their LOAD/STORE architectures. A large register file will significantly affect the context switch time since the register state must be saved whenever an interrupt occurs. The Motorola MC88000 uses a register scoreboard technique, thus allowing all instruction units to share the register file. This can improve context switching since all registers do not have to be saved as often. Similarly, the AM29000 processor from AMD employs a similar technique whereby all registers do not have to be saved during a context switch.

2.1.6 Memory requirements

Using RISC processors in real-time systems has two disadvantages. Firstly, due to a reduced instruction set the CPU memory traffic is higher since more work has to be done as compared to CISC. This requires a high speed memory system whose cost is usually too large for embedded real-time applications. Secondly, research shows that RISC code on the average occupies 30 percent more code and generates twice
as much bus traffic than a machine with a complex instruction set [5].

2.1.7 Real-time languages

A number of languages designed for real-time systems have been reported in the literature. Prominent among these are ADA, Pearl, FORTH and a few others. Probably the most important real-time language is ADA since it is very strongly supported by DOD (Department of Defense). ADA is a very powerful language and supports a large number of real-time features. Foremost among these is support of bit processing, file handling, task scheduling and controllability, synchronization mechanisms, resource reservation and allocation, interrupt handling, time scheduling, verification etc. Some of these features, however, are lacking in ADA. For instance, bit handling is indirectly possible, the only way for expressing time dependencies is by delaying the execution of tasks, absolute time specifications are impossible, facility of enabling or disabling interrupts is not available. According to one researcher, in order to be considered a full fledged real-time language, ADA must support the above listed features [12].

2.1.8 Architectural support of real-time systems

In order to satisfy predictability requirements of real-time systems a number of approaches are available. The underlying architecture of the system must provide predictable behavior. An architecture to support hard real-time applications is pro-
posed [2]. The aim in this research is to achieve predictable temporal behavior (i.e. a process tends to refer elements already referred recently). Program constructs leading to this concept are loops, temporary variables and process stacks. Predictable behavior is achieved by supporting inherent parallelism of instruction execution instead of pipelining. Caching is replaced by a separate, directly addressable program memory located close to the instruction fetch unit. To support high-level language programming a local memory file replaces the set of registers. This is done so as to remove allocation and optimization problems faced by the compiler. Behavioral predictability is not necessarily endangered though.

2.2 System architectures

Colnaric in [26] has extensively dealt with the issue of architectural support of real-time systems. Similarly to the processor architecture considerations like register file, cache memory and pipelining, in universal system architecture design some features were implemented to improve its average performance. These features could also introduce undesirable consequences which make process execution time prediction difficult or even impossible. Some of these are discussed below.
2.2.1 Direct Memory Access (DMA)

Since general processor is ineffective in transferring blocks of data, a direct memory access (DMA) technique was designed, employing dedicated controllers to transfer data between memory and massive peripheral devices. During the DMA transfer, the general processor is halted or performing operations on local data, according to its design. With respect to the delay due to the DMA transfer, there are two general modes of DMA operation, the cycle stealing and the burst mode. DMA controller operating in the cycle stealing mode is literally stealing bus cycles from the processor, while in the second mode the processor is stopped until the DMA transfer is completed. Thus, in both cases program execution is delayed for a period which depends on the length of the block and the bus protocol. Although during block mode operating DMA transfer processor has no control over the system until the transfer is completed, it seems more suitable when predictability is the main goal. If block length and data transfer initiation instant are known at the compile time, the delay can be calculated and considered in the program execution time estimation. The same prediction in the cycle stealing mode would be more complicated. Apart from that, block transfer is also faster because bus arbitration is only done once. However, precautions have to be taken to enable a processor, whose operation was suspended by DMA operation, to react in the case of catastrophic events.
2.2.2 Virtual addressing

To cope with limited amount of memory available in computer systems, a virtual addressing technique was developed. Its main principle is saving complete pages of memory on the mass storage devices. As data on the page are needed, the latter is loaded into the memory and the operation is performed. When data from another page are requested, the previous page is saved and overlaid by the new contents. Special memory management units were developed for managing the virtual addresses and mapping them onto the physical memory.

Data access time is very much dependent on whether the location is already in memory or the page has to be loaded first. To determine that, a complex program analyzer would be necessary. Even then, restrictions would have to be imposed. For example, register indirect addressing modes should only access data on the same page, what is difficult to assure when using a high level programming language.

From the above it can be concluded that virtual addressing should be renounced in hard real-time applications.

2.2.3 Data transfer protocols

Data transfer via microcomputer busses also requires some consideration. Synchronous data transfer protocols by definition ensure predictable data transfer times. Asynchronous ones are more flexible, however, their operation must be exactly de-
fined and followed by all communicating devices. Particular attention is to be paid in shared-bus systems. The only possible way to guarantee realistic data transfer times seems synchronous operation of all potential bus masters.

In distributed systems data are transmitted over local area networks which often operate non-deterministically. For example, broadly used Ethernet is based on the carrier sense multiple access with collision detection (CSMA/CD) bus protocol. If a collision on the bus is detected, all competing devices relinquish the operation and try again after an arbitrarily long delay. If they by any chance chose the same delay, the collision happens again. Thus, it is not possible to guarantee the data transfer time.

Currently, a large number of companies are designing and implementing RISC architecture in numerous products ranging from embedded systems to superscalar workstations. Comment on the merits and demerits of RISC and CISC machines can lead to unending debates especially when it concerns benchmarking. RISC, however, is a relatively new field that promises a good future. In [13], a good background of RISC architecture, the current scenario and what the future holds is presented.

2.3 Modeling in VHDL

VHDL is an IEEE standard for modeling complex digital systems at behavioral, dataflow and structural levels [16]. VHDL is used to convert a design idea into an
abstract behavioral level which can be further partitioned into dataflow and structural levels. The structural level is the lowest level and the most detailed description of the actual hardware. VHDL allows the user to keep the design technology independent by using generic statements. VHDL also supports concurrent execution which is useful for modeling hardware [24]. VHDL allows the user to specify architectures at all three levels for a design entity. For example, an adder can have a number of architectures, that is, a behavioral, a dataflow, or a structural one among others. Customized design libraries can be shared among designers in addition to the standard libraries that support VHDL functions, types etc. Good references can be found in [25] and [28].

VHDL is extensively being used for modeling processors, comparison and performance purposes, top-down modeling and as a front-end specification language in high-level synthesis systems. VHDL has extensively been used to model complex digital systems. Modeling of an address sequencer is described in [17]. The chip is the AM2910 that generates the next microprogram address depending on four previous states. A SDC data collection chip designed at Urbana-Champaign has been documented in [13]. This report also discusses techniques for modeling data acquisition systems. A priority interrupt controller for the Motorola MC68008 is described in [44]. This chip has been designed with the help of a silicon compiler and a standard cell library. A VHDL model of a cache memory using a FIFO buffer algorithm has been modeled in [41]. In the digital signal processing area, VHDL
has been used to model a discrete cosine transform [6], a Fast Fourier Transform (FFT) hardware model used in image compression [1] and a custom DSP complex to magnitude circuit [22] among others. In these examples, the power of VHDL is demonstrated in converting an algorithm to implementation. VHDL has also proved useful in modeling various performance models. Modeling of an Intel 80960 to investigate the merits and demerits of register file and cache memory is carried out in [32] and influence of number of registers and cache memory size in affecting RISC processor performance is evaluated.
Chapter 3

Real-time systems

“A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure.” [23]

Real-time systems are characterized by the fact that the correctness of a system depends on the logical result of a computation as well as the time at which it is produced. Nowadays, nearly all real-time systems are designed for special purposes due to which they are complex and costly.

Real-time systems are of three types. A soft real-time system may be a system like a word processing program. Failure to respond instantly will not be catastrophic. However, performance will be degraded and the system will be a torture to use. On the other hand, in some systems, failure to meet timing constraint results in system failure. These systems are called hard real-time systems. Hard real-time systems put a severe constraint on the performance of a system. The so called embedded systems.
that is, systems in which the computer forms an integral part of some system are in this category. Typical example is a control system in which the computer is in the feedback loop. A control algorithm is designed for a specific sampling rate and if the computation is not carried out at this rate then the calculated control value will be wrong. A further example is a system that has to record events in real-time. If two events occur and the computer fails to register the first one before the second one occurs, only one event will be recorded. In this system the minimum separation time between events will be specified and that will form the maximum allowable response time for the real-time computer system. Typical examples of such systems are aircraft avionics, nuclear power plant control etc. The literature also mentions firm real-time systems as systems that have strict timing constraints but can tolerate a low probability of error without failing [23].

In critical real-time systems, average or peak values are of no interest to the system designer. Since strict deadlines must be met, only those values are considered that can be guaranteed under the most demanding conditions. Features like a reduced instruction set, pipelining, cache memory etc., are based on statistical models and do not necessarily contribute to worst case performance. The external events are asynchronous and usually irregularly spaced. There must, however, be some known upper limit to their frequency and number if the system is to be correctly designed. At the hardware level these events typically make their presence known either by interrupts or by modifying a data register which is polled by software.
Table 1 shows the different types of real-time applications and their characteristics. This table is taken from [10]. A real-time system must deal with various types of delays which originate from the following.

3.0.1 Communication delays

These delays can be divided into propagation delays, transfer delays and access delays. Propagation delays are due to the finite speed of electrical or optical links. Transfer delays depend on the bit rate of a link and the finite bandwidth of the communication links. Access delays are due to the arbitration and waiting phases when the medium is shared.

3.0.2 Computation delays

These are due to the limited speed of the processor. For a given processor, the computation delay depends on the number of instructions and on the control structure (loops, branches) of the program. The computation delay is a function of the state of the variables.

3.0.3 Synchronization delays

Although processing power or link capacity may be available, a task may be waiting for a resource to become free, or for another task to produce data.
3.1 Desirable features of real-time systems

Real-time systems require some special design considerations in order to perform correctly. For example, a real-time system requires guaranteed response to external events within an absolute time deadline. An analysis of the event response for a given RISC based system is complicated. The problem is that the total response time is a sum of a variety of separate delays. There is the delay between the event and the recognition of the interrupt by the CPU, otherwise known as the interrupt latency time. There is the delay between the actual start of the interrupt sequence and the entry into the interrupt service routine.

In addition, there is the time the CPU requires to change context, to save and reload registers, invalidate and reload cache lines, replace FPU and MMU registers and flush write buffers. Finally, there is the time to execute the service routine. Also, real-time systems cannot be satisfied with good average response, rather must be able to specify a worst case response time. The correct estimation of delays is fundamental to any real-time system. These delays limit the performance of the system, and measures have to be taken to reduce them and suppress bottlenecks by using say faster processors, better organization or redundancy.

There are two schools of thought depending on whether or not the delays can be estimated, i.e., whether delays are deterministic or random. If the delays can be given a worst-case upper bound (according to the pessimistic school), then the system can
<table>
<thead>
<tr>
<th>Type of application</th>
<th>Response time requirements</th>
<th>Human interface</th>
<th>Design complexity</th>
<th>Functionality</th>
<th>Number of tasks handled</th>
<th>File system required</th>
<th>Dedicated or not?</th>
<th>Network implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central, district, data acquisition, process control, testing</td>
<td>Less than 1 millisecond</td>
<td>Complex</td>
<td>Low</td>
<td>Less than 5</td>
<td>Yes</td>
<td>Unusual</td>
<td>Ethernet, TCP/IP, UDP, IPX/SPX, OSI, ATM</td>
<td></td>
</tr>
<tr>
<td>Control, automation, robotics, process control, testing</td>
<td>Less than 1 millisecond</td>
<td>Complex</td>
<td>Medium</td>
<td>Medium</td>
<td>More than 50</td>
<td>Yes</td>
<td>Address a range of applications, interconnected with local network communication system</td>
<td></td>
</tr>
<tr>
<td>Computer data organization</td>
<td>Less than 1 millisecond</td>
<td>Graphical</td>
<td>High</td>
<td>Large</td>
<td>Many</td>
<td>Also handles many processes</td>
<td>Yes</td>
<td>Address a range of applications, interconnected with local network communication system</td>
</tr>
<tr>
<td>Medical imaging</td>
<td>Less than 1 millisecond</td>
<td>Graphical</td>
<td>High</td>
<td>Large</td>
<td>Many</td>
<td>Also handles many processes</td>
<td>Yes</td>
<td>Simple application with many components</td>
</tr>
<tr>
<td>Automobile, air traffic control</td>
<td>Less than 1 millisecond</td>
<td>Graphical, visual display, video display, types of display</td>
<td>High</td>
<td>Large</td>
<td>Many more than 50</td>
<td>Also handles many processes</td>
<td>Yes</td>
<td>Simple application with many components</td>
</tr>
<tr>
<td>Testing, machine control</td>
<td>Less than 1 millisecond</td>
<td>Graphical, visual display, video display, types of display</td>
<td>High</td>
<td>Large</td>
<td>Many</td>
<td>Also handles many processes</td>
<td>Yes</td>
<td>Address a range of applications, interconnected with local network communication system</td>
</tr>
<tr>
<td>Cell control, monitoring, testing, reporting</td>
<td>50 milliseconds to more than 1 second</td>
<td>Graphical, visual display, video display, types of display</td>
<td>High</td>
<td>Large</td>
<td>Many</td>
<td>Also handles many processes</td>
<td>Yes</td>
<td>Address a range of applications, interconnected with local network communication system</td>
</tr>
<tr>
<td>Communication</td>
<td>Less than 1 millisecond to 10 milliseconds, depending on the application</td>
<td>Usually complex, character-oriented, command line interface</td>
<td>High</td>
<td>Large</td>
<td>Many</td>
<td>Also handles many processes and controllers</td>
<td>Yes</td>
<td>Address a range of applications, interconnected with local network communication system</td>
</tr>
<tr>
<td>Flight control monitoring, control, maintenance</td>
<td>10 milliseconds to several seconds</td>
<td>Graphical, visual display, video display, types of display</td>
<td>High</td>
<td>Large</td>
<td>Many</td>
<td>Also handles many processes and controllers</td>
<td>Yes</td>
<td>Address a range of applications, interconnected with local network communication system</td>
</tr>
<tr>
<td>Online transaction processing</td>
<td>Capacity for high average throughput</td>
<td>Graphical, visual display, video display, types of display</td>
<td>High</td>
<td>Large</td>
<td>Many</td>
<td>Also handles many processes and controllers</td>
<td>Yes</td>
<td>Address a range of applications, interconnected with local network communication system</td>
</tr>
</tbody>
</table>
guarantee real-time response under any circumstances using cyclic or polled systems. Due to the above mentioned reasons, design of a real-time system requires careful specification and understanding of the problem. In order to understand the factors involved in the implementation of a real-time system, it is worthwhile to investigate features that must be satisfied in a real-time system or should be available in a processor that is part of a real-time system design. These factors are discussed below.

3.1.1 Predictability

For a design engineer, predictability is the foremost goal while designing a real-time system. Predictability is the ability to calculate the elapsed time between an external event and the system's response. Predictability of the temporal behavior of task executions is a precondition for schedulability analysis, which is the crucial point of interest in the design of hard real-time systems. In order to be able to predict a program's temporal behavior, every feature of the program and of the underlying computing system must behave predictably. Once a real-time system is designed to give deterministic response times, its performance can be improved. It is important that the timing requirement of each task of the real-time system be satisfied otherwise probability of failure will increase.

In order to accomplish this, it is imperative that resource scheduling be judicious. Only then is it possible to design a system whose timing behavior is deterministic.
and the system is understandable, maintainable and above all predictable. With the advent of high speed RISC processors, there is a general misconception that problems of real-time systems will be solved. High computational speed alone does not guarantee reliable and predictable operation [34]. RISC processors achieve high performance over their CISC counterparts by employing techniques like register-register computation, pipelining, small instruction sets, simple decode logic and cache memories. Since RISC processors have a small and simple instruction set, the memory bandwidth requirement is very high. In order to satisfy it, large on-chip cache memories are necessary.

RISC processors are designed to operate on an uninterrupted stream of data. A fast, well optimized cache can supply data at high rates. However, a cache provides only an average or statistical solution since sometimes it suffers performance loss due to "misses". The performance of a cache is based on its "hit" ratio. Whenever the cache is accessed and data is not found, it is termed a "miss". So the higher the hit ratio, better is the performance of the cache and consequently of the processor. But the hit/miss ratio cannot be predicted before hand whereas real-time applications require deterministic response times and a worst case prediction. In order to take advantage of RISC performance with cache memory, some clever techniques must be employed. These may include locking of specific routines into or out of cache memory to ensure deterministic behavior or storing critical interrupt routines in cache so that the designer may be confident of their timing behavior. If the program
code is in ROM instead of RAM then techniques like overlaying are not required. In addition, shadow RAM and interleaving should also be renounced especially for real-time systems.

3.1.2 Interrupt handling capability

Interrupts are a major factor to consider in the design of real-time systems. Interrupts are of two types: internal and external. Four components determine interrupt latency namely:

- execution time of the interrupted instruction.

- speed of external memory.

- priority level of interrupt.

- and total number of interrupts received.

For most CISC processors the instruction set component is largest since instructions take multiple cycles to execute. In real-time applications, the system must provide fast interrupt capability. Due to their finer instruction granularity, RISC processors enjoy a performance advantage while handling interrupts. A RISC processor’s shorter clock cycle enables it to respond more quickly to an interrupt. This factor minimizes instruction restarts, thereby improving performance and simplifying memory system design. However, one of the most important factor influencing
interrupt performance and hence overall system performance is a high interrupt latency. RISC processors employ a large number of registers in order to support their LOAD/STORE architecture. In the event of an interrupt, the processor state must be saved. A large number of registers means a large context must be saved. Hence it takes more time to save the processor state. Usual interrupt sources are timer counter overflow/underflows, serial channel byte received or transmit buffer empty, hardware pin inputs. A/D conversion ready etc.

In order to reduce interrupt latency, many methods have been reported. Literature reports using stack architectures to improve interrupt latency [19], register allocation across tasks [18], switching processor modes [38] use of processor status word etc. Most processors studied do support interrupt handling in one manner or the other. It, however, remains one of the most important factors in real-time system design.

3.1.3 High speed I/O

Real-time systems must provide high speed I/O capabilities since they require fast data acquisition, fast computation and fast data output. Usually, I/O is an afterthought. Or the I/O subsystem is designed as a special purpose CPU trying to maximize it's performance without regard to other elements in the system. Performance of CPUs improve at 50 to 100 percent per year and if I/O does not improve at the same rate, then every computation will eventually become I/O bound. Measure
of performance is throughput (tasks per hour) rather than response time. The I/O subsystem depends on a number of factors like CPU, cache memory, main memory, I/O device, I/O channel etc. The art of I/O design is to construct a system that is able to match the speed of all its components and to have a balanced system. In addition one must take into account the type of I/O device to connect to the computer system and the number of devices to connect [13]. Some of the RISC machines studied support high speed I/O by incorporating techniques like multiple bussing, burst mode transfers, I/O processors etc. Performance increases have resulted from enhancements made to the processor. With the advent of high speed processors like RISC, many systems spend more time waiting for data to be transferred from the disk than they do processing the information. Thus, it is essential to consider improving the data I/O throughput in order to match processor performance as close as possible.

3.1.4 Fault tolerance

The definition of the real-time operating mode has important consequences for the dependability of real-time systems, since the demanded permanent readiness of computers working in this mode can only be provided by fault-tolerant and especially with respect to inadequate handling robust systems. This issue is very important in the design of real-time systems. No matter how well a system is designed, there is always a non-zero probability of error. Since computers are being used in critical
roles like avionics, robotics, space exploration etc., these systems must be designed to be highly reliable and fault tolerant. Some form of fault tolerance must be incorporated in order to perform critical functions in face of component failure. Following are the important properties of fault tolerant systems:

- no erroneous output,
- integrity of data,
- continuously available service,
- recovery from system failure,
- safe control of critical systems,
- infrequent system failures and
- low maintenance costs.

It must be borne in mind that real-time systems may also fail if the system does not execute its critical workload within the time constraint [21]. In case of hard real-time systems, severe consequences will result due to malfunction. Hence, special care must be taken to ensure the system is ultra-reliable. Reliability can be improved by extensive test methods, incorporating redundancy or self checking circuitry, system monitors or by employing a formal verification procedure. In addition, hardware support of error detection and possible recovery from failure can
also be incorporated. Error control coding is being used to detect and correct errors in memory. For hard real-time systems, the standard for reliability is very high. For example, NASA has suggested that computers used in civilian fly-by-wire aircraft have a failure probability of no more than $10^{-9}$ per hour [21].

In addition to the above mentioned features, there are some other points worth considering. A real-time processor must have timers, counters, hardware multipliers and good bit manipulation support. Timers are important in real-time systems design since they help record events, trigger interrupts etc. Timers like watchdog, programmable and real-time clock are usually implemented in processors. Watchdog timers ensure that certain devices are serviced at regular intervals and that the CPU is functioning properly. If the CPU is required to reset a watchdog timer every second and a watchdog timer overflow occurs, then either the CPU has malfunctioned or the 1 sec cycle has time overloaded. Programmable timers permit precise measurement of a time interval.

A real-time clock's function is to generate periodic interrupts. A real-time clock permits us to control the timing of several events to the nearest tick time, where one tick is much slower than the execution speed of the micro-controller. Timer/counters may be configured to count either a divided version of the system clock or transitions on an input pin. Current count value is often accessible as a register variable and counter overflows can trigger system interrupts. In case the system is actually an embedded system that is part of a larger real-time system, then size, weight, cost
Table 3.1: Desirable features of real-time systems on some CPU's.

<table>
<thead>
<tr>
<th>FEATURES</th>
<th>68040</th>
<th>ARM</th>
<th>88100</th>
<th>i960</th>
<th>80196</th>
<th>AM29k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>CISC</td>
<td>RISC</td>
<td>RISC</td>
<td>RISC</td>
<td>CISC</td>
<td>RISC</td>
</tr>
<tr>
<td>Predictability</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Interrupts</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Data I/O</td>
<td>high</td>
<td>25Mb/s</td>
<td>high</td>
<td>53Mb/s</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Fault Tolerant</td>
<td>yes</td>
<td>Exep.</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>Exep.</td>
</tr>
<tr>
<td>(MIPS)@Mhz</td>
<td>20/25</td>
<td>6/12</td>
<td>15/20</td>
<td>7.5/20</td>
<td>n/a</td>
<td>17/25</td>
</tr>
<tr>
<td>No. of Reg.</td>
<td>16</td>
<td>27</td>
<td>32</td>
<td>36</td>
<td>7</td>
<td>192</td>
</tr>
<tr>
<td>Bit Ops</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Med</td>
<td>Low</td>
</tr>
<tr>
<td>Multiplier</td>
<td>FPU</td>
<td>Booth</td>
<td>FPU</td>
<td>FPU</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Timers</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

and power consumption must also be considered. A comparative study of different processors reveals that most of the processors currently available have nearly the same features. However, for real-time applications, some special features are required, both on the architectural as well as instruction set levels. A brief summary of this study is given in Table 3.1.

A study was carried out in order to determine how the desirable real-time features are supported. Two levels are defined: the instruction set and the architectural level. This is shown in Table 3.2.

### 3.2 Real-time operating systems

Obviously, a real-time operating system is the heart of a real-time system, forming a layer between the programmer and the base machine. One of the most impor-
Table 3.2: R-T features support at architectural and instruction set level.

<table>
<thead>
<tr>
<th>FEATURES</th>
<th>Architectural Level</th>
<th>Instruction Set Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictability</td>
<td>uncached, unpipelined systems</td>
<td>fixed execution time</td>
</tr>
<tr>
<td>Interrupts</td>
<td>interrupt controller</td>
<td>simple instructions</td>
</tr>
<tr>
<td>Data I/O</td>
<td>I/O controller</td>
<td>I/O, memory mapped</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>built-in test, traps</td>
<td>exception handling</td>
</tr>
<tr>
<td>(MIPS)@Mhz</td>
<td>simple design, shorter clock</td>
<td>reduced instructions set</td>
</tr>
<tr>
<td>No. of Reg.</td>
<td>Load/Store architecture</td>
<td>no. of bits allocated</td>
</tr>
<tr>
<td>Bit Ops</td>
<td>barrel shifter</td>
<td>bit manipulation instructions</td>
</tr>
<tr>
<td>Multiplier</td>
<td>FPU</td>
<td>multiply-add instruction</td>
</tr>
<tr>
<td>Timers</td>
<td>special registers</td>
<td>special instructions</td>
</tr>
</tbody>
</table>

The main goals of a real-time operating system is to manage a set of resources: CPUs, I/O devices, memory, files and others. In most systems, real-time performance is determined by the behavior of the scheduling algorithms used in the operating system’s resource manager. Many real-time scheduling algorithms have been developed, including fixed-priority scheduling, shortest-process-time scheduling, shortest-slack-time scheduling, earliest-deadline scheduling and others [10].

3.3 Real-time features available in commercial processors

In this section we will take a look at a number of commercially available RISC and CISC based processors. The objective is to find out the extent of real-time support in these processors.
3.3.1 Motorola 68040

A CISC, the 68040 is the high end processor of the 68000 family. Most of the instructions execute in one clock cycle since the integer unit (IU) is pipelined. The IU's ALU cycle time is matched to the cache access time. The Control Unit (CU) can access each cache once per clock cycle and is able to fix the peak execution rate to ALU cycle rate. The 68040 has a six stage pipeline. An instruction prefetch, a program counter calculation and decoding, effective address calculate, operand fetch, data execution and finally write back stage. The 68040 has 18 addressing modes. In addition there are a User mode and a Supervisor mode. The Supervisor mode has access to all registers. The 68040 handles interrupts on a priority level basis. Assuming zero-wait state memory and that the interrupted instruction completes on the following cycle, interrupt latency for an individual interrupt is 24 clocks plus 2 clocks for each register that's saved.

The 68040 has three pins for interrupt handling. They are: IPL2-IPL0 which provide an encoded interrupt level to the processor, IPEND which signals a pending interrupt and AVEC which is used during an interrupt acknowledge transfer to request internal generation of the vector number. All the three pins are active low. External interrupts are a particular case of exception. Exceptions are recognized at each instruction boundary in the execute stage of the integer pipeline and force later instructions that have not yet reached the execution stage to be aborted.
Exception vectors are located in supervisor address space. The 68040 also provides bit manipulation operations like bit test, bit test and set, bit test and clear, bit test and change etc. All bit operations can be performed on either registers or memory. The bit number is specified as immediate data or in a register. Register operands are 32 bits long while memory operands are 8 bits long. The 68040 supports variable length bit-field operations on fields of 32 bits. An on-chip FPU provides the necessary hardware for multiplication. See [36] for more details.

3.3.2 ARM VL86C010

The Acorn RISC Machine is targeted at the real-time, process control, embedded computer applications. This processor is capable of 6 MIPS at 12Mhz. Interrupt handling is by two pins: IRQ and FIRQ i.e Interrupt request input and Fast interrupt request input. A software interrupt is used for getting into supervisor mode, usually to request a supervisor function. Normal interrupt mode has two dedicated registers: one for return link and one for stack pointer. The fast interrupt mode has seven dedicated registers to allow I/O data transfers to be implemented in fast interrupt software without any register saving. The interrupt handling is good, typically achieving a worst case interrupt latency of 22.5 clock cycles or 1.87μs at 12Mhz.

One class of instructions perform loads and saves of multiple registers, which has a use in context changes, in procedure entry and return and data copy operations. Procedure calls are handled by overlapping registers across the processor modes. The
user mode has 16 registers, supervisor mode 2 registers, fast interrupt 7 registers and normal interrupt 2 registers. R14 is used as a link register for both subroutine and interrupt context changes. The ARM achieves high data bandwidth by supporting page mode transfers directly from the DRAM. It obtains approximately 50 percent more bandwidth from the same memory than other processors do. ARM is optimized for close coupling to dynamic memories. Due to the relatively low speed of access of DRAMs, the entire execution phase (register read, shift, ALU, write) can operate in one clock cycle. Need for complex forwarding logic to avoid data race hazards is removed. The long cycle also allows more efficient use of the instruction bits and more formats. The ARM has 46 instructions, 3 addressing modes and 6 instruction formats.

ARM supports a skip architecture, that is, all instructions are conditional, they must be tested true before they execute. Unexecuted instructions require a single processor cycle and keep the 3-stage pipeline intact. This approach was taken as opposed to the delayed branch technique in order to simplify the virtual memory page fault recovery process. A total of 16 registers out of 27 are available to the programmer. The extra 9 registers become available to the processor during interrupts to simulate a Direct Memory Access channel (DMA), without needing to save any of the user's registers. In this way a faster interrupt handling is achieved. The processor to memory bandwidth is 18 Mbytes/s. An I/O controller, VL86C410, allows standard peripheral devices attached to a separate I/O bus to be interfaced
to ARM.

The I/O controller contains four 16-bit counter/timer circuits, two are general purpose timers and two are baud rate generators. Sixteen interrupt sources are supported. The timers have fully programmable start/reload values. In case the coprocessor is absent, the instructions trap and are emulated by software. All data processing instructions except multiply execute in one clock cycle. A $32 \times 32$ multiply requires a maximum of 16 clock cycles. A barrel shifter increases the performance of shift intensive applications like graphics manipulations. This processor is one of the smallest 32-bit RISC processor available. Bit manipulation operations are also supported [38].

3.3.3 Motorola 88100

The 88100 is the first RISC processor from Motorola. It has a Harvard type architecture and an instruction set of 51 instructions, 3 addressing modes and 3 instruction formats. It has built-in fault detection and performs high speed interrupt handling with minimum latency. Assertion of the INT pin indicates that an external interrupt has been requested. When an interrupt exception is processed, the 88100 freezes its execution context and then proceeds with exception processing using the interrupt exception vector. The worst case interrupt latency $n$ can be expressed in clock cycles as: $3 + \max(4 \times \text{maxmem}(d), \text{maxmem}(i)) + \text{maxmem}(i)$

The processor needs one clock cycle for internal synchronization, one cycle for in-
struction propagation and decode and one cycle for instruction execute. Maxmem(d) is the longest data memory access latency of which a maximum of four can be outstanding. Maxmem(i) is the maximum instruction memory access latency which can take up to a maximum of two cycles.

The 88100 does not automatically save the processor context state to memory as part of exception processing: the processor state is stored in appropriate shadow and exception time registers. Saving the necessary context to memory is the responsibility of the software exception handler. The processor provides bit manipulation operations like clear bit field, extract bit field, find first bit set, find first bit clear, etc. Among all the RISC processors studied, the 88100 has the best bit manipulation support. A 32-bit combinatorial multiplier is available. An integer multiply takes four cycles. All the data processing instructions execute in one clock cycle. All instructions are directly implemented in hardware. This processor implements register scoring which allows all instruction units to share the register file. This further improves context switching since registers need not be saved as often.

The 88100 provides no direct support for input and output devices. I/O devices are mapped into normal memory address space. Load and store instructions can access the devices [27].
3.3.4 Intel 80960

The i960 is specifically designed for embedded applications. Burst mode capability helps it achieve a maximum bus bandwidth of 53M bytes/s at 20Mhz. In this mode, the i960 can read or write up to four words per access. Register scoreboard is employed to execute multiple instructions in parallel with load operations. The i960 is equipped with bit and bit-field instructions, FPU and an on-chip interrupt controller. Overall cost is reduced by including a four-channel 32-bit DMA controller and a 31 level 248 source interrupt controller. The i960 also has on-board ROM. Routines in the ROM can run for tens of microseconds and can be quite damaging to interrupt response. To overcome this problem, the i960 has a counter that monitors how long an interrupt has been waiting for the end of the instruction. If the interrupt has a high enough priority to be recognized and has been waiting for 100 clocks, the processor will suspend the ROM-based instruction, store its state for later resumption, and service the interrupt. The on-chip interrupt hardware is designed to work with an external interrupt controller.

The i960 automatically handles interrupts on a priority basis and tracks pending interrupts through its on-chip interrupt controller. Bandwidth savers like register caches, dedicated on-chip scratchpad RAM and extensive write buffers help the i960 to lower cost of memory system. DMA controller increases I/O performance. Using the multiple execution units, execution speeds of 2-3 instructions per cycle
are achievable. Register windowing minimizes the number of registers that must be saved to external memory on a context switch. It takes only 4 cycles on the i960, following a context switch to give the new process a fresh set of registers by simply moving the stack pointer. The i960 has 8 addressing modes, 5 instruction formats and 85 instruction types. I/O is memory mapped so there are no I/O type instructions. A self test facility is provided. All the major functional blocks are tested at reset by a program in the micro-instruction ROM which takes 47,000 clock cycles. If a failure is located, the failure pin is asserted and execution ceases.

The i960 has built-in debug capabilities. There are two types of breakpoints: hardware and software and six different trace modes. It has an automatic mechanism to handle faults. There are ten fault types including trace, arithmetic and floating point. When the processor detects a fault it calls the appropriate fault handling routine and saves the state information to make efficient recovery possible. The processor then posts diagnostic information on the type of fault to a fault record. Hardware support for procedure call and return is provided by using four banks of sixteen local registers. For each procedure call the i960 allocates 16 local registers (R0-R15). Upto 3 procedure calls can be made without having to access the procedure stack resident in memory. Multiple sets of local registers are stored on chip. The register cache holds upto four or more active procedures and if a new one is called, the processor moves the oldest set of local registers in the register cache to a procedure stack in memory. Global register G15 is used as the frame pointer for
the procedure stack [3].

3.3.5 Intel 80196

The 80196 is a high performance 16-bit micro-controller from Intel. This is a register-to-register machine with a 230 byte register file. The register operations can control many of the peripherals available. These peripherals include a serial port, an A/D converter, PWM output, upto 48 I/O lines and a high speed I/O subsystem which has 2 16-bit timer/counters, an 8-level input capture FIFO and an 8-entry programmable output generator. Efficient I/O operations are possible by directly controlling the I/O through the Special Function Registers. The main benefits of this scheme are the ability to quickly change context, absence of accumulator bottleneck and fast throughput and I/O times. A watchdog timer (WDT) provides a means to recover gracefully from a software upset. When the watchdog is enabled it initiates a hardware reset unless the software clears it every 64K state times. The WDT is independent of the other components and timers of 80196.

Clock failure detection can be enabled to cause the RESET line to be pulled low if the oscillator falls below a certain frequency. Since the I/O lines are statically driven to a known state with RESET low, the system can be made to go to a safe condition if it fails. The 80196 supports 28 sources of interrupts. In addition to NMI, there is TRAP and Unimplemented Opcodes. Protection is provided against executing unimplemented opcodes by the unimplemented opcode interrupt. The
hardware reset instruction (RST) can cause a reset if program counter goes out of bounds. The memory region as well as the status of the majority of the chip is kept intact while the chip is in Powerdown mode [4].

3.3.6 AMD 29000

The AM29000 has 192 registers which is a large number as compared to other RISC machines. The local registers are organized as a stack cache. This is more effective than devoting the same amount of memory to an on-chip data cache. There are some advantages to this. The register file has three ports, which allows two operand reads and a single write in one cycle, while a data cache would require a single port. Secondly, the registers are accessed during instruction decode, so that the access time has no effect on performance due to overlapping. A data cache would require the execution of load and store instructions to read and write data, which is not the same with the register file. Finally, cache brings unpredictability to the system which is undesirable for real-time systems. In addition, the register file boosts real-time performance. On a task switch, saving all registers requires about 8\(\mu\)s. However, if the registers are partitioned among many tasks, a task switch could occur in less than 700\(\mu\)s. Procedure calls are handled by registers as well as the run-time stack in main memory.

The AM29000 has three non-multiplexed 32-bit buses, a bidirectional data bus, an address bus and an instruction bus coming from instruction memory. Overall
bus bandwidth is 240Mbytes/sec. A full Harvard architecture is implemented i.e separate instruction and data memories are available. A timer counter register (SPR8) contains the counter for the timer facility. Timer reload register (SPR9) maintains synchronization of the timer counter register, enables timer interrupts and maintains timer facility status information. The AM29000 has a burst mode in which one address triggers a burst of data cycles. In this mode the CPU can reach peak transfer rates of 100Mbytes/s. Data can be transferred between source and destination at 50Mbytes/s. The CPU has an on-chip branch-target cache (with a 60 percent hit rate) to prevent the CPU’s four stage pipeline from emptying during repeated branch instructions thus providing a gain in performance. The register file can be divided into 12 blocks of 16 registers. When each block is dedicated to a separate process, real-time latency can be minimized because little or no off-chip data movement is needed to perform the context switch necessary to service an interrupt.

The processor can be interrupted by signals received at pins INTR0-INTR3 or by the timer facility. The 29000 has priority and vector hardware on-chip that is designed to work with an external interrupt controller. The AM29000 averages 1.5 cycles/instruction. All but four AM29000 instructions execute in one clock cycle. Two multiple-cycle instructions are interrupt-return sequences, which cannot be interrupted. The remaining two, i.e, Load Multiple and Store Multiple are interruptible, so they do not have any adverse effect on interrupt latency. Bit operations
are also supported [18].

3.4 A graphical display of RT features in some RISC CPUs

In this section we graphically represent some desirable real-time features found in commercially available RISC processors. This method of representation is called Kiviat Graph and one is able to summarize the relevant architectural parameters in an attractive manner [7].
Figure 3.1: Kiviat graph of desirable RT features of Motorola 88100.
Figure 3.2: Kiviat graph of desirable RT features of Intel 80960.
Figure 3.3: Kiviat graph of desirable RT features of ACORN RISC machine VL86010.
Figure 3.4: Kiviat graph of desirable RT features of proposed processor.
3.5 Future of real-time systems

Real-time systems will gain importance as powerful processors are being used to design time critical systems. These systems will range from space exploration to high speed communication networks. A brief outlook on the future of real-time systems is shown in the table below. This information is taken from a study by Borko Furht and Wolfgang A. Halang [10].
### Table 2
Driving Forces Which Will Determine the Profile of the Real-Time Computer of the Next Decade

<table>
<thead>
<tr>
<th>Technological Area</th>
<th>Expectations by the Year 2000</th>
</tr>
</thead>
</table>
| VLSI Technology              | • 0.25 to 0.5 micron Gigascale integration  
• 100 million to 1 billion transistors per die                                              |
| Processor Technology         | • RISC like superscalar microprocessors on chip  
• Clock rates: 250 MHz  
• 1,000 MIPS per CPU  
• 2 to 4 processors on the chip: up to 4000 MIPS  
• Multi-level memory hierarchy on chip                                                         |
| Processor Architecture       | • Balanced performance CPU & I/O  
• 64-bit architecture  
• Architectural support for:  
  - real-time operating system (UNIX)  
  - real-time scheduling algorithms  
  - real-time languages  
  - fault-tolerance  
  - fast context switch |
| Real-Time System Architecture| • Distributed system with each node as a multiprocessor  
• Fast real-time communications                                                                |
| Real-Time Operating System   | • Standard real-time UNIX  
(several implementations will exist)  
• New dynamic scheduling algorithms  
• Enhanced real-time performance                                                              |
| Software Tools               | • Powerful real-time graphics users' interface  
(using voice, touch, etc.)  
• New real-time programming languages  
• Specification and verification techniques for real-time systems  
• Distributed real-time databases                                                            |
| Application Generators       | • Real-time expert system shells  
• Self-learning and adaptive processors  
• Neural network technology embedded in many applications                                     |
Chapter 4

Instruction set design

4.1 Instruction set design

A machine has three components: a set of commands, called the instruction set, a storage, and an interpretive mechanism. The combination of instruction set and storage defines the architecture of a machine. In order to design an instruction set, a designer must rely on instruction set studies, his own experience and some ingenuity.

In the past, there have been a number of studies on instruction set measurement and analysis. Instruction set studies are carried out in order to determine which language constructs are used most often and help the computer designer to implement an architecture and later to improve it. Two kinds of instruction set measurements are possible, that is, static and dynamic. Dynamic measurements are taken while the program is running while static analysis measure the instruction count from a
Table 4.1: Static analysis of ED and ROFF, word processors under UNIX.

<table>
<thead>
<tr>
<th>Instruction(ED)</th>
<th>Occurence</th>
<th>Percent</th>
<th>Instruction(ROFF)</th>
<th>Occurence</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>438</td>
<td>26</td>
<td>branches</td>
<td>412</td>
<td>18</td>
</tr>
<tr>
<td>branches</td>
<td>371</td>
<td>22</td>
<td>mov</td>
<td>378</td>
<td>16</td>
</tr>
<tr>
<td>JSR</td>
<td>224</td>
<td>13</td>
<td>JSR</td>
<td>278</td>
<td>12</td>
</tr>
<tr>
<td>CMP</td>
<td>135</td>
<td>8</td>
<td>CMP</td>
<td>210</td>
<td>9</td>
</tr>
<tr>
<td>TST</td>
<td>99</td>
<td>6</td>
<td>RTS</td>
<td>145</td>
<td>6</td>
</tr>
<tr>
<td>JMP</td>
<td>83</td>
<td>5</td>
<td>ADD</td>
<td>120</td>
<td>5</td>
</tr>
<tr>
<td>CLR</td>
<td>66</td>
<td>4</td>
<td>JMP</td>
<td>88</td>
<td>4</td>
</tr>
<tr>
<td>MOVB</td>
<td>63</td>
<td>4</td>
<td>CLR</td>
<td>84</td>
<td>4</td>
</tr>
<tr>
<td>ADD</td>
<td>59</td>
<td>4</td>
<td>TST</td>
<td>78</td>
<td>3</td>
</tr>
<tr>
<td>INC</td>
<td>18</td>
<td>1</td>
<td>MOVB</td>
<td>61</td>
<td>3</td>
</tr>
<tr>
<td>others</td>
<td>112</td>
<td>17</td>
<td>others</td>
<td>168</td>
<td>20</td>
</tr>
</tbody>
</table>

program listing. Dynamic measurements are more important as they are able to provide much more information than static analysis.

In the early 70's a lot of work was done in the field of instruction set analysis. A static analysis of two word processing tools ED and ROFF running under UNIX showed that most of the computation consisted of simple move statements, branches and procedure calls [43]. A summary of this study is given in Table 4.1. In [39], a case study was carried out on six commonly used VAX compilers for the following languages BASIC, Bliss, COBOL, FORTRAN, Pascal and PL/I. This was a dynamic analysis of the instruction usage in popular compilers. A summary of this study is shown in Table 4.2. A number of other studies have been carried out and all make the same observations. Stated simply, these studies show that for
Table 4.2: Instruction usage of six compilers on VAX-11.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>22</td>
</tr>
<tr>
<td>branches(cond)</td>
<td>11.8</td>
</tr>
<tr>
<td>CMP</td>
<td>10</td>
</tr>
<tr>
<td>MOVAB</td>
<td>4.4</td>
</tr>
<tr>
<td>branches(uncond)</td>
<td>4.3</td>
</tr>
<tr>
<td>Calls</td>
<td>2.4</td>
</tr>
<tr>
<td>RET</td>
<td>2.4</td>
</tr>
<tr>
<td>ADDL2</td>
<td>1.7</td>
</tr>
<tr>
<td>INC</td>
<td>1.4</td>
</tr>
</tbody>
</table>

For general purpose computing, most instructions are compiler generated instructions like LOAD and STORE, instructions that perform looping and branching which include comparisons, incrementing and decrementing and branches. Following these instructions are less frequent and expensive ones like procedure calls. Surprisingly less frequent are arithmetic and logical operations. The instruction set architecture of a computer is its appearance as seen from a machine language programmer’s point of view. The instruction set architecture includes data types, register set, the instruction set, interrupt and I/O structure.

The instruction set of a RISC processor is small, with a fixed instruction length and most of the instructions take one clock cycle to execute. The RISC concept resulted from statistical analysis of how a processor’s resources are utilized by software. It was observed that a large number of instructions were simple, even for CISC machines. The objective of this study is to propose a suitable instruction set
for real-time applications. Real-time systems find use in a number of application areas. Our emphasis is on embedded real-time applications. For this reason, some high level decisions were made in the beginning of the design process. For instance, power hungry function units like FPU and MMU were removed. In order to achieve deterministic behavior, cache memory and pipelining which introduce unpredictability in the system are not recommended in the architecture. These decisions ensure that the resulting design will be small, simple, cost-effective and achieve predictable behavior albeit at some loss of performance due to lack of cache memory and pipelining [2]. In addition, hardware support of multiplication and division is renounced for real-time systems and instead software routines are invoked when required.

The instruction set designer is faced with a very large set of options, in fact the number of choices are unbounded. Some simplifying assumptions must be made at the early stages of the design. The RISC instruction set is based on a small number of simple instructions. Figure 4.1 shows the instruction mix in a typical program. These statistics are taken from a program written for the Motorola 88100 RISC processor. It is clear that the frequency of LOADs, STOREs and Branches are much greater than logical and floating point operations. The RISC approach results in minimum decode complexity but higher memory bandwidth requirement. In case of an architecture without registers (an all memory architecture), 47 percent of the data references would be for temporary storage of intermediate results within the evaluation of expressions. The addition of only two or three registers through
the use of a register instruction format basically eliminates these references [8].

![Instruction mix in a typical program](image)

**Figure 4.1:** Instruction mix in a typical program.

Even the addition of a single instruction requires a justification by the designer in terms of performance gained. A RISC instruction set begins with simple instruction formats and a 32 bit instruction length. A LOAD/STORE model is proposed i.e. only these instructions are allowed access to memory while the rest of the instructions are register based. Two advantages accrue from this design philosophy. Firstly, registers are faster than main memory and secondly a good and efficient optimizing compiler can do a good job of register allocation. Registers are addressed with less amount of bits, they can hold variables which helps reduce traffic between memory and CPU, reduces the size of the program code and speeds up program execution. In
addition to these advantages using registers is more flexible in handling expressions than either accumulators or stacks.

The optimum number of registers is still an open research topic. A recent study of cache memory and register file trade-off showed that for a fixed cache size, increasing the register file size from 64 onwards did not result in appreciable gain in computation time [32]. Albeit, a single stage Harvard architecture is shown to be the most effective for real-time applications, due to small size and low cost considerations we are proposing the simple Von Neumann processor model.

Since the processor is targeted at real-time applications, a large number of logical and bit-manipulation operations are supported. An instruction set designed for real-time applications must provide high speed computation, fast I/O operation and extensive bit manipulation support. In embedded control systems the processor usually performs a monitoring job. In order to facilitate this task, a number of bit testing and manipulation operations are necessary. In the proposed instruction set, the group of bit operations is very powerful. It is possible to manipulate data in a number of ways. The instruction set provides the capability of ORing, ANDing, XORing specific bits of any two registers. The find bit instructions can locate the position of a zero or a one in the specified data. Similarly, other powerful instructions like clr b and set b enable the programmer to selectively set (1) or clear (0) a specified width of bits at specified positions in the data. In addition, the branch instructions can take a jump depending on the bit condition in the processor
status word. Probably the most critical activity in real-time systems is the testing, setting and clearing of bits in control/status registers.

The instruction set also provides a number of comparison instructions which are useful for comparing two incoming streams of data. These instructions set the PSW and the control instructions can test them and take action accordingly. Most of the commercially available RISC processors do not provide good bit-manipulation support, exception being the Motorola 88100. In addition, real-time systems need fast response to external stimuli, i.e. fast interrupt facility which is usually handled at the architectural level. RISC processors have an advantage: the longest instruction time is small which results in reduced overall response time. In addition to the small instruction set, the number of addressing modes is only three and there are only four instruction formats. The three addressing modes are register, immediate and indexed. The four instruction formats are register, immediate, bit and branch type as shown in Figure 4.2.

A good instruction set provides a balance between performance and instruction encoding. Each instruction of the set is designed to perform a simple and straightforward operation. The goal of RISC design is to achieve an execution rate of one instruction per cycle. Techniques that help achieve this goal are:

- instruction pipelines,
- load/store architecture,
Table 4.3: Addressing mode usage on three VAX compilers.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Bliss</th>
<th>FORTRAN</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>38.1</td>
<td>41.0</td>
<td>45.1</td>
</tr>
<tr>
<td>displacement</td>
<td>25.2</td>
<td>20.6</td>
<td>26.1</td>
</tr>
<tr>
<td>literal</td>
<td>18.5</td>
<td>18.2</td>
<td>15.5</td>
</tr>
<tr>
<td>indirect</td>
<td>8.6</td>
<td>9.1</td>
<td>2.0</td>
</tr>
<tr>
<td>autoincrement</td>
<td>3.3</td>
<td>4.1</td>
<td>2.6</td>
</tr>
<tr>
<td>index</td>
<td>3.0</td>
<td>5.6</td>
<td>7.6</td>
</tr>
<tr>
<td>Totals</td>
<td>96.7</td>
<td>98.6</td>
<td>98.9</td>
</tr>
</tbody>
</table>

- delay load instruction and

- delayed branch instruction.

4.2 Addressing modes

An addressing mode can specify a constant, a register or a location in memory. Addressing modes can be register, immediate or displacement based, or can be indexed, direct, scaled etc. This large number of addressing modes is usually found among CISC machines and hence complicates the decode stage. Studies have shown that immediate and displacement addressing dominate addressing mode usage [13]. In a study of VAX addressing, it was shown that two addressing modes namely register and displacement account for over 60 percent of the addressing used and four formats including literal and indirect accounted for over 90 percent of the addressing used [39]. For reference see Table 4.3. For a small number of addressing
modes, the addressing bits can be encoded in the opcode itself. The number of registers and addressing modes can have a significant impact on the instruction size since the register and addressing mode field may appear many times in a single instruction. Infact, the usual case for RISC machines is that the number of bits for encoding the register fields and the addressing modes is greater than that for the opcode. Thus, the designer must give careful thought to the specification of the addressing modes since they will affect the decode logic.

We have allocated two bits for the addressing mode for a maximum of four modes to be specified. We have proposed only three modes i.e register, immediate and indexed. The last mode i.e indexed actually refers to an extra register file located on chip. The size of this register file depends on the available space after the chip has been laid out. The addressing modes available are:

1. Register based: In register based mode, both operands are available from two source registers. As an example consider an add instruction:
   \[
   \text{ADD R4,R1,R2} \rightarrow \text{add the contents of R1 and R2 and store in R4.}
   \]

2. Immediate addressing: Immediate addressing allows one of the operands to be supplied in the instruction. An example of immediate addressing is:
   \[
   \text{SUB R5,R2,#35} \rightarrow \text{subtract the contents of register 2 from an immediate value, 35, and store the result in register 5.}
   \]

3. Register indexed: Indexed addressing allows the processor to index into an
on-chip register file in order to fetch the second operand. As an example:

\texttt{MUL R3, R4, @R7} \implies multiply the contents of register 4 and value in register pointed to by register 7. Store the result of the operation in register 3.

### 4.3 Instruction formats

The instruction format is associated with the addressing mode and determines the set of operations to be defined for that format. The four proposed formats for our RISC machine are the following:

- Register type.

- Immediate type.

- Bit manipulation type.

- Control type.

\texttt{OPCODE} is a six bit field, \texttt{DEST}, \texttt{SRC1}, \texttt{SRC2}, \texttt{COND}, \texttt{TARG}, \texttt{BIT1} and \texttt{BIT2} are five bits wide. \texttt{IMMED} is a 14-bit constant, \texttt{OFFSET} is a 14-bit address and \texttt{AD} is a 2-bit field used to select the addressing mode. The \texttt{OPCODE} bit field allows a maximum of 64 instructions. The \texttt{DEST}, \texttt{SRC1} and \texttt{SRC2} are 5 bit wide and allow a maximum of 32 registers to be addressed. \texttt{BIT1} and \texttt{BIT2} are also 5 bits wide and allow any of 32 bits of a register to be addressed.
Figure 4.2: The instruction formats.
The register type format is used to perform operations between registers. This format contains two source and destination registers, that is, allows support of triadic addressing. The immediate type format is used to perform operations involving a 14 bit immediate value. That is, one of the source operands is replaced by an immediate value. The bit operations type format is a special one since it allows operations involving bit manipulation. An opcode field is specified, two source registers and two fields for specifying specific bits of each source register. The control instruction formats are used to perform jump and branch instructions. The instruction format contains a condition register to evaluate the type of jump or branch instruction to be executed. This format allows a jump or branch to an address given by a 14 bit offset.

4.4 Proposed instruction set

The proposed instruction set is divided into six groups as shown below. A complete description is given in Appendix A.

1. Arithmetic.
2. Flow control.
3. Logical.
4. Data movement.
5. Bit manipulation.
6. Special.
4.5 Pipelining

Figure 4.3 depicts a typical 5 stage pipeline. The first stage is responsible for fetching the next instruction to be executed. The second stage is responsible for decoding the instruction. The third stage, the ALU stage is concerned with ALU operations and calculating branch address. The fourth stage is the memory stage where access to memory operands is allowed while the fifth and last stage of this pipeline is responsible for writing the results back to the register file. Pipelining is the most common kind of parallel processing of instructions and deserves special consideration. Throughput of a processor is improved by exploiting instruction level parallelism. This is possible when instructions in a sequence are independent and can thus be executed by overlapping. The main idea of an instruction pipeline is to overlap the execution of multiple instructions. Then each pipeline stage can perform an operation and forward the results to the next stage. Whereas in the microcoded non-pipeline design, one unit does all the work and thus is slower.

With the help of pipelining, a designer can ensure that at least one instruction is started each cycle. In the ideal case the pipeline can reduce the number of cycles per instruction by a factor equal to the depth of the pipeline. The pipeline needs a steady stream of instructions to achieve this ideal speedup. Therefore, events such as branches, exceptions and interrupts can disrupt the pipeline and cause loss of performance. Management of these events make pipelining complex since addi-
Table 4.4: Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add</td>
</tr>
<tr>
<td>adds</td>
<td>Add signed</td>
</tr>
<tr>
<td>div</td>
<td>Divide</td>
</tr>
<tr>
<td>divs</td>
<td>Divide signed</td>
</tr>
<tr>
<td>mul</td>
<td>Multiply</td>
</tr>
<tr>
<td>muls</td>
<td>Multiply signed</td>
</tr>
<tr>
<td>sub</td>
<td>Subtract</td>
</tr>
<tr>
<td>subs</td>
<td>Subtract signed</td>
</tr>
</tbody>
</table>

Table 4.5: Logical Instructions

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>And logical</td>
</tr>
<tr>
<td>asl</td>
<td>Arithmetic shift left</td>
</tr>
<tr>
<td>asr</td>
<td>Arithmetic shift right</td>
</tr>
<tr>
<td>lsl</td>
<td>Logical shift left</td>
</tr>
<tr>
<td>lsr</td>
<td>Logical shift right</td>
</tr>
<tr>
<td>not</td>
<td>Logical not</td>
</tr>
<tr>
<td>nand</td>
<td>Logical nand</td>
</tr>
<tr>
<td>nor</td>
<td>Logical nor</td>
</tr>
<tr>
<td>or</td>
<td>Logical or</td>
</tr>
<tr>
<td>slt</td>
<td>Set on less than</td>
</tr>
<tr>
<td>sgt</td>
<td>Set on greater than</td>
</tr>
<tr>
<td>sle</td>
<td>Set on less than or equal</td>
</tr>
<tr>
<td>sge</td>
<td>Set on greater than or equal</td>
</tr>
<tr>
<td>seq</td>
<td>Set on equal</td>
</tr>
<tr>
<td>sne</td>
<td>Set on not equal</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive or</td>
</tr>
</tbody>
</table>
### Table 4.6: Special Instructions

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>No operation</td>
</tr>
<tr>
<td>swi</td>
<td>Software interrupt</td>
</tr>
</tbody>
</table>

### Table 4.7: Bit manipulation Instructions

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>andb</td>
<td>And specified bits</td>
</tr>
<tr>
<td>clrb</td>
<td>Clear width of bits</td>
</tr>
<tr>
<td>ff0b</td>
<td>Find first bit clear</td>
</tr>
<tr>
<td>ff1b</td>
<td>Find first bit set</td>
</tr>
<tr>
<td>orb</td>
<td>Or specified bits</td>
</tr>
<tr>
<td>setb</td>
<td>Set width of bits</td>
</tr>
<tr>
<td>xorb</td>
<td>Exclusive or specified bits</td>
</tr>
</tbody>
</table>

### Table 4.8: Flow Control Instructions

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beqz</td>
<td>Branch equal zero</td>
</tr>
<tr>
<td>bnez</td>
<td>Branch not equal zero</td>
</tr>
<tr>
<td>call</td>
<td>Procedure call</td>
</tr>
<tr>
<td>j</td>
<td>Jump to address</td>
</tr>
<tr>
<td>jr</td>
<td>Jump register</td>
</tr>
<tr>
<td>ret</td>
<td>Return from call</td>
</tr>
<tr>
<td>wait</td>
<td>Wait for a period</td>
</tr>
</tbody>
</table>

### Table 4.9: Data Movement Instructions

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lod</td>
<td>Load register from memory</td>
</tr>
<tr>
<td>mov</td>
<td>Move between registers</td>
</tr>
<tr>
<td>stor</td>
<td>Store register to memory</td>
</tr>
</tbody>
</table>
tional hardware logic must be incorporated in order to resolve data hazards, resource conflicts and restarts. Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. Thus, if branches appear frequently in a program, delayed branch pipelining principle provides predictable operation but little gain in performance. In addition, pipelining introduces unpredictability in the system due to flushing and thus pipelining should not be recommended in hard real-time embedded systems.

To fulfill the requirement of executing one instruction per clock cycle the pipeline must be prevented from breaking. However, predicting this is very difficult and complex code analyzers would be necessary. These analyzers are usually based on detailed information about instruction execution which is usually proprietary. Thus, in view of the above reasons it is recommended to either prevent breaking of the pipeline or renounce it completely for real-time systems.
Figure 4.3: A typical five stage pipeline.
Chapter 5

Data path design

5.1 Data path design

In the design of a processor the first step is the definition of the instruction set. In order to design a suitable instruction set, one must keep in mind the functionality of the instructions and have an insight into the needs of a compiler writer who is going to use these instructions. Once the instruction set is acceptable the hardware design commences. The most important part of the processor, the datapath is now designed. The datapath is designed by inspection of the instruction set so that all the functions required by the instructions can be implemented by the datapath. The aim is to implement all the functions in the smallest number of cycles since the datapath determines the processor cycle time. The datapath is designed for flow of data at each cycle of the instruction to check for possible resource conflicts.
The data path is designed keeping in view the instruction set, therefore it is crucial to design the instruction set carefully. Since RISC is a load/store architecture, most of the instructions are register based. Registers are more flexible in terms of evaluating expressions than stack or accumulators. Also registers can hold variables, thus reducing CPU memory traffic and speeding up program speed (since registers are the fastest storage device available) and improving code density as registers can be addressed with fewer bits than a memory location.

Other advantages of a register-register architecture are a simple fixed length instruction, and instructions taking same number of clock cycles to execute. However, registers have disadvantages too. A higher instruction count is expected, instruction coding may be wasteful and the optimizing compiler must be very good to optimize register allocation. A RISC designer must define an instruction set in which most of the instructions execute in the same number of clock cycles. However, instructions like LOAD and STORE that access main memory must allow a latency of one cycle due to slower main memory. A LOAD/STORE architecture provides a number of benefits.

- Help in reducing the memory traffic between CPU and memory.
- The instruction set is simplified since operations are between registers.
- Compilers can optimize register allocation since memory operations are eliminated.
Another important design consideration is the time per cycle which is usually determined by the slowest pipeline stage which in turn is determined by the datapath. The fetch instruction usually has the longest delay since it involves memory access. The time per cycle depends on a combination of the above factors, i.e. the instruction decode time, the instruction operation time, instruction access time and architectural simplicity. The time required to decode an instruction is related to the number of instructions in the instruction set and the number of instruction formats supported.

If the instruction formats are uniform, the processor can decode multiple bit fields simultaneously. The registers can be decoded at the same time owing to the orthogonality of the instruction set. The source and destination fields are always at the same position in each instruction format. The instruction operation time is much shorter in RISC since the time required for register-register operations is small as compared to memory operations. Achieving a one instruction per cycle rate is impossible since the memory can not deliver instructions at this rate. A Load/Store architecture helps reduce the memory bandwidth but increases the number of instructions per cycle since more simple instructions must be executed in order to equal one complex instruction of a comparable CISC machine. To handle an increased instruction rate, two techniques are employed.

- Hierarchical memory systems using cache memory.
• Supporting a Harvard architecture i.e. separate instruction and data caches to effectively double the cache memory bandwidth.

Another method to minimize the time required to fetch an instruction is to have fixed length instructions. Complying with the RISC philosophy, we have decided to implement a fixed one word instruction length of 32 bits. A shortcoming of the reduced instruction set is the increase in number of instructions per task as discussed above. The result is that the code written for a RISC machine is usually 20% to 30% larger than that written for a CISC machine [5].

5.2 Datapath cycle analysis

Figures 5.1 to 5.4 show the datapath resulting from the description of the instruction set.

5.2.1 Arithmetic and logical operations

Figure 5.1 shows the cycle by cycle description of the arithmetic and logical operations, i.e., add, sub, .and, or, xor..., SLT, SGT etc. During the first cycle one operand is loaded from the register file while the other can be loaded from either main memory, an extended register file or the main register file. The ALU performs the required operation on the data depending on the opcode and outputs the result to the register file or the extended variable file.
5.2.2 Branch operations

Figure 5.2 shows how conditional branches and jumps are handled. The condition field is evaluated to determine the type of branch. The address of the branch is sent to the MAR and loaded into PC. For jumps the PC unconditionally points to the location given by the offset field.

5.2.3 Bit manipulation operations

Figure 5.3 details the cycle by cycle execution of bit manipulation instructions like ff0b, ff1b, setb, clr b etc. Once the operands to be operated upon are loaded it is possible to individually manipulate bits of the operands. The result can be written into the register file or the on-chip variable file.

5.2.4 LOAD/STORE operations

Figure 5.4 shows the cycle by cycle execution of the LOAD and STORE operations. In case of LOAD operation a specified register can be loaded with data from memory and in case of STORE operations data from a specified register can be stored in memory.
Logical Operations

and, or, xor ..., SLT, SGT, ...

Figure 5.1: Arithmetic and logical operations.
Branch Instructions

The condition field is evaluated to determine the type of branch. The address of the branch is then sent to MAR. The address is loaded into PC.

For jumps the PC unconditionally points to the memory location pointed to by the offset field.

Figure 5.2: Branch instructions.
Bit Manipulation

andb, orb, xor (write to PSW)
ff0b, ff1b -
setb, clr -

Figure 5.3: Bit manipulation operations.
Load/Store

Load: Load a register with data from memory
   i) Send address of memory location
   ii) Load value in register

Store: Store a register value to memory

Figure 5.4: LOAD/STORE operations.
Chapter 6

Modeling decisions

6.1 Modeling

VHDL was chosen for modeling the proposed processor since it allows the designer to model complex digital systems at the behavioral level. This level of description is quite abstract since no actual components are described and thus it is not possible to calculate timing delays. Firstly, a behavioral model of the system is designed and then simulated for proper operation. Once this stage is over, and the designer is satisfied with the simulation results, he can decompose the system into its various components and create a structural model which closely resembles the actual hardware being designed.

For example, an 8-bit full adder may be decomposed into eight 1-bit full adders. The designer will specify the details of a 1-bit adder and interconnect eight to form a
complete 8-bit full adder. The advantages of this technique are obvious. Firstly, the
designer can spend more time specifying the details of the architecture of the 1-bit
adder. Secondly, he can verify this simple unit and thus guarantee that a complex
interconnection of these simple units will also function properly. In this thesis work
an instruction set optimized for real-time applications is proposed and modeled in
VHDL. In order to verify the operation of the proposed instructions, a simple model
of processor and memory is also written in VHDL.

6.2 Instruction set model

The basic instruction set guidelines are predictability of instruction execution times
and support for high level programming. Floating point arithmetic should be
avoided in the implementation. Besides being the most complex part of the hard-
ware, it is difficult to verify for correctness. The compiler can substitute the unimple-
mented instructions by invoking subroutines performing the same functions. Also,
dynamic features including recursion, dynamic data structures, references, dynamic
procedure calls, virtual addressing etc should not be present in order to remove
problems concerning time and memory space requirements.

Once the instruction set is proposed it is necessary to verify it by simulation.
The proposed instructions are modeled at the behavioral level in the form of sub-
programs and functions. Each instruction is tested and verified by applying data
and confirming that the instruction executes correctly. Once all the instructions are tested they are stored in a package. A four type logic is used in modeling the instruction set and the processor-memory model. In addition to the normal '0' and '1' levels 'Z'( high impedance) and 'X'( unknown) are defined. These additional logic levels allow easier handling of buses and invalid data.

In this thesis an instruction set suitable for real-time applications is proposed. The literature concerned with real-time systems deals with various issues of maintaining and designing fast, predictable and fault tolerant real-time computing systems. The literature reports a number of architectural and design features that have been implemented in an effort to provide architectural support for real-time systems and include high level language features that are missing from real-time languages. This thesis attempts to target this problem by carefully analyzing popular RISC processors and determining the extent of real-time features support.

6.3 RISC model

A RISC CPU core targeted at real-time applications is proposed. The objective of this thesis was to study and evaluate the important characteristics of real-time systems. During the course of this study, certain desirable features were identified. An effort is made to propose a core of a RISC CPU that supports these features. These desirable characteristics can be supported at the architectural as well as instruction
set level. The proposed model therefore has the following features.

6.3.1 A reduced instruction set

In order to satisfy predictability requirements of real-time systems, implementation of pipelining and caching is not recommended. A reduced instruction set helps reduce instruction decode time and thus programs are executed faster.

6.3.2 Fixed format instruction size

For our model we have chosen a fixed size of 32 bits as the instruction size. A fixed instruction size reduces decode time since the instruction formats are orthogonal and designed so that the source, destination and immediate fields are found in the same bit positions.

6.3.3 A three port $32 \times 32$ register file

A three port register file allows two simultaneous reads and one write. The register file has 32 registers of 32 bits each. In order to address two source and one destination register at the same time takes 15 bits of the instruction register. A register file size of 32 registers has been found to be adequate for RISC machines.
6.3.4 An uncached system

As far as real-time embedded systems are concerned the decision to implement an on-chip cache memory or not is debatable. We have opted to implement an on-chip variable storage rather than a cache subsystem. As discussed earlier, cache brings unpredictability into a real-time system. In case of a cache miss the cache has to be reloaded during the operation causing essentially different operand access times. In addition to unpredictability, an on-chip cache would certainly increase the cost of the processor which would add up to the overall system cost. For embedded systems cost is considered a major parameter which has to be minimised.

Some examples from the industry supporting cache-less systems for embedded systems are given below. Designers of the ACORN RISC machine [38] systematically stripped all non-essential hardware out of the architecture in order for the CPU to meet its personal computer cost goals. The result is an anomaly— a RISC CPU designed to run without caches or memory management hardware. The present ARM CPU core design goes back to the original decision not to use cache, but to exploit simple instructions and high memory bandwidth to minimize hardware requirements.

Integrated Device Technology have recently introduced an embedded system version of the R3000, the R3001 [40]. The primary feature in the new part was a redesign of the local bus coming out of the chip, permitting the system designer to
use less expensive off-chip caches or to eliminate caches altogether. The 29000 from AMD [18] rejects the notion of large general purpose off-chip caches in favour of a much more selective on-chip hardware for managing bus bandwidth.

Sometimes even locking a service routine into the cache won’t help. Albeit the speed of the routine will improve and result in predictable execution time but locking has its own disadvantages. Many of the caches are direct mapped and when a subroutine is locked into the cache it’s range of addresses is lost forever. This can be troublesome for embedded system developers. According to a Motorola publication, in MIPS R3000 and SPARC processors, simple cache implementation does not perform well in real-time and cache is often disabled to increase performance.

Where real-time systems have included caches, the resulting performance improvement often leads to an under utilization of system resources. This is exactly the case in the US Navy’s AEGIS Combat System [20]. The AN/UYK-43 computer which provides the central computing power for the system has a 32k word cache partitioned for instructions and data. However, due to unpredictable cache performance, all module(task) utilizations are calculated as if the cache were turned off (cache bypass option). As a result, the theoretically overutilized CPU is often underutilized at run-time when cache is enabled. According to the authors [20] present cache architectures do not provide predictable performance in an environment which allows interrupts and task preemptions like a real-time embedded system.

Usually, real-time applications do not allow the use of caches [14], because of
cache flush problems during critical execution phases i.e during interrupts, context switch etc. [14]. Implementation of a variable file represents a better solution since it provides support for real-time tasks, storage of variables for high-level languages and by providing deterministic execution times. Therefore, we chose to implement a variable register file rather than cache memory.

6.3.5 A LOAD/STORE architecture

In accordance with RISC philosophy, only LOAD and STORE instructions are allowed access to main memory. The remaining instructions are register based and the optimizing compiler has to do good register allocation. Adhering to this philosophy decreases program execution time since most operations are register based and registers are the fastest storage device available.

6.3.6 Local variable storage

The concept of an on-chip register file is very important for real-time systems. This extra file can be used to store variables of a high level language, help in reducing context switch time since procedures can exchange parameters on-chip. In addition, fast data transfers can take place between the main register file and this extended on-chip variable file. The concept of having an on-chip register file also helps mitigate the affects of loss of performance due to renouncing cache memory since registers are faster than cache and predictable. An added advantage is that High Level Languages
are supported by this feature. At the moment the size is not fixed, however, a size of 256 to 512 words of 32 bits seems reasonable.

Once the core of the processor is designed, a number of other features can be added as desired. These features enhance the processor capabilities at the cost of larger size and power consumption. RISC philosophy was chosen in order to provide better performance and overall smaller size. The functions that can be added are the following.

- Timers.
- Floating point unit.
- Dedicated interrupt controller.
- Cache memory (full Harvard architecture).
- Dedicated I/O controller.

### 6.3.7 Support for predictability

A reduced instruction set in which every instruction performs a single operation makes it possible to calculate the instruction execution time. The decode logic resulting from this design will be simple. A reduced instruction set thus introduces predictability, reliability and it becomes easy to verify the system. The inclusion of an on-chip variable storage increases high level language support and predictability.
of the system.

6.3.8 Support for high MIPS

High speed computation is possible due to a small instruction set, a LOAD/STORE architecture, single cycle execution of instructions and a shorter clock cycle. In addition, with the help of the on-chip variable file a high level language's data could be exchanged quickly with the main register file.

6.3.9 Support for fault tolerance

A processor can recover from run-time errors like divide by zero, or overflow/underflow by employing features like exception handling and traps. In addition, there can be built-in functions like POST(power on self test) and reset on an error etc. Emphasising simplicity in the system will increase reliability and will make it possible to easily verify the system.

6.3.10 Support for data I/O

In order to handle the large amounts of data input and output of a real-time system, it is necessary to relegate this function to a dedicated processor.
6.3.11 Support for interrupt handling

A real-time system usually resides in an environment in which a high data throughput and fast response to interrupts is desired. In order to achieve fast interrupt recognition and handling, a dedicated processor is recommended.

6.4 VHDL

A behavioral description of the instructions is written in VHDL, the IEEE standard hardware description language for describing complex digital systems. The instructions are verified for correct operation by simulation using data as input. Once the instructions are verified they are included in the instruction set. In this manner all the instructions are tested individually for correct operation. In order to validate the instructions for coding efficiency, some sample programs are written using the proposed instructions. In order to simulate these programs we have to handle placement of instruction addresses of loops and branches manually. The instructions are executed on a behavioral model of a processor and memory which have also been modeled in VHDL.

A LOAD/STORE architecture ensures that 80 percent of the instructions take a single cycle to execute. The memory is modeled as a linearly addressed space and functions like floating point unit (FPU) and memory management unit (MMU) are excluded from the design since they occupy a large space and consume a lot of
power. To ensure predictable behavior, cache memory system is also removed at the
cost of performance. The design is flexible since it allows a provision for another
addressing mode and an on-chip extended register file. In addition, the processor
model allows for 32 types of branch conditions and same number of jump types.
At the moment, however, only `beqz(branch equal zero), `bnez (branch not equal zero
and two jumps are implemented. In the future a number of modifications could be
designed into the architecture like data bypassing, powerful interrupt handling, fault
tolerance, Harvard architecture etc.

On the basis of instruction set studies a small instruction set of approximately 40
instructions is proposed. The important characteristic of these instructions are that
they comply with RISC philosophy and perform only one operation per instruction
in order to simplify the decode logic and to increase the speed of execution. All the
instructions have a fixed length of 32 bits and the instruction formats are designed in
such a way that the source and destination fields occupy the same bit positions thus
aiding the one instruction per cycle design. The instructions include powerful bit
manipulation support which is an essential feature of real-time systems as most of
them are designed for process control applications which usually perform monitoring
and polling functions.
Chapter 7

Simulation results and discussion

7.1 Simulation results

The proposed instructions are simulated at the behavioral level in VHDL using a compiler and simulator from Model Technology. The correct operation of these instructions is checked by giving as input random data and verifying the output. The simulation results of some individual instructions showing correct operation are shown on the following pages. Simulation results of instructions ASL, MUL, FF0b and SETb are attached as examples. In addition, simulation result of a sample real-time program is attached which does data input and monitoring, data conditioning and finally data output as in an actual real-time application. The program is coded using the proposed instructions and simulation results are attached.
7.2 Discussion of simulation results

Figure 7.1 shows simulation result of *setb* (set bits to 1) instruction. This instruction is part of the bit manipulation group and allows the programmer to fill any number of bits starting from any arbitrary bit with 1's. This instruction executes in a single cycle since the data to be manipulated and number of bits and position is available within the instruction.

Figure 7.2 shows simulation result of another powerful instruction belonging to the bit manipulation group of instructions. The *find first bit set to 1* or *ff1b* scans the input data from the most significant bit down to the least significant one and returns the bit position of the first bit it finds set to 1.

Figure 7.3 shows simulation result of a similar instruction belonging to the bit manipulation group of instructions. The *find first bit set to 0* or *ff0b* scans the input data from the most significant bit down to the least significant one and returns the bit position of the first bit it finds set to 0. Both *ff0b* and *ff1b* are powerful instructions that are useful in real-time systems which perform monitoring and data manipulation functions extensively.

Figure 7.4 shows simulation result of variable arithmetic shift left or *asl* instruction. This instruction allows the programmer to shift left the input data by a variable number of bits. Since the data to be manipulated and number of shifts are specified in the instruction itself this instruction executes in a single cycle.
Figure 7.5 shows simulation result of multiply instruction. This instruction allows the programmer to multiply two 32 bit numbers and return the result in a 32 bit destination register with the higher bits truncated.

Figures 7.6 and 7.7 show simulation results of a sample program written using the proposed instruction set in order to demonstrate the coding capability of the instructions. The program does a simple multiply by successive addition. As the simulation results show, the program loops and constantly checks the value held in register R1 which is the counter value in this case equal to 4. In figure 7.7 one can see the result that in register R3, in this case 8, being written to the memory using the store instruction.
Figure 7.1: Simulation result of setb instruction.
Figure 7.2: Simulation result of \textit{ff1b} instruction.
Figure 7.3: Simulation result of ff0b instruction.
| /clk | 0000 |
| /x   | 0080 |
| /sel | 0    |
| /res | 128  |
| /opr1| 08   |
| /opr2| 04   |
| /ir  | 00110110 |
| /pc  | 01   |
| /src1| 01   |
| /src2| 01   |
| /cin |      |
| /c   | 01   |
| /mem | 36, 36, 89, F9 |
| /reg | 0F, 08, 04, 03 |

Figure 7.4: Simulation result of asl instruction.
| /clk | /x.00 | /sel.0 | /res.32895 | /rest.0000 | /oprd1.0F | /oprd2.081 | /oprd3.03 | /bit4.01 | /ir.07 | /pc.01 | /src1.1 | /src2.3 | /dest.0 | /y.0 | /int.1 | /p | /result | /a.1 | /b.3 | /cm.0 | /mem.[11, 07, B9, F9] | /reg.[03, FF, 01, 81] |

Figure 7.5: Simulation result of *mul* instruction.
Example program

mnemonics

mov R0,0; move 0 into register R0
mov R1,#4; move 4 into register R1
mov R2,#2; move 2 into register R2

loop: add R3,R2,R3; add contents of R2, R3 and store in R3
sub R1,R1,1; decrement contents of R1 by 1
seq R1,R0; set PSW if R1 and R0 equal
bnez loop; branch till R1 becomes 0
store R3,#10; store the result in mem. loc 10
Figure 7.6: Simulation result of a program showing branch and loop part 1.
Figure 7.7: Simulation result showing write to memory part 2.
Example of a real-time program

mnemonics

**Data input and monitoring** mov R1, #19: move incoming data into register R1
mov R2, #4: initialise registers R2, R3 and R4 for
mov R3, #5: use in computation later
mov R4, #192
ff1b R1: scan the input data for a specified pattern
seq R5, R2: set the processor status word if pattern found
beq new.data: get new data if pattern not found
**Data conditioning** lsl R1, R1, R2: shift left data in R1 four times
setb R1, #2, #2: set two bits of R1 from offset bit 2
div R1, R1, R3: divide the contents of R1 by R3 store in R1
add R1, R1, R4: add contents of R1 and R4 store in R1
clr b R1, 4,3: clear four bits of R1 starting from bit no. 3
**Data output** stor R1, #14; write the result in R1 to memory location 14

A typical real-time system essentially consists of three stages. In the first stage new data is continuously sampled and filtered. In the above code, this is shown by
lines 0 to 6. In the above program, the data is repeatedly searched for a specific pattern and as soon as this pattern is detected the program exits the loop and starts conditioning the data. This stage is represented by lines 7 to 11 where the data is manipulated in a number of ways using some arithmetic operations and powerful bit manipulation instructions of the processor. In the third and final stage the data is written to a specific memory location depicted by line 12. In an actual program, the data is written to the output port of the processor in order to communicate with the outside world.
Figure 7.8: Simulation result showing output of a real-time example.
7.3 Conclusions

An in-depth study of real-time systems is carried out in order to identify their desirable features. An analysis of some existing RISC and CISC processors is done to show how these desirable features are supported at the instruction set and architectural level. Once these desirable characteristics are identified, an effort is made to propose an instruction set and an architecture aimed at overcoming shortcomings of existing areas and identifying new ones.

In this thesis an instruction set is proposed which boasts good bit manipulation support, a large repertoire of compare and set instructions, move instructions and branches. These choices are made by analyzing instruction set studies of general programs as well as real-time systems. The proposed RISC core has predictable operation, pipelining and cache memory are not recommended to be implemented and the instruction set contains only 40 instructions. The instruction formats are designed to be highly orthogonal in order to simplify the decode logic.

In order to facilitate HLL and support for real-time systems, an on-chip variable register file is proposed. The size of this register file is undetermined at this time but a size of between 256 to 512 words of 32 bits is feasible.

The proposed instruction set and architecture are modeled at the behavioral level in VHDL.
A REAL-TIME RISC PROCESSOR

Arithmetic Instructions

add
Operation: Destination \(\Leftarrow\) Source1 + Source2

Assembler Syntax:
- add rD,rS1,rS2    add unsigned
- add rD,rS1.IMM14  add immediate unsigned
- add rD,rS1,\#rS2  add indexed unsigned

The add instruction performs an add operation on the contents of two source registers. An immediate and an indexed value can also be added. This instruction is supported by all addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMME</td>
<td>D</td>
</tr>
</tbody>
</table>

adds
Operation: Destination \(\Leftarrow\) Source1 + Source2

Assembler Syntax:
- adds rD,rS1,rS2 signed add
- adds rD,rS1.IMM14 signed add immediate
- adds rD,rS1,\#rS2 signed indexed add

The adds instruction performs a signed add operation on the contents of source registers rS1 and rS2 and stores the result in destination register rD. Similarly, an immediate and an indexed value can also be added. This instruction is supported by all addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMME</td>
<td>D</td>
</tr>
</tbody>
</table>
A REAL-TIME RISC PROCESSOR

div

Operation: Destination ← Source1 ÷ Source2

Assembler Syntax:

• div rD.rs1.rs2               unsigned divide
• div rD.rs1.IMM14            unsigned divide immediate
• div rD.rs1.srs2             unsigned indexed divide

The divide instruction stores the result of the division of data in rs1 register by the data in the rs2 register. The rs2 register can contain an immediate value also. The divide instruction is supported by two instruction formats namely, the R-type and I-type. This instruction is supported by all addressing modes.

R-TYPE  
I-TYPE    


divs

Operation: Destination ← Source1 ÷ Source2

Assembler Syntax:

• divs rD.rs1.rs2            signed divide
• divs rD.rs1.IMM14          signed divide immediate
• divs rD.rs1.srs2           signed indexed divide

The divide instruction stores the result of the division of data in rs1 register by the data in the rs2 register. The rs2 register can contain an immediate value also. The divide instruction is supported by two instruction formats namely, the R-type and I-type. This instruction is supported by all addressing modes.

R-TYPE  
I-TYPE    

A REAL-TIME RISC PROCESSOR

mul

Operation: Destination ← Source1 × Source2

Assembler Syntax:

- mul rD, rS1, rS2  multiply
- mul rD, rS1, IMM14  multiply immediate
- mul rD, rS1, &rS2  multiply indexed

The multiply operation multiplies the data in rS1 and rS2 and stores the least 32 bits of the 64 bits result in rD. Alternately, an immediate value may be multiplied with rS1 and the result stored in rD. Since this instruction supports all three addressing modes it is possible to multiply rS1 with an indexed value in rS2. Multiply operation is supported by the R-type and I-type instruction formats.

R-TYPE [OPCODE | DEST | SRC1 | ADDR | SRC2 | X]
I-TYPE [OPCODE | DEST | SRC1 | ADDR | IMMED]

muls

Operation: Destination ← Source1 × Source2

Assembler Syntax:

- muls rD, rS1, rS2  multiply signed
- muls rD, rS1, IMM14  multiply signed immediate
- muls rD, rS1, &rS2  multiply indexed signed

The multiply operation multiplies the data in rS1 and rS2 and stores the least 32 bits of the 64 bits result in rD. Alternately, an immediate value may be multiplied with rS1 and the result stored in rD. Multiply operation is supported by the R-type and I-type instruction formats.

R-TYPE [OPCODE | DEST | SRC1 | ADDR | SRC2 | X]
I-TYPE [OPCODE | DEST | SRC1 | ADDR | IMMED]
A REAL-TIME RISC PROCESSOR

sub

Operation: Destination ← Source1 - Source2

Assembler Syntax:

- sub rD,rS1,rS2    subtract
- sub rD,rS1,IMM14  subtract immediate
- sub rD,rS1,śrS2   subtract indexed

The subtract operation subtracts the data in rS2 from rS1 or an immediate value from rS1 and stores the result in rD. It is supported by both the R-type and I-type instruction formats and all addressing modes.

R-TYPE | OPCODE | DEST | SRC1 | ADDR | SRC2 | X
---|---|---|---|---|---|---
I-TYPE | OPCODE | DEST | SRC1 | ADDR | IMMED | X

sub

Operation: Destination ← Source1 - Source2

Assembler Syntax:

- subs rD,rS1,rS2    subtract signed
- subs rD,rS1,IMM14  subtract signed immediate
- subs rD,rS1,śrS2   subtract indexed signed

The subtract operation subtracts the signed data in rS2 from rS1 or an immediate value from rS1 and stores the result in rD. It is supported by both the R-type and I-type instruction formats and all addressing modes.

R-TYPE | OPCODE | DEST | SRC1 | ADDR | SRC2 | X
---|---|---|---|---|---|---
I-TYPE | OPCODE | DEST | SRC1 | ADDR | IMMED | X
A REAL-TIME RISC PROCESSOR

Logical Instructions

and

Operation: Destination ← Source1 ∧ Source2

Assembler Syntax:

- and rD,rS1,rS2     logical and
- and rD,rS1,IM14    logical and immediate
- and rD,rS1,0rS2    logical and indexed

The and operation ANDs the data in rS2 with rS1 or an immediate value in rS2 with rS1 and stores the result in rD. This instruction is supported by both the R-type and I-type instruction formats and all the three addressing modes.

R-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
</table>

I-TYPE

| OPCODE | DEST | SRC1 | ADDR | IMMED |

asl

Operation: Destination ← Source1 << Source2

Assembler Syntax:

- asl rD,rS1,rS2     arithmetic shift left
- asl rD,rS1,IM14    arithmetic shift left immediate

The asl instruction shifts left the contents of source register rS1 by the number of bits specified in rS2 and stores the result in destination register. This instruction is supported by the R and I-type instruction formats and two addressing modes.

R-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
</table>

I-TYPE

| OPCODE | DEST | SRC1 | ADDR | IMMED |
A REAL-TIME RISC PROCESSOR

asr

Operation: Destination ← Source1 ⊙ Source2

Assembler Syntax:

- asr rD,rS1,rS2 arithmetic shift right
- asr rD,rS1,IMM14 arithmetic shift right immediate

The asr instruction shifts right the contents of source register rS1 by the number of bits specified in rS2 and stores the result in destination register. This instruction is supported by the R and I-type instruction formats and two addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMMED</td>
<td></td>
</tr>
</tbody>
</table>

lsl

Operation: Destination ← Source1 ≪ Source2

Assembler Syntax:

- lsl rD,rS1,rS2 logical shift left
- lsl rD,rS1,IMM14 logical shift left immediate

The lsl instruction shifts left the contents of source register rS1 by the number of bits specified in rS2 and stores the result in destination register. This instruction is supported by the R and I-type instruction formats and two addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMMED</td>
<td></td>
</tr>
</tbody>
</table>
A REAL-TIME RISC PROCESSOR

\textbf{lsr}

\textit{Operation:} \textit{Destination} \leftarrow \textit{Source1} \gg \textit{Source2}

\textbf{Assembler Syntax:}

- \texttt{lsr rD,rS1,rS2} \hspace{1cm} \text{logical shift right}
- \texttt{lsr rD,rS1,IMM14} \hspace{1cm} \text{logical shift right immediate}

The \texttt{lsr} instruction shifts right the contents of source register \texttt{rS1} by the number of bits specified in \texttt{rS2} and stores the result in destination register. This instruction is supported by the R and I-type instruction formats and two addressing modes.

\begin{center}
\begin{tabular}{l}
\textbf{R-TYPE} \hspace{1cm} | \hspace{1cm} \textbf{OPCODE} \hspace{0.5cm} | \hspace{0.5cm} \textbf{DEST} \hspace{0.5cm} | \hspace{0.5cm} \textbf{SRC1} \hspace{0.5cm} | \hspace{0.5cm} \textbf{ADDR} \hspace{0.5cm} | \hspace{0.5cm} \textbf{SRC2} \hspace{0.5cm} | \hspace{0.5cm} \textbf{X} \\
\textbf{I-TYPE} \hspace{1cm} | \hspace{1cm} \textbf{OPCODE} \hspace{0.5cm} | \hspace{0.5cm} \textbf{DEST} \hspace{0.5cm} | \hspace{0.5cm} \textbf{SRC1} \hspace{0.5cm} | \hspace{0.5cm} \textbf{ADDR} \hspace{0.5cm} | \hspace{0.5cm} \textbf{IMMED} \\
\end{tabular}
\end{center}

\textbf{not}

\textit{Operation:} \textit{Destination} \leftarrow \overline{\textit{Source1}} \hspace{1cm}

\textbf{Assembler Syntax:}

- \texttt{not rD,rS1,rS2} \hspace{1cm} \text{1's complement}
- \texttt{not rD,rS1,IMM14} \hspace{1cm} \text{complement immediate}
- \texttt{not rD,rS1,\&rS2} \hspace{1cm} \text{complement indexed}

The 1's complement operation inverts the contents of \texttt{rS1} register and stores the result in \texttt{rD}. This instruction is supported by the R and I-type instruction formats and all the three addressing modes.

\begin{center}
\begin{tabular}{l}
\textbf{R-TYPE} \hspace{1cm} | \hspace{1cm} \textbf{OPCODE} \hspace{0.5cm} | \hspace{0.5cm} \textbf{DEST} \hspace{0.5cm} | \hspace{0.5cm} \textbf{SRC1} \hspace{0.5cm} | \hspace{0.5cm} \textbf{ADDR} \hspace{0.5cm} | \hspace{0.5cm} \textbf{SRC2} \hspace{0.5cm} | \hspace{0.5cm} \textbf{X} \\
\textbf{I-TYPE} \hspace{1cm} | \hspace{1cm} \textbf{OPCODE} \hspace{0.5cm} | \hspace{0.5cm} \textbf{DEST} \hspace{0.5cm} | \hspace{0.5cm} \textbf{SRC1} \hspace{0.5cm} | \hspace{0.5cm} \textbf{ADDR} \hspace{0.5cm} | \hspace{0.5cm} \textbf{IMMED} \\
\end{tabular}
\end{center}
A REAL-TIME RISC PROCESSOR

nand

Operation: Destination ← Source1 ∧ Source2

Assembler Syntax:

- nand rD,rS1,rS2           logical nand
- nand rD,rS1,IMM14        logical nand immediate
- nand rD,rS1,addrS2       logical nand indexed

The nand operation nands the data in rS2 with rS1 or an immediate value in rS2 with rS1 and stores the result in rD. This instruction is supported by both the R-type and I-type instruction formats and all the three addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMMED</td>
<td></td>
</tr>
</tbody>
</table>

nOR

Operation: Destination ← Source1 ∨ Source2

Assembler Syntax:

- nor rD,rS1,rS2            logical nor
- nor rD,rS1,IMM14         logical nor immediate
- nor rD,rS1,addrS2        logical nor indexed

The nor instruction performs an nor operation on the contents of rS1 and rS2 registers and stores the result in rD. In addition it is possible to nor the contents of rS1 and an immediate value. It is also possible to nor contents of rS1 and an indexed value pointed to by rS2. This instruction is supported by both the R-type and I-type instruction formats and by all three addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMMED</td>
<td></td>
</tr>
</tbody>
</table>
A REAL-TIME RISC PROCESSOR

OR

Operation: Destination ← Source1 ∨ Source2

Assembler Syntax:

- or rD,rS1,rS2 logical or
- or rD,rS1,IMM14 logical or immediate
- or rD,rS1,@rS2 logical or indexed

The or instruction performs an or operation on the contents of rS1 and rS2 registers and stores the result in rD. In addition it is possible to or the contents of rS1 and an immediate value. It is also possible to or contents of rS1 and an indexed value pointed to by rS2. This instruction is supported by both the R-type and I-type instruction formats and by all three addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMMED</td>
<td></td>
</tr>
</tbody>
</table>

slt

Operation: Destination ← Source1 < Source2

Assembler Syntax:

- slt rD,rS1,rS2 set on less than
- slt rD,rS1,IMM14 set on less than immediate
- slt rD,rS1,@rS2 set on less than indexed

The slt instruction compares the contents of two registers and sets a bit in the processor status word if the result of the comparison was true. This instruction is supported by the R and I-type instruction formats and all the three addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMMED</td>
<td></td>
</tr>
</tbody>
</table>
A REAL-TIME RISC PROCESSOR

sgt

Operation: Destination ← Source1 < Source2

Assembler Syntax:

- sgt rD,rS1,rS2    set on greater than
- sgt rD,rS1,IMM14  set on greater than immediate
- sgt rD,rS1,@rS2   set on greater than indexed

The sgt instruction compares the contents of two registers and sets a bit in the processor status word if the result of the comparison was true. This instruction is supported by the R and I-type instruction formats and all the three addressing modes.

R-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
</table>

I-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>IMMED</th>
</tr>
</thead>
</table>

sle

Operation: Destination ← Source1 ≤ Source2

Assembler Syntax:

- sle rD,rS1,rS2    set on less than or equal
- sle rD,rS1,IMM14  set on less than or equal immediate
- sle rD,rS1,@rS2   set on less than or equal indexed

The sle instruction compares the contents of two registers and sets a bit in the processor status word if the result of the comparison was true. This instruction is supported by the R and I-type instruction formats and all the three addressing modes.

R-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
</table>

I-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>IMMED</th>
</tr>
</thead>
</table>
A REAL-TIME RISC PROCESSOR

sge
Operation: \(\text{Destination} \Leftarrow \text{Source1} \triangleleft \text{Source2}\)

Assembler Syntax:
- \text{sge rD, rS1, rS2} set on greater than or equal
- \text{sge rD, rS1, IMM14} set on less than immediate
- \text{sge rD, rS1, rS2} set on less than indexed

The sge instruction compares the contents of two registers and sets a bit in the processor status word if the result of the comparison is true. This instruction is supported by the R and I-type instruction formats and all the three addressing modes.

\[
\begin{array}{lllllll}
\text{R-TYPE} & \text{OPCODE} & \text{DEST} & \text{SRC1} & \text{ADDR} & \text{SRC2} & \text{X} \\
\text{I-TYPE} & \text{OPCODE} & \text{DEST} & \text{SRC1} & \text{ADDR} & \text{IMMED} &
\end{array}
\]

seq
Operation: \(\text{Destination} \Leftarrow \text{Source1} \triangleleft \text{Source2}\)

Assembler Syntax:
- \text{seq rD, rS1, rS2} set on equal
- \text{seq rD, rS1, IMM14} set on equal immediate
- \text{seq rD, rS1, rS2} set on equal indexed

The seq instruction compares the contents of two registers and sets a bit in the processor status word if result of the comparison of is true. This instruction is supported by the R and I-type instruction formats and all the three addressing modes.

\[
\begin{array}{lllllll}
\text{R-TYPE} & \text{OPCODE} & \text{DEST} & \text{SRC1} & \text{ADDR} & \text{SRC2} & \text{X} \\
\text{I-TYPE} & \text{OPCODE} & \text{DEST} & \text{SRC1} & \text{ADDR} & \text{IMMED} &
\end{array}
\]
A REAL-TIME RISC PROCESSOR

sne
Operation: Destination ← Source1 △ Source2

Assembler Syntax:
- sne rD,rS1,rS2 set on not equal
- sne rD,rS1,IMM14 set on not equal immediate
- sne rD,rS1,&rS2 set on not equal indexed

The sne instruction compares the contents of two registers and sets a bit in the processor status word if result of the comparison is true. This instruction is supported by the R and I-type instruction formats and all the three addressing modes.

R-TYPE | OPCODE | DEST | SRC1 | ADDR | SRC2 | X
I-TYPE | OPCODE | DEST | SRC1 | ADDR | IMMED

xor
Operation: Destination ← Source1 ⊕ Source2

Assembler Syntax:
- xor rD,rS1,rS2 xor
- xor rD,rS1,IMM14 xor immediate
- xor rD,rS1,&rS2 xor indexed

The xor instruction performs an exclusive or operation on the contents of rS1 and rS2 registers and stores the result in rD. This instruction is supported by both the R-type and I-type instruction formats and all three addressing modes.

R-TYPE | OPCODE | DEST | SRC1 | ADDR | SRC2 | X
I-TYPE | OPCODE | DEST | SRC1 | ADDR | IMMED
A REAL-TIME RISC PROCESSOR

Data Movement Instructions

lod

Operation: Register ← Memory

Assembler Syntax:

- lod rD,offset load from memory
- lod rD,IMM14 load immediate from memory
- lod rD,@rS2 load indexed

The lod command is used to transfer a word from a location in memory to a specified register. The memory location address can be calculated from an offset value or an immediate one or the register can be loaded from an indexed location.

R-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
</table>

I-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>IMMED</th>
</tr>
</thead>
</table>

MOV

Operation: Destination ← Source

Assembler Syntax:

- mov rD,rS1,rS2 move between registers
- mov rD,rS1,IMM14 move immediate into register
- mov rD,rS1,@rS2 move using indexed address

The move instruction transfers data between two registers or memory or moves a constant value into a register. Move can also be used to move a value from an indexed register to a destination register. This instruction is supported by the I-type and R-type instruction formats. All the addressing modes apply here.

R-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
</table>

I-TYPE

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>IMMED</th>
</tr>
</thead>
</table>
A REAL-TIME RISC PROCESSOR

**stor**

*Operation: Register ⇒ Memory*

Assembler Syntax:

- `stor rD,rS1,rS2` store
- `stor rD,rS1,IMM14` store immediate
- `stor rD,rS1,ADDR` store indexed

The store instruction is used to transfer a byte, a halfword or a word from a register to a specified location in memory using scaled mode. The address is calculated by using contents of source register1 while the value in the other register specifies whether storing byte, halfword or word. It is supported by the R and I-type instruction formats and all three addressing modes.

<table>
<thead>
<tr>
<th>R-TYPE</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SRC1</th>
<th>ADDR</th>
<th>SRC2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-TYPE</td>
<td>OPCODE</td>
<td>DEST</td>
<td>SRC1</td>
<td>ADDR</td>
<td>IMMED</td>
<td></td>
</tr>
</tbody>
</table>

**Special Instructions**

**nop**

*Operation: None ⇛ Aluoutput*

Assembler Syntax:

- `nop` no operation

The `nop` instruction is used to insert a no operation that is the ALU is kept artificially busy.

**swi**

*Operation: Cpu ⇛ Interrupt*

Assembler Syntax:

- `swi` software interrupt

The `swi` instruction is used to interrupt the cpu through software.
A REAL-TIME RISC PROCESSOR

Bit Manipulation Instructions

Double Operand Instructions

andb

Operation: Condition Flag ← Bit of Source1 ∧ Bit of Source2

Assembler Syntax:

- andb rS1,rS2,B1,B2 and specified bits

The andb operation is used to and a bit of one register with a bit of another register and set the condition flags. It is supported by the B-Type instruction format.

B-TYPE | OPCODE | SRC1 | SRC2 | ADR | BIT1 | BIT2 | X

clrb

Operation: Condition Flag ← Clear Source Bit Field

Assembler Syntax:

- clrb rS1,rS2,B1,B2 clear specified bits

The clrb instruction is used to clear a field of bits of the specified register and sets the condition flag. It is supported by the B-Type instruction format.

B-TYPE | OPCODE | SRC1 | SRC2 | ADR | BIT1 | BIT2 | X
A REAL-TIME RISC PROCESSOR

orb

Operation: Condition Flag ← Bit of Source1 ∨ Bit of Source2

Assembler Syntax:

- orb rS1,rS2,B1,B2 or specified bits

The orb instruction is used to perform an or operation between specified registers and set the condition flag. It is supported by the B-Type instruction format.

```
B-TYPE | OPCODE | SRC1 | SRC2 | ADR | BIT1 | BIT2 | X
```

setb

Operation: Condition Flag ← Set Source Bit Field

Assembler Syntax:

- setb rS1,rS2,B1,B2 set specified bits

The setb instruction is used to set a field of bits of the specified register and set the condition flag. It is supported by the B-Type instruction format.

```
B-TYPE | OPCODE | SRC1 | SRC2 | ADR | BIT1 | BIT2 | X
```
A REAL-TIME RISC PROCESSOR

xorb

Operation: Condition Flag ← Bit of Source1 ⊕ Bit of Source2
Assembler Syntax:

- xorb rs1,rs2,B1,B2  xor specified bits

The xorb instruction is used to perform an exclusive or operation between specified bits of registers and set the condition flag. It is supported by the B-Type instruction format.

B-TYPE [OPCODE SRC1 SRC2 ADR BIT1 BIT2 X]

Single Operand Instructions

ff0b

Operation: Destination ← First Bit Zero of Register

Assembler Syntax:

- ff0b rs1,rs2,B1,B2  find first bit clear

The ff0b instruction is used to perform a search for occurrence of the first bit that is found clear in the specified register. This instruction is supported by the B-Type instruction format.

B-TYPE [OPCODE SRC1 SRC2 ADR BIT1 BIT2 X]
A REAL-TIME RISC PROCESSOR

ff1b
*Operation: Destination ⇔ First Bit One of Register*

Assembler Syntax:

- \( ff1b \ r1, r2, b1, b2 \) find first bit set

The \( ff1b \) instruction is used to perform a search for occurrence of the first bit that is found set in the specified register. This instruction is supported by the B-Type instruction format.

**B-TYPE**

```
| OPCODE | SRC1 | SRC2 | ADR | BIT1 | BIT2 | X |
```

**Flow Control Instructions**

beqz
*Operation: Destination Address ⇔ Offset + Register*

Assembler Syntax:

- \( brch \ cond, offset \) branch equal zero

The \( beqz \) instruction branches upon reading a bit of the processor status word. The instruction branches if the bit is set indicating that the previous operation produced a zero. It is supported by the J-type instruction format.

**J-TYPE**

```
| OPCODE | TARG | COND | ADR | OFFSET |
```

bnez
*Operation: Destination Address ⇔ Offset + Register*

Assembler Syntax:

- \( brch \ cond, offset \) branch not equal zero

The \( bnez \) instruction branches upon reading a bit of the processor status word. The instruction branches if the bit is reset indicating that the previous operation produced a non-zero result. It is supported by the J-type instruction format.
A REAL-TIME RISC PROCESSOR

j
Operation: Destination Address ← Offset + Register

Assembler Syntax:
• j cond,offset    unconditional jump to address

The j instruction jumps unconditionally to the address generated by offset and register. This instruction is supported by the J-type instruction format.

J-TYPE  |  OPCODE | TARG | COND | ADR | OFFSET |

jr
Operation: Destination Address ← Offset + Register

Assembler Syntax:
• jr cond,offset   jump register

The jr instruction jumps unconditionally to the address given in the register. This instruction is supported by the J-type instruction format.

J-TYPE  |  OPCODE | TARG | COND | ADR | OFFSET |

call
Operation: Destination Address ← Offset + Register

Assembler Syntax:
• call cond,offset call on condition

The call instruction is used to transfer control to an exception or interrupt depending on the condition field. The address is generated by adding the offset and contents of the register. This instruction is supported by the J-type instruction format and register addressing mode.

J-TYPE  |  OPCODE | TARG | COND | ADR | OFFSET |
A REAL-TIME RISC PROCESSOR

ret
Operation: Destination Address $\leftarrow$ Offset + Register

Assembler Syntax:

- ret register return from call

The ret instruction is used to return from a call, or an interrupt or an exception. The return address is read from a designated register. This instruction is supported by the J-type instruction format and the register addressing mode.

J-TYPE | OPCODE | TARG | COND | ADR | OFFSET |
-------|--------|------|------|-----|--------|

Bibliography


Vita

- Syed Asaf Maruf Ali

- Born in Karachi, Pakistan

- Received Bachelor’s degree in Electrical Engineering from the N.E.D. University of Engineering and Technology, Karachi, Pakistan in March, 1990.

- Completed Master’s degree requirements at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in June, 1994.