Processor performance in real-time systems

Roger Johansson
Department of Computer Engineering
Chalmers University of Technology
S–412 96 Göteborg
Sweden.
E-mail: roger@ce.chalmers.se

October 9, 1992

Abstract

During the last decade, RISC (Reduced Instruction Set Computer) processors, introduced mainly in work station applications, have brought excellent performance at low costs. In real time system design, the question arises; How do RISC processors comply to the specific demands of such a system?

This thesis describes seven RISC processors from an architectural point of view. Their ability to perform in a real-time system is elaborated and reported. Finally, real-time system hardware considerations are made from six different designs using three different processors. The system hardware considerations shows that in a real-time system design there is not very much to gain with a modern, general purpose RISC design such as SPARC. On the contrary, while the estimated performance for SPARC was just about the level of THOR, the board area became approximatly 40% larger, the power consumption 70% more and the expected failure became 45 % greater.

This thesis is a revised version of two reports earlier published as a part of the ESTEC "RISC evaluation study ". performed by Saab Space (contract number 8686/89/NL /JG(SC)) during late 1990, namely: "Work Package 3: Survey of commercial RISC processors, Part 2: Detailed Architectural Survey" and "Work Package 4, Evaluation of processor configurations, part 1: Hardware Designs".

Keywords: Hard Real-Time Systems, RISC-architectures.

Contents

List of Tables

List of Figures

Introduction

As computers become smaller, faster and more reliable the range of computer applications has grown. From the computers initial role as equation solvers, their usage has extended into several areas from toys to spacecraft control.

A rapidly expanding area of computer exploitation is applications that *require information processing* in order to carry out their prime function rather than *do the information processing* as a prime function. These types of computer applications are called *real-time systems*. A real-time system can be understood as any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period [You82] . In a *hard real-time system* the ability to respond within a specified time is as important as producing a correct result. That is, if the response or result arrives to late it is of no use. The system will eventually crash or become unable to fulfill it's task. A dedicated application system such as for process control etc is an *embedded system*. Throughout this thesis the terms "real-time system" will be used in the meaning of an *embedded, hard real-time system*. During the last decade, RISC (Reduced Instruction Set Computer) processors, introduced mainly in work station applications, have brought excellent performance at low costs. In real time system design, the question arises; How do RISC processors comply to the specific demands of such a system?

This thesis describes seven RISC processors from an architectural point of view. Their ability to perform in a real-time system is elaborated and reported. Finally, real-time system hardware considerations are made from six different designs using three different processors. The subject will be treated as follows: chapter 1 will recapture the development path leading to today's RISC architectures. In chapter 2, different processors will be described in detail from an architectural point of view. Chapter 3 will give a thorough discussion of real-time systems requirements and how the studied processors meet these demands. A real-time system's hardware requirements tend to degrade the total system performance, which is the reason why hardware considerations are emphasised in chapter 4. Chapter 5 gives concluding remarks.

Seven different processors have been selected for this study. One selection criterion was to include RISC processors commonly used today. The following selection was made:

- "Motorola MC 88100"
- "Intel Iapx80960".

- "MIPS R2000 (R3000)"
- "Cypress SPARC "

Another criterion was to select processors which are claimed by their manufacturers to facilitate real-time system support and to be suitable for this range of applications. From this group of processors the following selection was made:

- "Advanced Micro Devices Am 29000"
- "Inmos T800 transputer"
- "Saab-Ericsson Space THOR"

From lack of sufficient time another selection had to be made for the hardware considerations in chapter 4. The three processors (SPARC, T800 and THOR) that were selected, were considered as providing information representative for the entire group.

This thesis is a revised version of two reports earlier published as a part of the ESTEC "RISC evaluation study ". performed by Saab-Space (contract number 8686/89/NL /JG(SC)) during late 1990, namely: "Work Package 3: Survey of commercial RISC processors, Part 2: Detailed Architectural Survey" and "Work Package 4, Evaluation of processor configurations, part 1: Hardware Designs".

Acknowledgements

I wish to thank my supervisor, Jan Torin, He is a major contributor to this work.

I also thank:

Jiri Gaisler, who pointed out disambiguities in the original reports.

Jonas Vasell, who contributed with valuable aspects on the first three chapters.

Mats Svenningsson, for his willingness of sharing his great knowledge in numerous discussions, his ideas and encouragement.

Arne Carlsson, who shared his great experience from the design and construction of real-time systems.

Chapter 1

The Background Of RISC

1.1 Computer Architecture

A Computer is a high-speed device that performs arithmetic operations and symbol manipulation through a set of machine dependent instructions. A computer consists of several important parts; there are memory systems, input/output devices ranging within a large scale of complexity, the Central Processing Unit (CPU) with datapaths, control unit and other subsystems.

There are at least two principal different ways of managing the central processing. One of these is the data-flow machine, another is the von Neumann- machine. A von Neumann-machine does information processing by sequentially executing algoritms which are organized as programs and stored in a memory. The programs detail interpretation and processing of information coded as data and stored in the same memory. The von Neumann-machine consists consequently of at least one processor that sequentially interprets instructions in the program and a primary memory that stores program and data. These architectures may degrade performance from the so called "von Neumann bottleneck" which means that execution speed is highly dependent of the rate at which primary memory can be accessed, the memory bandwith. This comes from the fact that code (processor instructions) and data resides in the same memory and are accessed sequentially. Hence, the presence of data obstructs the speed of instruction fetching. This is a fact with influence on RISC design considerations.

The principle of a "stored program" or a von-Neumann architecture can be implemented in several ways which has also been done. To distinguish between different von Neumann-architectures we speak more generally about computer architecture. This concept, created by Amdahl while working with the IBM 360, can be summarized as:

> The image that the computer presents to the machine language programmer and the compiler writer.

Consequently, the processors instruction set, its registers, and other details that are essential for programming the device. The coding and interpretation of a program constitutes the instruction set, thus, this is a main component of a computer architecture. The register file is heavily utilized by a compiler writer, thus it is another major component of the architecture. Different instructions exhibit different execution times, therefore in some special occasions, there is need for the programmer to know something about the CPU-datapaths or at least the instruction timing.

Recently the term "computer architecture" has been given an extended meaning, [Hen90], which makes it cover computer hardware and computer organization as well. For the subject as treated in this work however, Amdahls definition will suffice.

1.2 Trends in computer architectures

To gain understanding of the design decisions behind RISC-machines it is necessary to recapture the historical development of processors and their instruction-sets. Ever since the first digital processing units, the instruction sets have been extended and the instructions have grown in complexity. The MARK-1 (1948) had seven quite simple instructions while a mainframe from the late seventies such as VAX has over 300 instructions. Some of these instructions are extremely complex requiring a large amount of hardware and several clock cycles to be executed. This, in turn, leads to sophisticated technics for pipelining, prefetching and the use of cache memories. This development, from small and simple to large and complex instruction-sets is remarkable when it comes to single chip processors. For example, if comparing the Motorola 6800 with the 68020 we find that eleven new addressing modes have been added, the number of instructions has doubled, new functions have been added for instruction caches and coprocessors. Furthermore the instructions complexity has grown tremendously.

The general trend towards modern CISC (Complex Instruction Set Computer) is a result of several factors. New models within a computer family have to be compatible with their predecessors. As a result the number of functional units in the processor increases. In this way new functions can be added in new machines without wasting earlier software development efforts. Several efforts have been done to decrease the "semantic gap" between high level programming languages and the instruction set. This has been done by implementing instructions that were close to the high level statements. Such instructions have a tendency of being extremely complex and not applicable for every possible language. Thus, it turns out that the compiler can not make use of these special instructions. Meanwhile these instructions require a lot of hardware which in many cases increases the processor cycle time.

To make the machines run faster, designers have moved functions from assembly program to microcode and further on from microcode to hardware. By adding extra hardware in the decoding unit one could get to a point where a machine cycle has to be lengthened. Thus, adding a certain instruction may slow down the execution of every instruction in the set. Development tools and methods used in the design of large VLSI circuits, is a

support for design of large architectures.

Microcoding is a particular interesting technic that encourages complex instructions. It is a structured way of implementing, creating and modifying those algoritms that control the execution of complex instructions in the processor. The steady grow of CISC-functions is further supported by large micromemorys. It is easy to add a new instruction if only there is room enough in the micromemory.

1.3 Considerations that lead to the RISC

At least historically, in most computer applications, a program written in assembly language exhibits the shortest execution times. This has been due to the fact that assembly language programmers know the computer architecture well and are capable of taking every advantage of it. It is difficult to accomplish this in an automatic manner and for general cases which are the requirements for compiler to generate code. However, assembly language programming, as a way of increasing program performance suffers from some heavy disadvantages. It is probably the most time-consuming method to write software. Thus it is very expensive and yields results much later than high level programming. Hence, for a new processor architecture theres has to be a compiler for a high level language.

It has been found that it is difficult to construct an efficient compiler for a computer with a large instruction set. The compiler cannot make use of all of the sophisticated instructions that the architecture offers. Therefore, the compiler uses simpler instructions and generates larger code, thus making programs run slower, and wasting primary memory in a way that should not be needed if an assembly language programmer wrote the same piece of code. With the experience of these facts some designers began to question whether CISCs are as fast as they could be, bearing the capabilities of the underlying technology in mind. A few designers offered the hyphothesis that increased performance should be possible through a streamlined design and instruction set simplicity, hence a Reduced Instruction Set Computer [MIP87].

Consider this expression for processor performance,

$$P = \frac{Time}{Task} = C\ T\ I$$

where:

- C = cycles/instruction
- T = time/cycle
- I = instructions/task

It is clear that P should be kept as small as possible under given the circumstances. There must be at least three different ways of minimizing P.

1. Reduce the number of cycles per instruction.
2. Reduce the time per cycle.
3. Reduce the number of instructions per task.

Let us have a closer look at each of these.

1. The cycle time could be made very small through pipelining technics. I.e, several instructions can be executed simultaneously, each one occupying different stages of the pipeline. This will keep most of the hardware busy most of the time. The cycle time will be equivalent to the slowest stage in the pipeline. Hence, pipelining is a way of reducing C.
2. T can only be kept low through the use of instructions that can be decoded and executed by non-complex, and thereby fast, subsystems, therefore, keeping instructions simple will decrease T.
3. I can, theoretically, be made as low as 1, I.e when there exists an instruction for each high-level program construction that a task can constitute. This is hard to achieve but the principle is clear. Complex instructions are required to minimize I.

As we can see, there is no way of meeting all of these requirements at the same time. In fact, there are several contradictions in the requirements such as 1) and 3), 2) and 3), and a closer look will show even more.

The RISC approach is to reduce C and T. This can only be done at "the cost of" I. To minimize this cost, one attempts to reduce I with the aid of highly optimizing compilers. Therefore, one must bear in mind, that the absence of such program development tools will dramatically affect a RISC system.

1.4 A RISC design decision graph

The RISC approach leads to several design decisions. Figure 1.1 illustrates how fundamental criteria lead to design decisions that constitutes a RISC-processor.

An attempt to acheive single cycle execution, i.e reduce C, without affecting cycle time T leads to a *pipe-lined architecture*. The pipe-line should be divided into stages wich all meet the cycle-time requirement stated as T.

To fully exploit the advantages of a pipe-line, a uniform instruction fetch and execution must be accomplished. This may possibly be disturbed by data-dependencies which prevent an early stage of an instruction from being executed before a later stage of the preceeding instruction has been completed. Changes in program flow forces a stop/flush and refill of the pipe-line. A score- board mechanism that indicates registers in use will

detect data- dependencies. Pipe-line forwarding technique may prove helpful for reducing the penalties. Delayed branch, (which means that the instruction immediatly following a branch, conditional or unconditional is always executed) is used to reduce penalty associated with changes in program flow. However, this requires a careful strategy by the compiler. Optimising compilers could take advantage from this feature.

A uniform instruction execution can only be acheived by using uniform instructions. This leads to a rather simple and reduced instruction set. Data should be accessed within a single cycle, therefore a large, on chip, register file is needed in the top of the memory hierarchy. Since instructions/addressing modes should be kept simple, and data should be kept in registers there are strong implications for special load/store instructions that perform data traffic, hence the commonly used name *load/store- architecture*.

A large register file will create significant 'overhead' in the case of context switch. A special support for such occasions is therefore needed. Optimising compilers could provide such support. Register windows is another way of reducing context switch overhead.

Approximately 20 percent of the executed instructions are used about 80 percent of the time spent executing a program [Rad83], the so called "20/80-rule" . Analysing the instruction mix shows that simple instructions dominate among these 20 percent [Hen90]. We can see strong needs for careful code generation or the increase of performance may be outbalanced by an increase of static and dynamic instruction count. This is a very strong implication for optimizing compilers.

For implementation, a constant chip area should be maintained. A simple decoding logic saves chip and implies simple instructions.

Uniform instruction execution demands uniform instruction fetch. One instruction should be fetched in each cycle but disturbances from data traffic make this difficult to acheive. Since the memory bandwidth is assumed to be constant we have another implication for a large on-chip register file.

We may thus conclude: The RISC high performance relies heavily on : low cycle time, single cycle execution which implies a Reduced Instruction Set with simple, uniform instructions and efficient optimising compilers.

1.5 Early RISCs

The RISC concept was, in fact, adapted very early by Seymour Cray in an effort to design a very fast vector processor. The CDC 6600 was register based and all operations used data from registers local to the arithmetic units. The instruction set was simple and executions were pipelined. Cray realized that all operations must be simplified for maximal performance. One bottleneck in processing may cause all other operations to degrade performance.[Sie82]

Starting in the mid 1970s, the IBM 801 research team investigated the effect of a small

Figure 1.1: A Risc Design Decision Graph

instruction set and optimizing compiler design on computer performance . They performed dynamic studies of the frequency of use of different instructions in application programs. In these studies, they found that approximately 20 percent of the available instructions were used 80 percent of the time. Also, complexity of the control unit necessary to support rarely used instructions slows the execution of all instructions. Thus through careful study of program characteristics, one can specify a smaller instruction set consisting only of instructions which are used most of the time, and are executed quickly.[Rad83]

The first major university RISC research project was at the University of California, Berkeley . David Patterson, Carlos Séquin and a group of graduate students investigated the effective use of VLSI in microprocessor design. The Berkeley RISC concept was adopted by Sun Microsystems where the SPARC architecture was defined.[Pat82]

Shortly after the Berkeley group began its work, researchers at Stanford University, under the direction of John Hennessy , began looking into the relationship between computers and compilers. Their research evolved into the design and implementation of optimizing compilers and reduced instruction sets. Since this research pointed to the need for single cycle instruction sets, issues related to complex,deep pipelines were also investigated. This research resulted in a RISC processor for VLSI that is commonly referred to as "the Stanford MIPS" (Microprocessor without Interlocked Pipeline Stages). [Hen84]

1.6 A brief overwiev of some RISC projects

Berkeley SPUR (Symbolic Processing Using RISC) is a multiprocessor research machine for investigations in paralell processing [Hil85] [Hil86]. The SPUR processor is a general-purpose RISC with support for LISP and floating point arithmetic. From 6 to 12 SPUR processors may be attached to shared memory and shared I/O devices by the SPUR bus.

University of Wisconsin PIPE (Parallel Instructions and Pipelined Execution) project was an attempt to reduce three common processor bottlenecks with a reduced architecture [Smi83]. In the PIPE, programs are decomposed in separate address and computation tasks. Two independent identical processors performs these tasks. An access processor is responsible for all memory addressing and access operations. An execute processor performs all data processing.

Reading University RIMMS(Reduced Instruction Set architecture for Multi- Microprocessor Systems) resulted from a study of CPU design for SIMD and MIMD multiprocessor systems [Mil83]. The research group saw that the performance gains through concurrency have the potential beeing much more significant than performance gains throuh increased device speeds.

The Ben-Gurion University MODHEL RISC system [Tab87] was intended as an investigation tool in the study of RISC computing systems . The MODHEL system can be used in experiments with benchmark programs in studies aimed at finding an optimal instruction set.

Hewlett-Packard has developed a family of computers based upon RISC design. Two of these computers, the Series 930 and the Series 950 are realizations of the *HP Precision Architecture* [Bir85] RISC-type system.

The IBM 6151 RT PC is basically a workstation which uses the IBM ROMP (Research Office products division MicroProcessor) and a MMU (Memory Management Unit) [Hin86] The ROMP/MMU represents one of the commercial spinoffs from the IBM 801 research project.

Chapter 2

Description Of RISC Architectures

In this chapter a detailed description of seven RISC processors, mostly from an architectural point of view, will be given. Basic features that will be described are:

- Instruction Set
- Data formats
- CPU register description
- Instruction formats and addressing modes
- Processor states

The following literature was chosen as sources (See the bibliography for a complete reference): "MC88100 RISC microprocessor user's manual" [Mot90], "80960KB programmer's reference manual" [Int88], "MIPS R2000 RISC architecture" [MIP87], "SPARC RISC user's guide" [ROS90], "The Transputer databook" [Inm89], "Am29000 streamlined instruction processor user manual" [Adv88], "THOR, Stack RISC microprocessor instruction set architecture for prototype chip"[Saa92]. For THOR, additional information was gathered from draft-issues of a forthcoming user's manual.

The purpose of this chapter is to give a standardised description of the selected RISC processors. The varying ways of implementing floating point support, memory management etc, will only be mentioned briefly and no detailed descriptions will be given.

2.1 Motorola MC88100

In early 1988, Motorola Inc. presented 88000. The basic architecture consists of a processor chip, MC88100 and two identical cache chips, MC88200. This offers a full system solution for a reduced instruction set architecture. The MC88100 has capability for concurrent operations. There are four execution units: the Integer/Bit-Field Unit and the Floating Point Unit execute data manipulation instructions. The Data Unit performs data memory accesses while the Instruction Unit performs instruction prefetches. There are separate data and instruction memory ports (Harvard Bus Structure) and pipelined Load and Store operations. The MC88100 also has three internal buses; a source 1 bus, a source 2 bus and a destination bus that are used for passing operands between the register file and the different execution units.

2.1.1 MC88100 instruction set

The MC88100 instruction set contains 51 instructions. All integer arithmetic, logical, bitfield and certain flow-control instructions execute in a single clock cycle. Memory access and floating point instructions are performed by dedicated execution units. All instructions are implemented directly in hardware, precluding the need for microcoded operations. An instruction set summary is given in appendix.

2.1.2 MC88100 data formats

- Integer signed (2's complement) and unsigned data formats: 64-bits (double word), 32-bits (word), 16- bits (half-word), 8-bits (byte). Data items are aligned so that they do not cross word boundaries, i.e half-words may have only even addresses, words may have addresses divisible by four, double words may have addresses divisible by eight and byte data may be placed at any address. An attempt to cause misaligned access causes an exeption (if enabled).
- Signed and unsigned bit fields from 1 to 32 bits.
- IEE 754 single precision (32 bits) floating point. IEE 754 double precision (64 bits) floating point

Bytes and half-words are packed, in memory, according to the "little endian" or the "big-endian"-scheme. The byte ordering in effect is controlled by a bit in the processor status register. A signed byte or half-word stored in a register is automatically signed-extended. Data is placed in the least significant part while remaining bits are filled with the sign of the data value. In the case of unsigned byte or half-word the most significant part of the register is filled with zeros. The least significant bit in a data item is denoted b0, the next bit b1 and so on.

2.1.3 MC88100 registers

The register set consists of general-purpose registers, registers dedicated for floating point operations and control-registers. There are also some internal registers, not available in any of the register models; they can only be used and modified indirectly.

General Purpose registers

r0-r31 (table 2.1)contain program data. Their usage are dedicated due to software conventions (further discussed in chapter 3). All of these registers with the exeption of r0 (constant zero) has read/write access. A write operation to r0 has no effect.

Floating-point operation registers

fcr1-fcr7 are used to hold floating point operands and results while the rest holds various status from the floating-point unit (table 2.2).

Control Registers

Control registers (table 2.3) contain status, execution control and exception processing information. Some of the registers have read/write access; others are read only.

Internal Registers

Internal registers (table 2.4) located in the register file/sequencer and instruction unit control instruction execution and data availability. These registers are not explicitly accessible for the programmer.

2.1.4 MC88100 instruction formats/addressing modes

All instructions are 32 bits in length. Immediate operands and displacements are encoded in the instruction word. All other operands are located in registers which can be moved to and from memory with load and store instructions.

There are three instruction types: flow control, data memory accesses and register to register operations. Each type has unique addressing capabilities. Flow control instruction references are made by the instruction unit. Data memory access instructions address those sections of memory that contain program data. Register to register instructions access only the general purpose registers or, in some cases, the control registers.

name	*proposed usage*
r0	zero
r1	subroutine return pointer
r2-r9	called procedure parameter registers
r10-r13	called procedure temporary registers
r14-r25	calling procedure reserved registers
r26	linker
r27	linker
r28	linker
r29	linker
r30	frame pointer
r31	stack pointer

Table 2.1: MC88100 general purpose registers

name	*usage*
fcr0	f.p. exeption cause register
fcr1	f.p. source operand 1 high register
fcr2	f.p. source operand 1 low register
fcr3	f.p. source operand 2 high register
fcr4	f.p. source operand 2 low register
fcr5	precise operation type register
fcr6	f.p. result high register
fcr7	f.p. result low register
fcr8	f.p. imprecise operation type register
fcr62	f.p. user status register
fcr63	f.p. user control register

Table 2.2: MC88100 floating point registers

name	*usage*
cr0	processor identification register
cr1	processor status register
cr2	exeption time processor status register
cr3	shadow scoreboard register
cr4	shadow execute instruction pointer
cr5	shadow next instruction pointer
cr6	shadow fetched instruction pointer
cr7	vector base register
cr8	transaction register 0
cr9	data register 0
cr10	address register 0
cr11	transaction register 1
cr12	data register 1
cr13	address register 1
cr14	transaction register 2
cr15	data register 2
cr16	address register 2
cr17	supervisor storage register 0
cr18	supervisor storage register 1
cr19	supervisor storage register 2
cr20	supervisor storage register 3

Table 2.3: MC88100 control registers

name	*function*
XIP	eXecute Instruction Pointer contains the address of the instruction that is currently being executed.
NIP	Next Instruction Pointer contains the address of the instruction that is currently being received from memory and decoded by the instruction unit.
FIP	Fetch Instruction Pointer points to the memory location of the next accessed instruction. For sequential execution FIP=XIP+4. Jump target addresses are received from the jump instruction operand. Unconditional branch addresses are computed from the XIP and a 26-bit signed displacement, i.e. FIP=XIP+d26. Conditional branch addresses for the branch taken case are calculated as FIP=XIP+d16.
SB	Scoreboard Register contains a bit corresponding to each register r1-r31. If a bit is set the corresponding register is currently in use.

Table 2.4: MC88100 internal registers

Register to Register Instructions

Depending on instruction this format provides four addressing modes.

1. *Triadic Register Addressing* uses three five-bit fields to specify two source register fields `S1,S2` and a destination register field `D`. The `OPCODE` field directs processing to the integer unit or the floating point unit. Not every instruction uses all three register selection fields. For arithmetic and logical instructions there is a `SUBOPCODE` field wich specifies the full operation

2. *Register with 10-bit immediate addressing* is used in bit-field instructions. Data in `rS1` is processed and the result is placed in `rD`. The 10- bit immediate value represents

Triadic register		10-bit immediate	
bits	*encoding*	*bits*	*encoding*
31-26	`OPCODE`	31-26	`OPCODE`
25-21	`D`	25-21	`D`
20-16	`S1`	20-16	`S1`
15-5	`SUBOPCODE`	15-10	`SUBOPCODE`
4-0	`S2`	9-0	`IMM10`

Table 2.5: MC88100 Triadic register and 10-bits immediate instruction formats

16-bit immediate		control register	
bits	*encoding*	*bits*	*encoding*
31-26	OPCODE	31-26	OPCODE
25-21	D	25-21	D
20-16	S1	20-16	S1
15-0	IMM16	15-14	OP
		13-11	SFU
		10-5	CRS/CRD
		4-0	S2

Table 2.6: MC88100 16-bit immediate and control register addressing instruction formats

two 5-bit fields specifying the bit-field width and offset respectively.

3. *Register with 16-bit immediate addressing* is used by arithmetic and logical instructions requiring a 16-bit (unsigned) immediate value.This value is zero-extended before processed by any arithmetical instruction.

4. *Control Register Addressing* is used to reference the general control and FPU control registers. General purpose registers may be loaded from, stored to or exchanged with the control registers. The CRS/CRD field specifies the control register which is a source register in the case of a load instruction, a destination register otherwise. The D-field specifies a general-purpose register that is loaded with the contents of the selected control register. This field is ignored in store operations. The S1 field specifies the general purpose register whose contents are to be transferred to the selected control register. This field is ignored in load instructions. The OP field identifies the particular instruction. The SFU field specifies a *special function unit* accessed by the instruction: the value zero specifies the integer control unit registers, the value one specifies the floating point unit registers. Other values (2-7) cause an SFU precise exception for the addressed SFU. The S2 field finally, must contain the same value as the S1 field (for decoding purposes).

Data Memory Access Instructions

MC88100 supports three adressing modes for accessing data in memory or to generate a memory address. Address calculations are performed by the use of unsigned arithmetic. Overflows are not detected and results are truncated to the number of available bits.

1. *Register Indirect with 16-bits zero-extended immediate index.*

   The contents of rS1 is added to the 16-bit zero- extended immediate index contained in the I16 field of the instruction. The result is a data memory address. This address is:

   - (for LDA instruction) loaded into the register specified by the D field
   - (for STORE and EXCHANGE instructions) used as the memory address where contents of D field register are stored

immediate index		register index	
bits	*encoding*	*bits*	*encoding*
31-26	OPCODE	31-26	OPCODE
25-21	D	25-21	D
20-16	S1	20-16	S1
15-0	I16	15-5	SUBOPCODE
		4-0	S2

Table 2.7: MC88100 indexed addressing instruction formats

- (for LOAD instruction) used as the memory address from which the D field register is loaded.

2. *Register indirect with index* is similar to the previous mode but contents of register specified by the S2 field are used as index rather than as immediate value. SUBOPCODE field specifies the particular instruction.

3. *Register indirect with scaled index* The index is scaled by the size of the access before it is used in the address calculation. Here, SUBOPCODE specifies the particular instruction as well as the scaling factor.

Flow Control Instructions

Flow control instruction address or reference instruction memory by the use of four different addressing modes. Address calculations are performed using signed arithmetic. Overflows are not detected and results are truncated to the number of available bits.

1. *Triadic Register Addressing* is used to specify the target of a jump instruction or the operands of a trap-on-bound instruction. All three of the operands do not have to be used. The SUBOPCODE identifies the particular instruction. For jump instructions the S2 field specified register contents are placed in the FIP, causing program execution to be transferred to that address. The lower two bits of S2 field register are ignored so that FIP contains a word address. The S1 and D fields are ignored. For trap-generating bound-checks instructions the data in registers specified by S1 and S2 fields are compared. A trap is taken if the source 1 data is greater than the source 2 data (unsigned). The D field is ignored. If the trap is taken, execution is transferred to the bound check exception vector by concatenation of the VBR, bounds-check exception vector and three trailing zeroes, forming a 30-bits instruction address.

2. *Register with 9-bit vector table index* is used by bit test trap instructions where the bit in S1 field register specified by the B5 field is tested for either a set or clear condition. It is also used by the conditional trap instructions where the source 2 register is tested for the conditions specified in the M5 field (see below). In either case, if the test condition is true, the contents of VBR is concatenated with the VEC9 field of the instruction and three trailing zeroes. Exception processing starts

triadic register		9-bit vector table	
bits	*encoding*	*bits*	*encoding*
31-26	OPCODE	31-26	OPCODE
25-21	D	25-21	B5/M5
20-16	S1	20-16	S1
15-5	SUBOPCODE	15-9	SUBOPCODE
4-0	S2	8-0	VEC9

Table 2.8: MC88100 Flow control; triadic register and 9-bit vector table index instruction formats

at the vector specified by the resulting address. The SUBOPCODE field specifies the particular instruction. The M5 field specifies which out of four possible conditions to test out:

- **bit 25** Reserved, must be zero
- **bit 24** Maximum negative number
- **bit 23** Less than zero
- **bit 22** Equal to zero
- **bit 21** Greater than zero

Note that multiple conditions can be specified by setting more than one bit in this field.

3. *Register with 16-bit displacement/immediate* is used by branch and trap instructions for target address and test condition generation. The OPCODE field identifies the particular instruction. For bit test branch instructions the bit in source 1, specified by the B5 field is tested for either a set or clear condition. For condition test branch instructions source 1 is tested for the condition(s) specified in the M5 field. In either case, if the test condition is true, the 16-bit displacement specified in the instruction D16 field is shifted left two positions and sign-extended to 32 bits. This value is added to the XIP and the result is loaded into FIP, thus program execution is transferred to that address. For trap-generating bound-check instructions the data in source 1 is compared to the specified immediate operand. A trap is taken if the register data is greater than the (unsigned) operand. If the trap is taken, the bounds-check vector number is combined with VBR, the result is concatenated with three trailing zeroes and loaded into the FIP. Exception processing begins from the bounds-check exception vector.

4. *26-bit branch displacement*

   This form is used to specify the branch target instruction in unconditional branch instructions which use a sign-extended 26- bit displacement to calculate the location of a new target instruction. The displacement is shifted left by two bits and sign-extended to 32 bits. The two least significant bits are cleared to force word alignement. This value is then added to the XIP to form the address of the target instruction. The computed address is placed in the FIP, causing program execution to be transferred to that address. The OPCODE field identifies the particular branch instruction.

16-bit displacement		26-bit displacement	
bits	*encoding*	*bits*	*encoding*
31-26	OPCODE	31-26	OPCODE
25-21	B5/M5	25-0	D26
20-16	S1		
15-0	D16		

Table 2.9: MC88100 16-bit displacement and 26-bit displacement instruction formats

2.1.5 MC88100 processor states

The MC88100 may be in one of three states:

- Normal instruction execution
- Exception
- Reset

Normal Execution

During normal execution the processor operates at either the supervisor or user level of privilege. These levels defines which memory space is accessed during external bus transactions and which registers are available to the programmer. When operating in supervisor mode memory access reference the supervisor address space in data or instruction memory. This mode allows execution of all instructions and allows access to all control registers and general purpose registers.

Kernel software typically executes in supervisor mode. The kernel may provide services such as resource allocation, exception handling and software execution control. Execution control normally includes control of user programs and protecting the system from accidental corruption by a user program.

The user mode changes to supervisor mode if:

- an exception occurs
- a reset is signalled
- a trap instruction is executed by a user program
- an interrupt or memory access fault occur

Exceptions

Exceptions are conditions that causes the processor to suspend execution of the current stream and perform exception processing. Exceptions can occur at any time during normal instruction execution. Exceptions are recognized internally when the processor is between instructions.

Exceptions occur due to to four types of conditions:

- Interrupts which are signalled externally
- Externally signaled errors (such as bus errors)
- Internally recognized errors (such as zero-divide)
- Trap instructions

The processor begins exception handling at the next instruction boundary after the event is recognized. It freezes the execution context in "shadow-" and "exception time registers", which also precludes other interrupts from occuring, and enters the supervisor mode. The FPU is disabled and the data unit is allowed to complete pending accesses. Instruction execution transfers in an orderly manner to the appropriate interrupt handler routine which is defined by the "exception vector" associated with that particular interrupt.

Exceptions fall into two categories: precise and imprecise. With a precise exception, the exact processor context, when the exception occured, is available, and the exact cause of the exception is always known. With an imprecise exception, the exact processor context is not known when the exception is processed. The context is not known because concurrent operations have affected the information that comprises the processor context.

The integer unit maintains copies of certain internal registers for use during MC88100 exception processing. The data unit and FPU also maintain copies of internal registers to allow full recovery when exceptions occur. The copies of internal registers are referred to as shadow registers and are updated on every clock cycle when shadowing is enabled. For shadowing to occur, it must be specifically enabled. This may be done by clearing the "shadow freeze bit" in PSR or by executing an rte-instruction. The shadow freeze bit is set by hardware when an exception is processed in order to preserve the processor context.

"Exception vectors" are entry points into the interrupt handler routines. The MC88100 maintain a vector table consisting of 512 exception vectors on a 4 KB memory page pointed to by the vector base address in the "vector base address register" (VBR).

Each interrupt and "exception vector" has a corresponding number which is generated by hardware or specified as a nine-bit field in a trap instruction. This number is used as an index into the vector table. Each "exception vector" is two instructions (eight bytes)

long. "Exception vectors" 0-127 are reserved for various events while "exception vectors" 128-511 are user defined.

Due to concurrent execution units of the MC88100 multiple exceptions can occur at the same time whithin the processor. When this happens they are recognized by the processor according to a predefined priority. Exceptions that have the same priority never occur simultaneously.

2.1.6 MC 88100 pipelining

There are four separate execution units which allow MC88100 to perform up to five different operations simultanously:

- Access program memory
- Execute an arithmetic ,logical or bit-field instruction
- Access data memory
- Execute floating point or integer divide instruction
- Execute floating point or integer multiply instruction

The instruction unit pipeline supplies the appropriate execution unit with instructions that are to be executed by a concurrent pipeline. Data memory access instructions are dispatched to the data unit, whereas floating point ,integer multiply and integer divide instructions are dispatched to the FPU. The FPU contains two pipelines handling floating point add, subtract, compare and conversions between integer and floating-point, as well as integer and floating-point divide instructions. All other instructions are executed by the integer unit, or instruction unit for branches, in one machine cycle.

All execution units contain an additional level of parallelism. Instruction decode and source operand fetches from the registers are performed simultanously. Branch instruction decode and branch target address calculation are performed in parallel with the next instruction fetch. Three internal register buses allow three simultaneous register accesses.

2.2 Intel 80960KB

The 80960KB is an implementation of the 80960 32-bit architecture from Intel. This architecture has been designed to meet the needs of embedded applications such as machine control, robotics, process control, avionics and instrumentation.

The architecture provides 32 registers, 28 of which are available for general use. These are divided into two types; globals and locals. There is a 512 byte instruction cache on chip and multiple set of local registers. Execution of some instructions may me overlapped. This is accomplished by register scoreboarding.

2.2.1 80960 KB instruction set

The 80960 KB processor implements all the instructions in the 80960 instruction set, which includes all of the data movement, arithmetic, logical, and program control instructions commonly found in computer architectures. The processor also includes a set of floating-point instructions and several instructions to handle architectural extensions found in the processor. All instructions are 32 bits long and aligned on 32 bit boundaries. There are over 50 instructions that can be executed in a single clockcycle. A summary of the 80960 KB instruction set is given in Appendix B.

The processor provides a mode and stack switching mechanism called the user-supervisor protection model. This protection model allows a system to be designed in which kernel code and data resides in the same address space as the user code and data, but access to the kernel procedures (called supervisor procedures) is only allowed through a controlled interface. This interface is provided by the system procedure table.

2.2.2 80960KB data formats

The 80960KB operates on seven data types. Integer, real, ordinal and decimal data types can be thought of as numeric data types. The remaining types, bit- field, triple word and quad word, represent grouping of bits or bytes that the processor can operate on as a whole, regardless of the nature of the data contained in the group.

Integers are signed whole numbers, which are stored and operated on in two's complement format. The processor recognizes four sizes of integers: 8-bit (byte integers), 16 bit (short integers), 32-bit (integers) and 64-bit (long integers).

Ordinals are a general purpose data type. The processor recognizes four sizes of ordinals: 8-bit (byte ordinals), 16-bit (short ordinals), 32-bit (ordinals), and 64-bit (long ordinals). The processor uses ordinals for both numeric and non- numeric operations. For numeric operations, ordinals are treated as unsigned whole numbers. The processor provides several arithmetic instructions that operate on ordinals. For non-numeric operations, ordinals contain bit-fields, byte strings, and Boolean values.

Reals are floating point numbers. The processor recognizes three sizes of reals: 32-bit (reals), 64- bit (long reals), and 80-bit (extended reals). The real number format conforms to the IEEE standard for binary floating point arithmetic.

The processor provides three instructions that perform operations on decimal values when the values are presented in ASCII-format. Each decimal digit is contained in the least significant byte of an ordinal (32 bits). For decimal operations, bit 8 through 31 of the ordinal containing the decimal are ignored.

An individual bit is specified for a bit operation by giving its bit number in the ordinal in which it resides. The least significant bit of a 32 bit ordinal is b0. The most significant bit is b31. A bit-field is a contignous sequence of bits of from 0 to 32 bits in length within a 32-bit ordinal. A bit field is defined by giving its length in bits and the bit number of its lowest numbered bit.

Triple and Quad words refer to consecutive bytes in memory or in registers; a triple word is 12 bytes and a quad word is 16 bytes. These data types facilitate the moving of blocks of bytes.

2.2.3 80960KB registers

The processor provides three types of data registers: global, floating-point and local. The 16 global registers (g0-g15) constitute a set of general purpose registers, the contents of which are preserved across procedure boundaries. The 4 floating point registers are provided to support extended floating point arithmetic. Their contents are also preserved across procedure boundaries. The 16 local registers (r0-r15) are provided to hold parameters specific to a procedure. For each procedure that is called, the processor allocates a separate set of 16 local registers. For any one procedure within a program, 36 registers are thus available; the 16 global registers, the 4 floating point registers and the 16 local registers. These are all maintained on the processor chip.

Global Registers

The 16 global registers are 32-bits registers. Registers g0 through g14 are general purpose registers, g15 is reserved for the current frame pointer (FP). The FP contains the address of the first byte in the current stack frame.

Floating-Point Registers

The four floating-point registers (fp0 through fp3) are 80-bits registers. These registers can be accessed only as operands of floating-point instructions. All numbers stored in these registers are stored in extended real format. The processor automatically converts floating point values from real or long-real format into extended real format when a floating point

register is used as a destination for an instruction.

Local Registers

The 16 local registers are 32-bits registers, like the global registers. The purpose of the local registers is to provide a separate set of registers aside from the global and floating point registers, for each active procedure. Each time a procedure is called, the processor automatically sets up a new set of local registers for that procedure and saves the local registers for the calling procedure.

Local registers r0 through r2 are reserved for special functions as follows: register r0 contains the previous frame pointer (PFP), r1 contains the stack pointer (SP) and r2 contains the return instruction pointer (RIP). The processor accesses the local registers at the same speed as it does the global registers.

Register Scoreboarding

A mechanism called register scoreboarding can, in certain situations, permit instructions to execute concurrently. While an instruction is being executed, the processor sets a scoreboard bit to indicate that a particular register or group of registers is being used in an operation. If the instruction that follows does not use registers in that group, the processor, is in some instances able to execute those instructions before execution of the prior instruction is complete.

Instruction Pointer

The instruction pointer (IP) is the address of the instruction currently being executed. This address is 32 bits and the 2 least significant bits are always zero. Instructions in the processor are one or two words long. The IP gives the address of the lowest order byte of the first word of the instruction.

Arithmetic Controls

The processor arithmetic controls are made up of a set of 32 bits. These bits include condition codes, floating-point control and status bits, integer control and status bits and a bit that controls faulting on imprecise faults, i.e faults where the entire processor status is not known.

bits	*encoding*
31-24	OPCODE
23-19	SRC/DST
18-14	SRC2
13	M3
12	M2
11	M1
10-7	OPCODE
6-5	0
4-0	SRC1

Table 2.10: 80960KB REG-instruction format

Process and Trace Controls

The processors process controls are a set of 32 bits that control or show the current execution state of the processor. The trace controls are a set of 32 bits that control the tracing facilities of the processor.

2.2.4 80960KB instruction formats

All of the 80960KB instructions are one word long and begin on word boundaries. One group of instructions allows a second word which contains a 32-bit displacement. There are four basic instruction formats: REG,COBR,CTRL and MEM. Each instruction has only one format which is defined by the opcode field of the instruction.

REG format

The REG-format (Table 2.10) is for operations that are performed on data contained in the global, local or floating point registers.

The opcode is 12 bits long and is split between bits 7 through 10 and bits 24 through 31. The SRC1 and SRC2 operand fields specify source operands for the instruction. The operands can be either literals or registers. The mode bits, M1 for SRC1 , M2 for SRC2 and the instruction type, floating-point or non- floating point, determine whether an operand is a register or a literal. For non-floating point instructions, if a mode bit is set to 0, the respective SRC1 or SRC2 field specifies a global or local register. If the mode bit is set to 1, the field specifies an ordinal literal (5 bits) in the range of 0 to 31. For floating-point instructions, if the mode bit is set to 0, the respective SRC1 or SRC2 field specifies a register just as it does for non- floating point instructions. If the mode bit is set to 1 the field specifies either a floating point register or one of the two real number literals (+0.0 or +1.0).

The SRC/DST field can specify either a source operand or a destination operand or

bits	*encoding*
31-24	OPCODE
23-19	SRC1
18-14	SRC2
13	M1
12-2	DISPLACEMENT
1-0	0

Table 2.11: 80960KB COBR-instruction format

both depending on the instruction. The mode bit M3 and the instruction type determine how this field is used. For non-floating point instructions, if M3 is clear the SRC/DST is a global or local register. If M3 is set the SRC/DST operand can be used only as a src operand that is an ordinal literal. For floating-point instructions the SRC/DST field is only used to encode the destination operands. If M3 is clear the destination operand is a global or local register. If M3 is set the destination operand is a floating point register.

COBR format

The COBR format (Table 2.11) is used primarily for control-and- branch-instructions. The opcode field is 8 bits. The SRC1 and SRC2 fields specify source operands for the instruction. The SRC1 field can specify either a global or local register or a literal as determined by mode bit M1. The SRC2 field can only specify a local or global register. The displacement field contains a signed, two's complement number that specifies a word displacement. The processor uses this value to compute the address of a target instruction that the processor goes to as a result of a comparison. The displacement field can range from -2^{10} to $2^{10}-1$. To determine the IP of the target instruction, the processor converts the displacement value to a byte displacement. It then adds the resulting byte displacement to the IP of the next instruction.

CTRL format

The CTRL (Table 2.12) format is used for instructions that branch to a new IP, including the branch-if,"bal" and "call" instructions. The return instruction also uses this format. The opcode field for this format is 8 bits. The instructions that use this format have no operands. The target address for a branch is specified with the DISPLACEMENT field in the same manner as is done with the COBR format instructions. Here, the DISPLACEMENT field specifies a word displacement that can range from -2^{21} to $2^{21}-1$. For the "return" instruction DISPLACEMENT field are ignored.

bits	*encoding*
31-24	OPCODE
23-2	DISPLACEMENT
1-0	0

Table 2.12: 80960 CTRL-instruction format

MEMA		*MEMB*	
bits	*encoding*		
31-24	OPCODE	31-24	OPCODE
23-19	SRC/DST	23-19	SRC/DST
18-14	ABASE	18-14	ABASE
13	MD	13-10	MODE
12	0	9-7	SCALE
11-0	OFFSET	6-5	0
		4-0	INDEX

Table 2.13: 80960 MEMA,MEMB instruction formats

MEM format

The MEM(A) or MEM(B), (table 2.13) ,formats is used for instructions that require a memory address to be computed. These instructions include the load-, store- and "lda" instructions. Also, the extended versions of the branch, branch-and-link, and call instructions uses this format. The MEMB format offers the option of including a 32-bit displacement contained in a second word, to the instruction. Bit 12 of the first word of the instruction determines whether the format is MEMA (clear) or MEMB (set).

1. MEMA format

   For both formats the opcode field is 8 bits long. The SRC/DST field specifies a global or local register. For load-instructions, the SRC/DST field specifies the destination register for a word loaded into the processor from memory or, for operands larger than one word, the first of successive destination registers. For store instructions, this field specifies the register or group of registers that contain the source operand to be stored in memory.

   The mode bit (or for MEMB mode bits) determine the address mode used for the instruction.

   The MEMA format provides two addressing modes: absolute offset and register indirect with offset. The offset field specifies an unsigned byte offset from 0 to 4096. The ABASE field specifies a global or local register that contains an address in memory. The address is interpreted as either a virtual address or a physical address depending on whether the processor is operating in virtual addressing or physical addressing mode respectivly.

   For the absolute offset addressing mode (the MD bit is clear), the processor interprets the offset field as an offset from byte 0 of the current address space. The ABASE field

is ignored. The use of this addressing mode along with the "lda" instruction allows a constant of from 0 to 4096 to be loaded into a register.

For the register indirect with offset addressing mode (the `MD` bit is set), the value in the `OFFSET` field is added to the address in the `ABASE` register. Setting the offset value to zero creates a register indirect addressing mode, however, this operation can generally be carried out faster by using the MEMB version of this addressing mode.

2. MEMB format

   The MEMB format provides seven addressing modes: absolute displacement, register indirect, register indirect with displacement, register indirect with index, register indirect with index and displacement, index with displacement, IP with displacement. The `ABASE` and `INDEX` fields specify local or global registers, the contents of which are used in the address computation. When the `INDEX` field is used in an addressing mode, the processor automatically scales the value in the index register by the amount specified in the `SCALE` field. The optional displacement field is contained in the word following the instruction word. The displacement is a 32 bit signed, two's complement value.

2.2.5 80960KB addressing Modes

The processor offers 11 modes for addressing operands. These modes are grouped as follows: Literal, Register, Absolute, Register Indirect, Register Indirect with displacement, IP with displacement. Most of the instructions use only the literal and register modes. The remaining modes are used for memory related instructions.

Literals

The processor recognizes two types of literals: ordinal literal and floating point literal. An ordinal literal can range from 0 to 31 (5 bits). When an ordinal literal is used as an operand the processor expands it to 32 bits by adding leading zeroes. If the instruction specifies an operand larger than 32 bits, the processor zero-extends the value to the operand size. If an ordinal literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

The processor also recognizes two floating point literals(+0.0 and +1.0). These floating point literals can only be used with floating point instructions. As with the ordinal literals, the processor converts the floating point literals to the operand size specified by the instruction.

A few of the floating point instructions use both floating-point and non floating-point operands, e.g the convert integer to real-instructions. Ordinal can be used in these instructions for non-floating point operands.

Register

A register is referenced as an operand by giving the register number. Both floating point and non floating point instructions can reference global and local registers in this way. However floating point registers can only be referenced in conjunction with floating-point instructions.

Absolute

Absolute addressing is used to reference a memory location directly as an offset from address 0 of the address space ranging from -2^{31} to 2^{31}. At the machine level two absolute addressing modes are provided, depending on the instruction format, i.e MEMA or MEMB. For the MEMB format the offset is an integer called a displacement ranging from -2^{31} to $2^{31} - 1$. After evaluating an absolute address, the assembler will convert the address into an offset and select the appropriate machine-level instruction type and addressing mode.

Register Indirect

The Register Indirect addressing modes allow an address to be specified with an ordinal value (32 bits) in a register or with an offset or displacement added to a value in a register. Here the value in the register is referred to as the address base.

Register Indirect with Index

The register indirect with index addressing modes allow a scaled index to be added to the value in a register. The index is specified by means of a value placed in a register. This index value is then multiplied by the scale factor. The allowable scale factors are 1,2,4,8 and 16. A displacement may also be added to the address base and scaled index.

Index with Displacement

A scaled index can also be used with a displacement alone. Again, the index is contained in a register and multiplied by a scaling constant before the displacement is added to it.

IP with Displacement

The IP with displacement addressing mode is often used with load and store instructions to make them IP relative. With this mode the displacement plus a constant of 8 is added to the IP of the instruction.

2.2.6 80960 KB processor states

The 80960 KB has four different operating states: executing, interrupted, stopped and stopped-interrupted. The processor is placed in one of two states (executing or stopped) at initialization. After that the processor and software controls the processor's state.

The processor can switch between the executing and interrupted states or between the stopped and stopped-interrupted states. However, the processor never switches from the executing state to the stopped state unless it detects a series of fault conditions that it cannot handle.

Interrupts, IACs and Faults

The processor defines two methods of asynchronously requesting services from the processor: interrupts and IAC (InterAgent Communication) messages. Interrupts are the more common of the two.

An interrupt is a break in the control flow of a program so that the processor can handle a more urgent chore. Interrupt requests are generally sent to the processor from an external source, often to request I/O services. When the processor receives an interrupt request, it temporarily stops work on its current task and begins work on an interrupt handling procedure. Upon completion of the interrupt handling procedure, the processor generally returns to the task that was interrupted and continues work where it left off. Interrupts also have a priority, which the processor uses to determine whether to service the interrupt immediatly or to postpone the service until a later time.

The 80960 KB processor provides an alternate method of communicating with other agents in the system called IAC messages, or simply IACs. Using the IAC mechanism, other agents on the system bus are able to communicate with the processor through messages that are exchanged in a reserved section of memory.

Like interrupts, IACs are used to request that the processor stop work on its current task and begin work on another task. However, where an interrupt generally causes an temporary break in the execution of a program, an IAC often causes a permanent change in the control flow of the processor.

While executing instructions, the processor is able to recognize certain conditions that could cause it to return an inappropriate result or that could cause it to go down a wrong and possibly disastrous path. One example of such a condition is a divisor operand of zero in a divide operation. Another example is an instruction with an invalid opcode. These conditions are called faults. The processor handles faults almost the same way that it handles interrupts. When the processor detects a fault, it automatically stops its current processing activity and begins work on a fault-handling procedure.

2.3 AMD Am29000

In 1987, Advanced Micro Devices (AMD) released the first microprocessor ever designed by the company, the Am29000. The processor operates at a 25 MHz clock rate and a 40 ns instruction cycle time. AMD claims that it can hit a peak execution rate at 25 mips and a sustained performance level at 17 mips. Am29000 is an "enhanced RISC design", meaning that key RISC concepts have been combined with conventional design to reach highest possible performance. Among other things it features a four-stage pipeline, 128 bytes instruction branch target cache and an on chip memory management unit.

2.3.1 Am29000 instruction set

The Am29000 instruction set contains 112 instructions divided into 9 classes: integer arithmetic, compare, logical, shift, data movement, constant, floating point, branch and miscellanous instructions. The processor executes all instructions in a single cycle except for interrupt returns, load multiple and store multiple. The complete instruction set is given in Appendix B.

There are two mutually-exclusive modes of program execution; the supervisor mode and the user mode. In the supervisor mode executing programs have access to all processor resources. In the user mode, certain processor resources may not be accessed; any attempted access causes a trap.

2.3.2 Am29000 data formats

A word is defined as 32 bits of data. A half-word consists of 16 bits and a double-word consists of 64 bits. Bytes are 8 bits in length. Within a word, bits are numbered in increasing order from right to left, starting with the number 0 for the least significant bit. Within a word, bytes and half-words are numbered in increasing order from left to right starting with 0 (big endian scheme) or right to left (little endian scheme) as controlled by the processor configuration register.

Most instructions deal directly with word-length integer data; integers may be either signed or unsigned depending on the instruction. Some instruction (e.g AND) treat word length operands as strings of bits. In addition, there is support for character, half-word, and Boolean data types. Floating point data (single and double precision) are defined but not directly supported by processor hardware.

The processor supports character data through extraction (EXBYTE) and insertion (INBYTE) operations on word length operands, and by a compare (CPBYTE) operation on byte length fields within words.

The processor supports half-word data through extraction (EXHW) and insertion (INHW) operations on word-length operands. There is also an Extract Half Word Sign

absolute register number	*general purpose register*
0	Indirect Pointer Access
1	Stack Pointer
2-63	Not Implemented
64-127	Global Registers 64-127
128	Local Register 125
129	Local Register 126
130	Local Register 127
129	Local Register 0
131	Local Register 1
...	...
254	Local Register 123
255	Local Register 124

Table 2.14: Am29000 general purpose registers

Extended instruction (EXHWS) which acts similar to EXHW.

The Boolean format used by the processor is such that the Boolean values TRUE and FALSE are represented by 1 or 0 respectively, in the most significant bit of a word.

The floating point format defined for the processor conforms to the IEEE Floating Point standard P754.

2.3.3 Am29000 register description

The Am29000 has three classes of registers which are accessible by instructions. These are: general-purpose registers, special- purpose registers and translation-look-aside buffer (TLB) registers. Any operation available can be performed on the general-purpose registers, while the special purpose registers and the TLB registers are accessed only by explicit data movement to or from a general purpose register. Table 2.14 lists the 192 general purpose registers and their functions.

The following terminology is used to describe the addressing of general-purpose registers:

1. Register Number is a software level number for a general purpose register (0-255).
2. Global Register Number is a software level number for a global register ranging from 0-127.
3. Local Register Number is a software level number for a local register ranging from 0-127.

4. Absolute Register number is a hardware level number used to select a general purpose register in the Register File. These numbers range from 0-255.

The 192 registers are divided into 64 global and 126 local registers. Global registers are addressed with absolute register numbers while local registers are addressed relative to an internal stackpointer. The general purpose registers may be accessed indirectly, with the register number specified by the content of a special purpose register rather than the instruction field. Three independent indirect register numbers are contained in three separate special-purpose registers. The number for Global Register 0 specifies indirect register-addressing. An instruction can specify an indirect register for any or all of the source operands or result.

General registers may be partitioned into segments of 16 registers for the purpose of access protection. A register in a protected segment may be accessed only by a program executing in the Supervisor mode. An attempted access by a User-mode program causes a trap to occur.

The Am29000 contains 23 special purpose registers which provide controls and data for certain processor functions. Special Purpose registers are accessed by data movement only. Any special purpose register can be written with the contents of any general purpose register and vice versa. Some special purpose registers are protected and can be accessed only in the Supervisor mode. This restriction applies to both read and write accesses. Any User mode program violation of this restriction causes a trap to occur.

The special-purpose registers are partitioned into protected an unprotected registers. Special purpose registers numbered 0-127 and 160-255 are protected and the remaining are unprotected. Not all of these are implemented. The special purpose registers and their definitions are listed in table 2.15.

Vector Base Area Address - Defines the beginning of the interrupt/trap Vector Area.

Old Processor Status - Stores a copy of the current processor status when an interrupt or trap is taken. It is later used to restore the current processor status on an interrupt return.

Current Processor Status - contains control information associated with the currently executing process such as interrupt disables and the supervisor mode bit.

Configuration - contains control information which normally varies only from system to system and is usually set only during system initialisation.

Channel Address - Contains the address associated with an external access and retains the address if the access does not complete successfully. The Channel Address Register in conjunction with the Channel Data and Channel Control registers allow restarting of unsuccessfull external accesses.

Channel Data - Contains Data associated with a store operation and retains data if the operation does not complete successfully.

register number	*protected registers*	*register number*	*unprotected registers*
0	Vector Base Address	128	Indirect Pointer C
1	Old Processor Status	129	Indirect Pointer B
2	Current Processor Status	130	Indirect Pointer A
3	Configuration	131	Q
4	Channel Address	132	ALU Status
5	Channel Data	133	Byte Pointer
6	Channel Control	134	Funnel Shift Count
7	Register Bank Protect	135	Load/Store Count Remaining
8	Timer Counter		
9	Timer Reload		
10	Program Counter 0		
11	Program Counter 1		
12	Program Counter 2		
13	MMU Configuration		
14	LRU Recommendation		

Table 2.15: Am29000 special purpose registers

Channel Control - Contains information associated with a channel operation and retains this information if the operation does not complete successfully.

Register Bank Protect - Restricts access of User Mode programs to specified groups of registers. This facilitates register banking for multi-tasking applications and protects operating system parameters kept in the global registers from corruption by User mode programs.

Timer Counter- supports real-time control and other timing related functions.

Timer Reload- maintains synchronisation of the Timer Control. It includes control bits for the Timer facility.

Program Counter 0 - Contains the address of the instruction being decoded when an interrupt or trap is taken. The processor restarts this instruction upon interrupt return.

Program Counter 1 - Contains the address of the instruction being executed when an interrupt or trap is taken. The processor restarts this instruction upon interrupt return.

Program Counter 2 - Contains the address of the instruction just completed when an interrupt or trap is taken. This address is provided for information only and does not participate in an interrupt return.

MMU Configuration - Allows selection of various memory management options.

LRU Recommendation - Simplifies the reload of entries in the translation look-aside buffer by providing information on the least recently used entry of the TLB when a TLB miss occurs.

bits	*encoding*
31-22	OP
22	A/M
21-16	RC
	I17..I10
	I15..I8
	VN
	CE/CNTL
15-8	RA
	SA
7-0	RB
	RB or I
	I9..I2
	I7..I0
	UI/RND/FD/FS

Table 2.16: Am29000 instruction formats

The unprotected special-purpose registers are defined as follows:

Indirect Pointer C - Allows the indirect access of a general purpose register.

Indirect Pointer B - Allows the indirect access of a general purpose register.

Indirect Pointer A - Allows the indirect access of a general purpose register.

Q - Provides additional operand bits for multiply and divide operations.

ALU Status - Contains information about the outcome of arithmetic and logical operations and holds residual control for certain instruction operations.

Byte Pointer - Contains an index of a byte or half-word within a word. This register is also accessible via the ALU status register.

Funnel Shift Count - Provides a bit offset for the extraction of word-length fields from double word operands. This register is also accessible via the ALU status register.

Load/Store Count Remaining - Maintains a count of the number of loads and stores remaining for load-multiple and store-multiple operations. The count is initialised to the total number of loads or stores to be performed before the operation is initiated. This register is also accessible via the Channel Control Register.

2.3.4 Am29000 instruction format

All instructions for the Am29000 are 32 bits in length, and are divided into four fields. These fields have several alternative definitions. In certain instructions, one or more fields are not used, and are reserved for future use.

The instruction format is shown in table 2.16 and the various fields are interpreted as follows:

- OP, this field contains an operation code definig the operation to be performed. In some instructions the least significant bit selects between two possible operands. For this reason this bit is sometimes labelled A or M with the following interpretations:

  Absolute, the A-bit is to differentiate between program- counter relative (A=0) and absolute (A=1) instruction addresses when these addresses appear within instructions.

  IMmediate, the M-bit selects between a register operand (M=0) and an immediate operand (M=1) when the alternative is allowed by the instruction

- RC, the RC field contains a global or local register number

- I17..I10, this field contains the most significant 8 bits of a 16- bit instruction address. This is a word address and may be program counter relative or absolute, depending on the A bit of the operation code.

- I15..I8, this field contains the most significant 8 bits of a 16- bit instruction.

- VN, this field contains an 8-bit trap vector number

- CE/CNTL, this field controls a load or store access

- RA, the RA-field contains a global or local register number

- SA, the SA-field contains a special register number

- RB, the RB-field contains a global or local register number

- RB or I, this field contains either a global or local register number, or an 8-bit instruction constant depending on the value of the M-bit of the operation code.

- I9..I2, this field contains the least significant 8 bits of a 16- bit instruction address. This is a word address, and may be program counter relative or absolute, depending on the A-bit of the operation code.

- I7..I0, this field contains the least significant 8 bits of a 16 bits instruction constant

- UI/RND/FD/FS, this field controls the operation of the CONVERT instruction.

2.3.5 Am29000 processor states

Normal program flow may be preempted by an interrupt or trap for which the processor is enabled. The effect on the processor is identical for interrupts and traps; the distinction is in the different mechanisms by which the interrupt and traps are enabled. The intension is that interrupts be used for suspending current program execution and causing another program to execute, while traps be used to report errors and exception conditions.

An interrupt or trap is said to occur when all conditions which define the interrupt or trap are met. An interrupt or trap which occurs is not necessarily recognized by the processor, either because of various enables or because of the processor's operational mode. An interrupt is taken when the processor recognizes the interrupt and alters its behaviour accordingly.

Interrupts are caused by signals applied to any of the external inputs INTR0 - INTR3 or by a timer facility. The processor may be disabled from taking certain interrupts by the masking capability provided by the "Disable all interrupts and traps" (DA), "Disable Interrupts" (DI) bit and "Interrupt Mask"(IM) field in the current processor status register. The INTR0 cannot be disabled by the IM-field, thus its a non-maskable interrupt line.

Traps are caused by signals applied to one of the inputs TRAP0-TRAP1 or by exceptional conditions such as protection violation.

Interrupt and trap processing relies on the existence of a user managed vector area in external instruction/data memory or instruction read only memory (instruction ROM). The Vector Area begins at an address specified by the Vector Area base Address Register, and provides for 256 different exception handling routines. The processor reserves 32 routines for system operation and 32 routines for FP multiply and divide instructions.

When an exception is taken, the processor determines an 8-bit vector number associated with the exception. Vector numbers are either predefined or specified by an instruction causing the trap as shown in table 2.17.

2.3.6 Am29000 pipelining

The Am29000 implements a four-stage pipeline for instruction execution. The four stages are: fetch, decode, execute and write back. During the fetch stage, the Instruction Fetch Unit *IFU* determines the location of the next processor instruction to the decode stage. The instruction is fetched either from the Instruction Prefetch Buffer, the Branch Target Cache, or an external instruction memory. During the decode stage the Execution Unit *EU* decodes the instruction selected during the fetch stage and fetches and/or assembles the required operands. It also evaluates addresses for branches, loads and stores. During the execute stage, the Execution Unit *EU* performs the operation specified by the instruction. In the case of branches, loads, and stores the Memory Management Unit *MMU* performs address translation if required. During the write-back stage, the results of the operation performed during the execution stage are stored. In the case of branches, loads and stores the physical address resulting from translation during the execute stage is transmitted to an external device or memory.

Most pipeline dependencies which are internal to the processor are handled by forwarding logic in the processor. For these dependencies which result from the external system, the Pipeline Hold mode insures proper operation. In a few special cases the processor pipeline is exposed to software executing on the Am29000.

vector	*exception*
0	Illegal Opcode
2	Unaligned Address
3	Out of Range
4	Coprocessor Not Present
5	Coprocessor Exception
6	Instruction Access Violation
7	Data Access Violation
8	User Mode Instruction TLB-miss
9	User Mode Data TLB-miss
10	Supervisor Mode Instruction TLB-miss
11	Supervisor Mode Data TLB-miss
12	Instruction TLB protection violation
13	Data TLB protection violation
14	Timer
15	Trace
16	INTR0
17	INTR1
18	INTR2
19	INTR3
20	TRAP0
21	TRAP1
22-63	Reserved or associated with FP-instructions
64-255	User defined

Table 2.17: Am29000 exception vectors

2.4 MIPS R2000 processor

The R2000 is based on research work carried out at Stanford in the beginning of the eighties. Especially a base level instruction set was proposed from the experience gained during work with optimizing compilers. The R2000 processor consists of two tightly coupled processors implemented on a single chip. The first processor is a full 32-bit RISC CPU. The second processor is a system control coprocessor (CP0), containing a TLB (Translation Lookaside Buffer) and control registers to support a virtual memory subsystem and separate caches for instruction and data. A predecessor, R3000, adds a floating-point processor to R2000. Thus, what is said in this chapter also applies to the R3000 microprocessor.

2.4.1 R2000 instruction set

The R2000 instruction set contains 74 instructions divided into 6 groups: load/store, computational, jump and branch, coprocessor, coprocessor 0,and special instructions. A summary is given in Appendix B.

The R2000 has two operating modes: user mode and kernel mode. The R2000 normally operates in the user mode until en exception is detected forcing it into kernel mode. It remains in kernel mode until an *Restore From Exception* instruction is executed.

2.4.2 R2000 data formats

The R2000 defines a 32-bit word, a 16-bit halfword and an 8-bit byte. The byte ordering is configurable (configuration occurs during hardware reset) into either big-endian or little-endian byte ordering. Bit 0 is always the least significant (rightmost) bit. Thus bit-designations are always little-endian. The R2000 uses byte-addressing with alignment constraints, for half word and word accesses; half word accesses must be aligned on an even byte boundary and word accesses must be aligned on a byte boundary divisible by four. Special instructions are provided for addressing words that are not aligned on 4-byte (word) boundaries (Load/Store-Word- Left/Right; LWL,LWR,SWL,SWR). These instructions are used in pairs to provide addressing of misaligned words with one additional instruction cycle over that required for aligned words.

2.4.3 R2000 register description

The register set consists of general-purpose registers as well as dedicated registers.

- The R2000 provides 32 general purpose 32-bit registers. r0 .. r31 each consists of a single word. The registers are treated symmetrically with two exeptions. Register r0 is hardwired to a zero value and r31 is the link register for jump and link instructions.

I-type		J-type		R-type	
bits	*encoding*	*bits*	*encoding*	*bits*	*encoding*
31-26	OP	31-26	OP	31-26	OP
25-21	RS	25-0	TARGET	25-21	RS
20-16	RT			20-16	RT
15-0	IMMEDIATE			15-11	RD
				10-6	SHAMT
				5-0	FUNC

Table 2.18: R2000, instruction formats

- The two multiply/divide registers (HI,LO) store the double-word, 64-bits result of multiply operations and the quotient and remainder of divide operations.
- A 32-bit program counter.
- Exception Handling Registers:
  - the *Cause* register describe the last exception.
  - the *EPC* (Exception Program Counter) contains the address where processing can resume after an exception has been serviced.
  - the *Status* register contains all major status bits.
  - the *BadVAddr*(Bad Virtual Address) register saves the entire bad virtual address for any addressing exception.
  - the *Context* register provides information useful for a software TLB exception handler.
  - the *PRId*(Processor Revision Identifier) register contains information that identifies the implementation revision level of the Processor and System Control Coprocessor.

2.4.4 R2000 instruction format

Every R2000 instruction consists of a single word (32 bits) aligned on a word boundary. There are three instruction formats described in table 2.18,

The interpretation of the fields are as follows:

- `OP` is a 6-bit operation code
- `RS` is a 5-bit source register specifier
- `RT` is a 5-bit target register (source/destination) or branch condition
- `IMMEDIATE` is a 16-bit immediate branch displacement or address displacement
- `TARGET` is a 26-bit jump target address

- `RD` is a 5-bit shift amount
- `FUNCT` is a 6-bit function field

2.4.5 R2000 processor states

The normal instruction execution may be preempted by an exception. When the R2000 detects an exception, the normal sequence of instruction execution is suspended; the processor is forced into Kernel mode where it can respond to the abnormal or asynchronous event. When an exception occurs, the R2000 loads the EPC (Exception Program Counter) with an appropriate restart location where execution may resume after the exception has been serviced. The restart location in the EPC is the address of the instruction which caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediatly preceeding the delay slot. The R2000 aborts the current instruction, which may be an instruction causing the exception, and also aborts all those following in the instruction pipeline which have already began execution. The R2000 then performs a direct jump into a designated exception handler routine.

The following exceptions are recognised by the R2000:

- *Reset* Assertion of the R2000's reset signal causes an exception that transfers control to the special vector at address 0xBFC00000
- *UTLB miss* User TLB miss. A reference is made to a page that has no matching TLB entry.
- *TLB miss* A referenced TLB entry's valid bit is not set or there is a reference to a page that has no matching TLB entry.
- *TLB modified* During a store operation, the valid bit is set but the Dirty bit is not set.
- *Bus Error* Assertion of the R2000's BERR* signal due to such external events as bus timeout, backplane bus parity errors, invalid physical address or invalid access type.
- *Address Error* Attempt to load, fetch or store an unaligned word; that is, a word or halfword at an address not evenly divisible by 4 or 2 respectively. Also caused by reference to a virtual address with most significant bit set while in user mode.
- *Overflow* Two's complement overflow during add or subtract.
- *System Call* Execution of the *syscall* instruction.
- *Breakpoint* Execution of the *break* instruction.
- *Reserved Instruction* Execution of an instruction with an undefined or reserved major operation code, or a *special* instruction whose minor opcode is undefined.

- *Coprocessor Unusable* Execution of a coprocessor instruction when the CU (Coprocessor Usable) bit is not set for the target processor.
- *Interrupt* Assertion of one of the R2000's six hardware interrupt inputs or setting of one of the two software interrupt bits in the Cause Register.

2.4.6 R2000 pipeline

The execution of a single instruction consists of five pipeline stages:

1. IF Instruction Fetch. Access the TLB and calculate the instruction address required to read an instruction from the I-cache. The instruction is not actually read into the processor until the beginning of the RD pipe-stage.
2. RD Read any required operands from CPU-registers while decoding the instruction.
3. ALU Perform the required operation on instruction operands.
4. MEM Access memory (D-Cache) if required(for Load/Store instructions)
5. WB Write back ALU results or value loaded from D- cache to register file.

Each of these steps require approximatly one CPU- cycle.

The R2000 uses different technique internally to enable execution of all instructions in a single cycle. However, as discussed earlier, there are load and store instruction as well as jump and branch which could disturb the smooth flow of instructions through the pipeline. In R2000, the execution continues, despite the delay. Loads ,jumps and branches do not interrupt the normal flow of instructions through the pipeline. The processor always executes the instruction immediately following one of these "delayed" instructions. Instead of having the processor deal with pipeline delays, the R2000 turns over the responsibility for dealing with delayed instructions to software.

2.5 Cypress SPARC CY7C600

The SPARC (Scalable Processor ARChitecture), designed by Sun Microsystems is an open computer architecture. SPARC is an architecturally driven standard, with binary compatibility of software between processor versions ensured by enforcing compliance to the architecture standard. CY7C600 chip set is a 32-bit custom CMOS implementation of the SPARC architecture, currently available in clock speed of 40 MHz. The chip set includes integer unit, floating point unit, cache/memory management controllers and cache RAMs. In this chapter the integer unit as well as the floating point unit will be referred to with the name SPARC.

2.5.1 SPARC instruction set

SPARC defines 55 basic integer instructions, 14 basic floating point instructions and two coprocessor-operate instruction formats. The instructions fall into five basic categories: load/store, arithmetic/logical/shift, control transfer, read/write control register, and floating point-operate/coprocessor-operate.

Load and store instructions are the only way to access memory or external registers. Addresses are calculated using the contents of two registers or one register and a constant. The destination may be either an integer unit, floating point unit or coprocessor register, which either supplies or receives the data.

SPARC employs a supervisor/user mode model of operation. The state determines which address space is accessed with the ASI bits (see below) and whether or not privileged instructions may be used. Privileged instructions restrict control register access to supervisor software, preventing user programs from accidentally altering the state of the machine.

Whenever an address is sent to the address bus, the processor also generates 8 bits of address space identifier (ASI). The ASI pins identify for the external system which of the 256 possible address spaces is to be accessed. The address space identifier is intended for use by the operating system software, and the instructions that specify a particular ASI value are privileged and can only be executed in supervisor mode.

Arithmetical/logical/shift instructions compute a result using two source operands and place the result in a destination register. In addition to standard arithmetic this processor includes tagged arithmetic operations to support languages such as LISP and Prolog. Control transfer instructions include jumps, calls, branches and traps. A summary of the complete instruction set is given in Appendix B.

Register numbers	Name
r[24] to r[31]	*ins*
r[16] to r[23]	*locals*
r[8] to r[15]	*outs*
r[0] to r[7]	*globals*

Table 2.19: SPARC Register Addressing

2.5.2 SPARC data formats

SPARC supports nine data types. Integer data types includes *byte, unsigned byte, halfword, unsigned halfword, word* and *unsigned word.* The IEEE floating point types include *single, double* and *extended.* A *byte* is 8 bit wide, a *halfword* is 16 bits, a *word* is 32 bits, a *single* is 32 bits, a *double* is 64 bits and an *extended* is 128 bits.

2.5.3 SPARC registers

The integer unit has two types of registers associated with it: working registers *r registers* and control/status registers. Working registers are used for normal operations, and control/status registers keep track of control and the state of the IU. The FPU has 32 working registers (called *f registers*), and two control/status registers: the Floating-point State Register (FSR), and the Floating-point Queue (FQ). All *r registers* are 32 bits wide. They are divided into 8 global registers and 7 blocks called windows. Each window contain 24 *r registers.* The windows are addressed by the CWP, a field of the Processor State register (PSR). The CWP is incremented by a RESTORE or RETT instruction and decremented by a SAVE instruction. The *active* window is defined as the window currently pointed to by the CWP. The Window Invalid Mask (WIM) is a register which, under software control, detects the occurence of IU register file overflows and underflows.

The registers in each window are divided into *ins* ,*outs* and *locals.* Registers are addressed as shown in table 2.19. The *globals* may be addressed when any window is active.

Each window shares its *ins* and *outs* with adjacent windows. The register overlap in such a way that, given a register with address o where $7 < o < 16$, o refers to exactly the same register as $(o + 16)$ after the CWP is decremented by 1 modulo 7 (points to the next window). The windows are joined together in a circular stack, where the highest numbered window is adjacent to the lowest. The *outs* of window 6 are the *ins* of window 0.

The global register r[0] is hardwired to zero. Thus reading this register yields a zero result while writing to it has no effect.

The *out* register r[15] is used for storing the return address when a CALL instruction is executed.

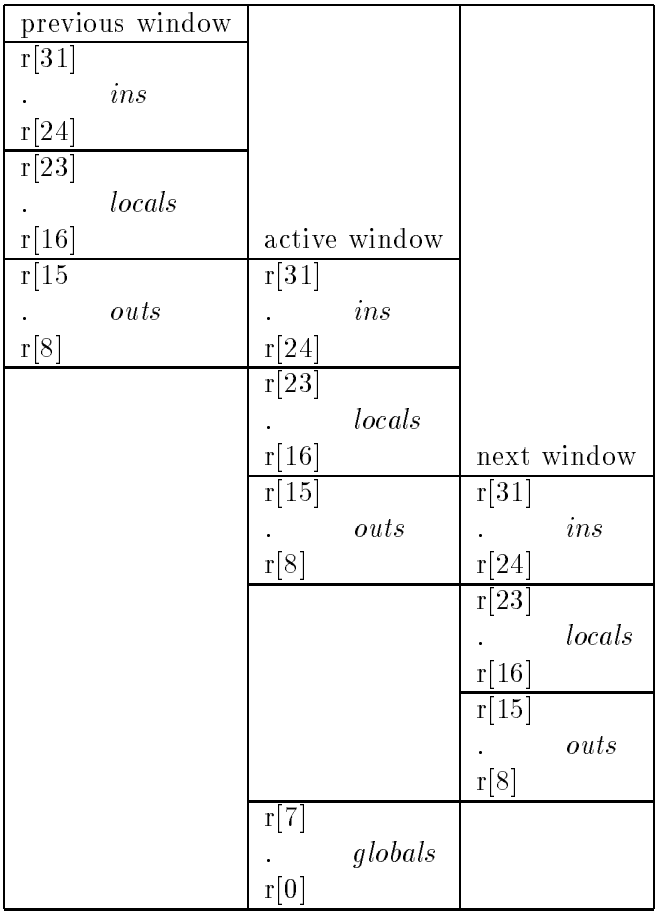

Figure 2.1: Three overlapping windows and globals

Because the processor logically provides new *locals* and *outs* after every procedure call, register local values need not be saved and restored across calls. Figure 2.1 shows how parameters may be passed to and from subroutines.

The IU's control/status registers are all 32-bit read/write registers unless specified otherwise. They include the program counters (PC and nPC) the Processor State Register (PSR), the Window Invalid Mask Register (WIM), the Trap Base Register (TBR), and the Multiply step (Y) register. The PC contains the address of the instruction currently being executed and nPC hold the address of the next instruction to be executed assuming no trap occurs.

The 32-bit PSR contains various fields describing the state of the IU. Among these are: `ICC` which contains the IU's condition codes. These bits are modified by dedicated instructions and by the WRPSR (write processor status register) instruction. The `EC` bit determines whether or not the coprocessor is enabled. The `EF` bit determines whether or not the FPU is enabled. Processor interrupt level is reflected by the contents in `PIL` field. The processor only accepts interrupts whose interrupt level is greater than the value in

PIL. The S bit determines whether the processor is in supervisor mode or not. Supervisor mode can only be entered by a software or hardware trap. The PS bit contains the value of the S bit at the time of the most recent trap. ET is the Trap Enable bit. When it is set, traps are enabled. When ET is disabled, all asynchronous traps are ignored. A synchronous trap will cause the processor to halt and enter "error mode", i.e perform a RESET. CWP comprise the Current Window Pointer, which points to the current active *r* register window. It is decremented by traps and the SAVE instruction, and incremented by RESTORE and RETT instructions.

The Window Invalid Mask Register (WIM) is used to determine whether a window overflow or window underflow trap should be generated by a SAVE,RESTORE or RETT instruction. Each bit in the WIM corresponds to a window. The register may be written by WRWIM and read by RDWIM instructions. Bits corresponding to nonexistent windows read as zeroes and values written are ignored.

The Trap Base register (TBR) contains three fields that generate the address of the trap handler when a trap occur. The Trap Base Address TBA, which is controlled by software. It contains the most significant 20 bits of the trap table address. The TBA field can be written by the WRTBR instruction. The trap type (tt) field is an 8-bit field that is written by the processor at the time of a trap, and retains its value until the next trap. It provides an offset into the trap table. The WRTBR instruction does not affect the tt field.

In addition to this there is a Floating Point State Register (FPR) that contain FPU mode and status information.

2.5.4 SPARC instruction formats/addressing modes

The SPARC instructions are classified into three major formats, simply called *format1, format 2* and *format 3*. These are summarised in tables 2.20 and 2.21. Two formats include subformats.

The OP field selects formats(format1,format2 or format3).

1. The *format 1* is used by the CALL instruction and contains a 30-bit sign-extended

format 1		*format 2*			
		SETHI		BRANCH	
bits	*encoding*	*bits*	*encoding*	*bits*	*encoding*
31-30	OP	31-30	OP	31-30	OP
29-0	DISP30	29-25	RD	29	A
		24-22	OP2	28-25	TCOND
		21-0	IMM22	24-22	OP2
				21-0	DISP22

Table 2.20: SPARC *format 1* and *format 2* instruction formats

other integer instructions				FP/COPROC operations	
bits	*encoding*	*bits*	*encoding*	*bits*	*encoding*
31-30	OP	31-30	OP	31-30	OP
29-25	RD	29-25	RD	29-25	RD
24-19	OP3	24-19	OP3	24-19	OP3
18-14	RS1	18-14	RS1	18-14	RS1
13	0	13	1	13-5	OPF/OPC
12-5	ASI	12-0	SIMM13	21-0	RS2
4-0	RS2				

Table 2.21: SPARC *format 3* instruction formats

word displacement, DISP30.

2. The *format 2* is used by SETHI and branch-instructions:

   - OP2 contains instruction opcode for *format 2*.
   - RD, For store instructions, this register selects an *r register* (or an *r register pair*), or an *f register* (or an *f register pair*) to be the source. For all other instructions, this field selects an *r register* (or an *r register pair*), or an *f register* (or an *f register pair*) to be the destination.
   - The A bit means "annul" in format 2 instructions. This bit changes the behaviour of the instruction encountered immediatly after a control transfer.
   - TCOND, This field selects the condition code for format 2 instructions.
   - The IMM22 field contains 22-bit constant value used by the SETHI instruction.
   - DISP22, This field contains a 22-bit sign-extended value used for PC-relative addressing when a branch is taken.

3. Remaining instruction uses *format 3*:

   - The OP3 op3 field selects one of the format 3 opcodes.
   - ASI, This 8-bit field is the *address space identifier* generated by load/store alternate instructions.
   - RS1, This 5-bit field selects the first source operand from either the *r* registers for integer instructions, a *f* register for floating point instructions or a *c* register for coprocessor instructions.
   - RS2, This 5-bit field selects the second source operand from either the *r* registers for integer instructions, a *f* register for floating point instructions or a *c* register for coprocessor instructions.
   - *SIMM13*, This field is a sign-extended 13-bit immediate value used as the second ALU operand. It is sign-extended to full word size when used.
   - *OPF/OPC*, This 9-bit field identifies a floating point operate(FPop) instruction or a coprocessor operate (CPop) instruction.

2.5.5 SPARC traps and exceptions

SPARC supports three types of traps: *synchronous, floating-point/coprocessor* and *asynchronous*. Asynchronous traps are also called *interrupts*. Synchronous traps are caused by an instruction and occur before the instruction is completed. Floating-point/coprocessor traps are caused by floating-point/coprocessor instructions and occur before the instruction is completed. Asynchronous traps occur when an external event interrupts the processor. They are not related to any particular instruction and occur between the execution of instructions.

An instruction is defined to be *trapped* if any trap occurs during the course of its execution. If multiple traps occur during one instruction, the highest priority trap is taken. Lower priority traps are ignored because the traps are arranged under the assumption that the lower priority traps persist ,recur or are meaningless due to the presence of the higher priority trap. The ET-bit in the PSR must be set for traps to occur normally. If a synchronous trap occur while traps are disabled the processor halts and enters an error state.

The Trap Base Register (TBR) generates the exact address of a trap handling routine. When a trap occurs, the hardware writes a value into the trap type (*tt*)field of the TBR. This uniquely identifies the trap and serves as an offset into the table whose starting address is given by the TBA field of the TBR. The 8-bit wide *tt* field allows for 256 distinct types of traps as defined in table 2.22.

Trap	Priority	*tt*
reset	1	-
instruction access exception	2	1
illegal instruction	3	2
privileged instruction	4	3
fp disabled	5	4
cp disabled	5	36
window overflow	6	5
window underflow	7	6
mem address not aligned	8	7
fp exception	9	8
cp exception	9	40
data access exception	10	9
tag overflow	11	10
trap instruction	12	128-255
interrupt level 15	13	31
interrupt level 14	14	30
interrupt level 13	15	29
interrupt level 12	16	28
interrupt level 11	17	27
interrupt level 10	18	26
interrupt level 9	19	25
interrupt level 8	20	24
interrupt level 7	21	23
interrupt level 6	22	22
interrupt level 5	23	21
interrupt level 4	24	20
interrupt level 3	25	19
interrupt level 2	26	18
interrupt level 1	27	17

Table 2.22: SPARC trap vector table

2.6 INMOS T800 transputer

Transputer is a family of 16-bit and 32-bit processors. It is a RISC designed for multiprocessor applications. The architecture allow multiprocessor network of arbitrary size and topology to be built. A word-length independent architecture allows the same software to run on any Transputer. Inmos has developed "OCCAM", a language that provides a model for concurrency and communication for all Transputers.

The Transputer has a stack oriented instruction set. Most of the instruction operates on top of an evaluation stack. It has extensive hardware support for concurrency and special communication links supporting large multiprocessor systems.

The IMS T800 is a 32-bit microcomputer with a 64-bit floating point unit and graphics support. It has 4 KBytes on-chip RAM, a configurable memory interface and four standard INMOS communication links.

2.6.1 T800 data formats

The OCCAM model provides 7 different data formats:

1. *BOOL* is a true or false value.
2. *BYTE* is an unsigned 8-bit number.
3. *INT16* is a signed 16-bit number.
4. *INT32* is a signed 32-bit number.
5. *REAL32* conforms to the IEEE-754 single precision standard.
6. *REAL64* conforms to the IEEE-754 double precision standard.

2.6.2 T800 instruction set

The T800 provides a vast instruction set with groups of instructions not found among conventional RISCs. Besides loads/stores, integer arithmetic/logical, floating point arithmetics control transfer and control operation instructions there are block moves, cyclic redundancy check, timer handling ,scheduling instructions to mention a few. There are also facilities for real-time-system software debugging. An instruction set summary is given in Appendix B.

2.6.3 T800 instruction formats and addressing modes

All instructions have the same format designed to give a compact representation. Each instruction consists of a single byte divided into two 4-bits parts. The four most significant

bits of the byte are the function code and the four least significant bits are a data value. This representation provides for sixteen functions, each with a data value ranging from 0-15. Ten of these are used to encode the most important functions. Two more function codes allow the instruction to be extended in length; *prefix* and *negative prefix*. All instructions are executed by loading the four data bits into the least significant four bits of the operand register, which is then used as the instructions operand. All instructions except the prefix instructions end by clearing the operand register, ready for the next instruction.

The *prefix* instruction loads its four data bits into the operand register and then shifts the operand register left four bits. The *negative prefix* instruction is similar, except that it complements the operand register before the shifts. Consequently, operands can be extended to any length up to the length of the operand register by a sequence of prefix instructions. In particular, operands in the range -256 to 255 can be represented using one prefix.

2.6.4 The T800 registers

Expressions are evaluated on the evaluation stack formed by three registers. No hardware mechanism is provided to detect that more than three values are loaded onto the stack. The entire user accessible register set consists of:

- The *Workspace Pointer* which points to an area for local variables.
- The *Instruction Pointer* which points to the next instruction to be executed.
- The *Operand Register* which is used in the formation of instruction operands.
- Three registers A, B and C which form an Evaluation stack. The Evaluation stack is used for expression evaluation, to hold the operands of scheduling and communication instructions, and to hold parameters of procedure calls.

2.7 Saab-Ericsson Space THOR

THOR is a microprocessor primarily intended for embedded real time systems. Among other things it facilitates Ada-(programming language) hardware support, i.e dedicated registers and instructions for implementation of Ada *Task Switches , Rendezvous, Interrupts, Exceptions* and *Real-Time Clock*. Similar to the Inmos T800, THOR performs operations on an *Evaluation Stack*. In addition to this, data can be accessed *Relative to the top of stack*. This makes THOR an interesting synthesis of a traditional stack-computer architecture, and a Reduced Instruction Set Computer. The microprocessor has built-in test support that allows test and debug of hardware/software. Like the T800, multiprocessor configurations are encouraged by the processor architecture.

2.7.1 THOR instruction set

The instruction set is made up from 76 different instructions. Some of these are protected when the processor is running in user mode. There is an unusual group of instructions supporting the ADA "task" concept added as extensive support for the ADA programming language. A summary of all instructions is given in Appendix B.

Instructions may be executed either in privileged mode or user mode. When in privileged mode all instructions can be executed, and no memory protection checks are made, apart from ensuring that addresses are within the 2 GByte address space. In user mode all accesses to each task's stack are protected from access by any other task using memory protect registers (see below). When in user mode some instructions are privileged, an an exception will occur on an attempt to execute them.

2.7.2 THOR data types

Different instruction operates on one (or more) of the following data types: 32-bit integer (unsigned/signed), 32-bit IEE-754 single precision floating point.

2.7.3 THOR instruction formats and addressing modes

There are five different instruction formats (Table 2.23). The format determines the instruction length (in bytes) and how to interpret the parameter (if present).

A 16-bit encoded instruction designated "2". The format designated "2a" is still encoded in 16-bits but includes a parameter "P" which is interpreted as a twos complement value -127 - 128. The format "2b" is identical with "2a" except from the interpretation of the parameter "P". In this format it is interpreted as a binary value 0-255. The format "4a" is encoded in 32 bits and contains a parameter which is interpreted as a twos complement number $-2^{23} \text{to} 2^{23} - 1$. The format "4b" is identical with "4a" except from

bits	*2*	*2a/b*	*4a/b*
16-8	opcode	opcode	-
7-0	ext. opcode	parameter	-
31-24	-	-	opcode
23-0	-	-	parameter

Table 2.23: THOR instruction formats

the interpretation of the parameter "P". In this format it is interpreted as a binary value 0 to $2^{24} - 1$. All instructions with operands use the stack top as implicit source and/or destination operand effective address. There are five different addressing modes: Stack relative, program counter relative, indirect, immediate and register.

Stack Relative addressing mode

The Operand Effective Address is calculated relative to the top of stack (TOS), either implicit or by adding the parameter to TOS.

Program Counter Relative addressing mode

The Operand Effective Address is calculated relative to PC by adding the parameter and PC (shifted right one bit to get word boundary alignment).

Indirect (X) addressing mode

The Operand Effective Address is calculated by adding the parameter and the value on the stack top appearing two instructions previously.

PC Indirect addressing mode

The Operand Effective Address is calculated by adding PC (shifted right one bit) and the value on the stack top appearing two instructions previously.

TOS Indirect addressing mode

The Operand Effective Address is calculated by adding TOS and the value on the stack top appearing two instructions previously.

Mnemonic	Name	Size(bits)
CR	Configuration Register	32
EAR	Error Address Register	31
SIR	Signal Input Register	8
SOR	Signal Output Register	4
RTL	Real Time Clock (MSL)	32
RTM	Real Time Clock (MSH)	32
TP	Task Pointer	3
IR	Identification Register	32

Table 2.24: THOR registers

Immediate (I)

The Operand Effective Address is the TOS, and the source operand is part of the instruction.

Register (R)

The parameter designates the register to be used either as source or as destination operand.

2.7.4 THOR registers

The processor maintains on-chip registers as described in table 2.24.

The **Configuration Register** is used for hardware specific parameters and includes the following fields:

- **CLK** Clock Frequency is used to set a division factor (1 to 255) of the chip clock to get the real time clock and delay register frequency, nominally 1 MHz. Clocks are stopped when this field is zero.
- **CC** *Cache Control* controls the use of data and instruction cache.
- **RM** Controls the IEE-754 floating point *Rounding Mode*.
- **S** Determines the *Scheduling Mode* used.
- **F** Enables *flow control*.
- **B** Enables *bus timeout* exception.
- **WS** *Waitstate* , sets the number of waitstates in the first 1 GByte of memory. From 0 up to 6 waitstates can be used. Setting this field to 7 indicates use of the *Ready* signal.

- **DC** *Data Check* sets the data error checking mode in the first 1 GByte of memory. Mode may be one of: *Odd/Even Parity, EDAC* or *disabled*.

The **Error Address Register** (EAR) is set to the first external memory address which caused an error. The register contains a word address.

The **Identification Register** (IR) is a read-only register holding the chip manufacturer identity, part number and version number.

The **Real-Time-Clock** (RTL,RTM) is a 64 bits value read as two 32-bit registers. Incrementation of this register is due to contents in the Configuration Register.

The **Signal Registers** are used to hold the status of the chip signals used for multiprocessing and interrupts. There is one input register (SIR) and one output register (SOR). Each bit in the registers corresponds to a signal on the chip. There are 6 inputs and 4 outputs.

The **Task Pointer** (TP) points to the task information block in memory.

The **Delay Register** (DR) is the delay counter. It holds the delay of the task. This is a two's complement integer. Normally the register is decremented every microsecond. When decremented below zero (and this task's Status Register DLY flag is set) scheduling is performed.

The **Task Register** (TR) holds task status information for each of the on-chip tasks. TR holds the following information:

- **Ready Flag** (RF) is set when the task is ready to execute.
- **Delay Flag** (DF) is set when the task is delayed.
- **Accept Wait Flag** (AW) is set when this task is waiting for an accept statement.
- **Entry Call Flag** (EF) is set when this task is performing an entry call.
- **Remote Task Flag** (RT) is set when this task is doing a rendevouz with a remote task.
- **Queued Entry Flag** (QE) is set when queued calls exist for an entry called by this task.
- **Rendevouz Field** (RZ) is set to the calling task number when a rendevouz with this task starts, or defines the entry number when this task performs an entry call.
- **Priority Field** (PR) reflects the tasks priority.
- **Accept Field** (AR), when an entry call is pending the bit corresponding to the calling task is set.

Mnemonic	*Name*	*Size(bits)*
RR	Result Register	32
ER	Exception Register	31
SR	Status Register	32
TOS	Top of Stack	29
TOP	Top Register	32
PC	Program Counter	31
EOS	End of Stack	29
BOS	Beginning of Stack	29

Table 2.25: THOR Task Control Registers

For each task there is a **Task Control Block** (TCB) on the processor chip. The TCB's have identical sets of registers as described in table 2.25.

The **Result Register** (RR) holds the least significant half of arithmetic instructions that yuilds 64-bit results.

The **Exception Register** (ER) points to the exception information block in the stack. ER is a word pointer.

The **Status Register** (SR) holds condition codes, hardware exception numbers and Ada support information as follows:

- The **Negative Flag** (N), **Zero Flag** (Z) **Carry Flag** (C) and **Unsigned Flag** (U) is set according to arithmetic conditions.
- The **Task Switch Inhibited Flag** (TSI) is set when no task switch should occur for this task.
- The **User Mode Flag** (UM) is set when this task is in user mode.

The **TOS** register points at the word on top of stack.

The **TOP** register holds the word at the stack top (pointed at by TOS). The 32 words next to top of the runtime stack are cached on the processor chip.

The **Program Counter** (PC) holds the address of the last instruction read from memory. This address is a halfword address.

BOS and **EOS** defines the region in memory where this task's data stack is located. The memory protection check is active in user mode. If an access using the stack addressing mode is not within BOS and EOS, or if TOS would move outside BOS or EOS an exception is raised.

2.7.5 THOR processing states

Normal executing may be preempted by an interrupt condition, by an internal generated exception or by exceptions raised by software

THOR interrupt handling

THOR:s six input pins (reflected in SIR) is regarded as different priority interrupt pins. Anyone turning to an active state forces an interrupt condition. Upon receiving an interrupt, THOR activates a hardware scheduler, the interrupt priority which also may be regarded as a task number, causes the scheduler to dispatch the corresponding task. This mechanism may be used to synchronise tasks running under different microprocessors in a multiprocessor environment. The entire scheme has some similarities with a conventional *vectored interrupt*. External events is thus rapidly gaining the microprocessors attention which ensures a minimal interrupt latency time.

THOR exception handling

THOR exception handling has adapted the Ada language definition. To each fragment of code, or rather, each subprogram, there exists an "Exception Information Block", dynamically allocated and initialised before the subprogram entrance. This provides for different exception processing in different subprograms of same type of exception. This strategy obviously decrease the overhead required by a software kernel. To each exception there is a corresponding Exception number. The first 15 numbers are defined by hardware (table 2.26) but they can also be raised by software, remaining exception numbers are user defined.

2.8 Conclusions

Historically the major goal with developing new processor architectures has been to acheive *increased performance without dramatical increase of the cost*. The RISC approach, single cycle execution, offers high performance at resonable costs. Current RISC architectures are characterisized by:

- a large register file
- instructions that are fast to decode
- pipelined execution
- few addressing modes
- fixed instruction format

Number	*Exception*	*Description*
1	Bus Error	An external memory access failed to complete within 255 clock cycles.
2	Address Error	Attempt to access non physical or protected memory
3	Data Error	Uncorrectable error in data read
4	Instruction Error	Attempt to execute privileged instruction in user mode, or illegal instruction
5	Jump Error	Attempt to jump to, call or return to an invalid address
6	Reserved	
7	Reserved	
8	Constraint Error	A constraint of a CLL or CUL instruction was not satisfied
9	Access Check	Attempt to use a zero indirect address with the PSHX and POPX instructions, i.e follow a null pointer
10	Storage Error	Attempt to access memory outside the task's stack in user mode
11	Overflow Check	Overflow of signed integer or float arithmetic operation
12	Underflow Check	Underflow or denormalised result of float arithmetic operation
13	Division Check	Attempt to divide by zero
14	Illegal Operation	Illegal float arithmetic instruction caused by any denormalised/NaN operand
15	Tasking Error	Reserved for future use, currently not raised by hardware

Table 2.26: THOR exception numbers

In combination with careful memory hierarchy design, memory management units and floating point support, on chip or of chip by a coprocessor, these RISC processors seems suitable for embedded systems such as laser printers and other general purpose systems such as work-stations.

These observations are true for MC88100, I80960, R2000, Am29000 and SPARC. T800 and THOR shows another approach, these processors facilitates stack architectures which eliminates the need for a large register file. The instruction format is flexible while pipelined execution is maintained and few addressing modes are available.

Chapter 3

Real-Time System requirements

The design of reduced instruction set computers is guided by a design philosophy. It does not rely upon inclusion of a set of required features. There is no strict definition of what constitutes a RISC-design. However one may observe some common features. Pipelining is used in all RISC designs to provide simultaneous execution of multiple instructions. Simple instructions/addressing modes are used. This results in an instruction decoder that is small, fast and easy to design. With few addressing modes it is easier to map instructions onto a pipeline since the pipeline can be designed to avoid computation related conflicts. A carefully designed memory hierarchy is required for increased processing speed. A typical hierarchy includes high speed registers, cache (buffers) located on the CPU chip, memory management schemes to support off-chip cache and memory devices. The hierarchy must permit fetching of instructions and operands at a rate that is high enough to prevent pipeline stalls. Optimizing Compilers provide a mechanism to prevent or reduce the number of pipeline faults by reorganizing code.

From these observation we may conclude that RISC designs are intended for personal computers, work-stations and embedded systems where high performance is the primary goal.

In a real-time system, high performance is of course desirable. However the set of needs extends due to the specific tasks that the system should carry out. Real-time systems must provide rapid process switches and fast interrupt handling so as to meet time requirements. It must be able to perform real-time synchronisation of events. High-level language support and optimizing compilers are essential and fall into several underlying characteristics, for example:

- The instructions set should be a suitable target for high level languages used for real-time systems.
- Real-time systems require reliable memory devices, which in turn are large, power consuming and expensive. Consequently there is an implicit demand for compilers that produce dense code for the target processor.

- Subprograms are frequently used by application programmers and the processor should provide for subprogram calls with a minimum of overhead.

This chapter will discuss essential real-time system support provided by the studied processors. That includes subprogram calls, interrupt handling, process switch, real-time synchronisation facilities and debug support. Other aspects on the high level language support are not within the scope of this work.

3.1 Subprogram Calls

A subprogram call is a result of a high level language function/procedure call statement. In the case of *func(p1,p2 ... ,pn);*, the compilers function is to generate code for a subprogram call with n parameters. The traditional way to do this is to push the n parameters on stack and perform a subroutine (subprogram) call, then modify the stackpointer and continue. But this requires at least n memory accesses with possible penalty and degraded performance. Thus, it is preferable to hold and pass the parameters in registers. This is made possible by a large number of registers and conventions for the use of these register. That is; directives for the compiler writer of how to dispose the register set. The register usage conventions are connected with the processor architecture and these conventions will be described in the next paragraphs.

Besides parameter passing a compiler generates specific code for a subprogram, which is to be executed before the actual, translated high-level program (*subprogram entry*) as well as after the high-level program (*subprogram exit*). *Subprogram entry* code, should for example, allocate memory required for local variables, possibly perform stack checking, check pointers for valid memory accesses i.e limits for memory space that the subprogram may access. Some high level languages, such as ADA, supports differentiated error handling; i.e different subprograms use different error handling routines for the same type of error, which will cause extra overhead during run-time. As examples of *subprogram exit* code we have deallocation of local variables, placing return values at appropriate location and possibly error checking.

In real-time systems it often turns out that stack-checking, memory access violation checking and differentiated error handling must be discarded in favour of more dense code and faster execution. However, during the debug phase of real-time system software, these facilities may be of great importance.

3.1.1 MC 88100 register conventions

The outline of the MC88100 general purpose registers is described in paragraph 2.1.3, page 26 The register usage are as follows:

- Register *r0* always contains zero, which is used in instructions requiring the constant zero as an operand. This is a hardware convention; the software can write to *r0* but this operation has no effect.
- Register *r1* contains the return pointer generated by *bsr* or *jsr* to subroutine instructions. This is a hardware convention; both of these instructions overwrite the data in *r1* when they execute. However, this register is not protected; software can read or overwrite the return pointer (or any other data) contained in *r1*.
- Registers *r9* through *r2* are used for passing parameters to a called routine. These registers can be overwritten by the called routine. This is a software convention.
- Registers *r13* through *r10* are used for temporary storage. They can be overwritten by a called routine but do not contain parameters for the called routine. This is a software convention.
- Registers *r25* through *r14* are used as data storage for the current routine. A called routine must ensure that the data in these registers is returned without modification when it finishes execution. These registers must be preserved for the calling routine. This is a software convention.
- Registers *r29* through *r26* are reserved for use by the linker, which is a software convention.
- Register *r30* is reserved for use as a software frame pointer, which is a software convention.
- Register *r31* is reserved for use as a software stack pointer, which is a software convention.

Thus, the architecture gives good support to subprogram calls with up to eight parameters passed in registers. It should be noted though, that nested subprogram calls require stacking of registers used for parameters during the previous call.

3.1.2 I80960KB register conventions

The 80960 provides sets of 16 local register for each subprogram. There are 4 sets of these registers on chip. If a nesting depth larger than 4 is used, the processor automatically saves the local register contents on stack, thus freeing local registers for use by the subprogram.

The global register g15 is reserved for use as a Frame Pointer. Local registers r0,r1 and r2 are reserved for use as: Previous Frame Pointer, Stack Pointer and Return Instruction Pointer, respectively.

Parameters are passed using global registers accessible regardless of which local register set is currently active, thus 15 parameters could conveniently be passed to (or from) a subprogram. Nested calls therefore requires stacking of parameters.

3.1.3 Am29000 register conventions

The Am29000 utilises a large, on chip register set which is organized as a run-time stack. When a subprogram is called, a new activation record, or "stack frame" is allocated. This record includes local variables, arguments to the subprogram and a return address. A compiler targeted to the Am29000 should use two run-time stacks for activation records: one for often used scalar data and another for structured data and additional scalar data. The scalar portion of the activation record can then be mapped into the processor's local registers, because of the stack-pointer addressing which applies to the local registers.

Allocation and de-allocation of activation records can occur largely within the confines of the local registers. The term "stack-cache" refers to the use of local registers to cache a portion of the activation record stack.

The principle of locality of reference - which allows any cache to be effective - also applies to the stack cache. The entries in the stack cache are likely to remain there for re-use, because the dynamic nesting depth of activated procedures tends to remain near a given depth for long periods of time. As a result, the size of the run-time stack does not change very much over long intervals of program execution.

Since activation records are allocated and de-allocated within the local registers, most procedure linkage can occur without external references. Also, during procedure execution, most data accesses occur without external references, because the scalar data in an activation record is most frequently referenced. Activation records are typically small, so the 128 locations in the local register file can hold many activation records from the run-time stack.

3.1.4 MIPS R2000 register conventions

Mips R200 assembler denotes the 32 general purpose registers $0,$1 .... $31. The register usage are as follows:

- Register **$0** always contains zero, which is used in instructions requiring the constant zero as an operand.
- Register **$1** is reserved for the assembler.
- Registers **$2** and **$3** are used for expression evaluations and to hold integer function results. They are also used to pass the static link when calling nested procedures.
- Registers **$4** through **$7** are used to pass the first 4 words of integer type actual arguments; their values are not preserved across procedure calls.
- Registers **$8** through **$15** are used for temporary storage. Their values are not preserved across procedure calls.
- Registers **$16** through **$23** are saved registers; their values must be preserved across procedure calls.

- Registers **$24** and **$25** are used for expression evaluation; their values are not preserved across procedure calls.
- Registers **$26** and **$27** are reserved for the operating system kernel.
- Register **$28** contains the global pointer.
- Register **$29** contains the stack pointer.
- Register **$30** is a saved register (like $16 ...$23).
- Register **$31** contains the return address. Used for expression evaluation.

According to software conventions, four (or fewer) parameters could be passed in registers.

3.1.5 SPARC register conventions

The organisation of SPARC register windows was described in paragraph 2.5.3, page 58. Figure 2.1,(page 59) shows how 32 general purpose registers are divided into 4 groups. The "*outs*" (8 registers) in the active window are are identical to the *ins* of the next window. The *out* register r[15] is used for saving current address by the CALL instruction. Thus seven parameters may be passed, using registers, during a subprogram call. By software convention, fewer parameters can be assumed thus providing additional local registers. If a nesting depth exceeds 4, a trap occurs and the real-time kernel must take approriate actions.

3.1.6 T800 /THOR

Both T800 and THOR are stack architectures. Consequently parameters are passed via the stack. In THOR, 32 words from Top of Stack and downwords are reflected in registers on chip. A writeback mechanism provide for consistency with memory contents. The writeback is simultaneous with other processor activities.

3.2 Deviation from normal execution

By "normal flow of instruction execution" we generally mean the execution of sequential instructions in memory, JUMP, BRANCH and CALL instructions, in short an easily predetermined behaviour from the computer system. A break in normal flow of instruction execution is an event of some kind, such as:

- An interrupt, normally caused by an external device pulling a dedicated pin on the processor active. That is: A system activity.

- An exception, caused by the execution of an instruction preventing finishing execution of the instruction. Examples are: Arithmetic faults (divide by zero, attempt to draw the root from a negative number etc), violation of permissions such as attempt to access supervisor memory in user mode, attempt to execute privileged instructions etc. An exception is also raised when a page fault occur in a virtual memory system. An exception condition may leave the registers in a consistent state so that the elimination of the cause and the restart of the instruction will give correct results. Such exceptions are often called *faults*. An exception that potentially leaves the registers and memory in an indeterminate state is often called *abort*.
- A trap, caused by a special instruction and providing method of implementing operating system calls etc. A trap may be conditional such as TRAP on OVERFLOW and used in conjunction with arithmetic operations.

Real-time systems are event-driven, i.e an external event should affect the internal state of the system and/or require som form of attention. In a real-time system, the ability to respond to such an event within a specified time is a major requirement. Hardware support for event handling is provided by the processor's interrupt mechanism. The following paragraphs describes these mechanisms.

3.2.1 MC 88100

Upon recognition of an interrupt the MC 88100 acts as follows:

1. Finish current instruction (synchronize)
2. Freeze all pipelines except the data unit
3. Allow data unit to complete (or fault)
4. Freeze all shadow registers and copy the PSR to the TPSR.
5. Set new PSR to indicate exception processing
6. Generate vector
7. Prefetch vector and vector+4

3.2.2 I80960KB

Whenever the processor receives an interrupt signal, it performs the following action;

1. It temporarily stops work on its current task, whether it is working on a program or another interrupt procedure.
2. It reads the interrupt vector.

3. It compares the priority of the vector with the processor's current priority.

4. If the interrupt priority is higher than that of the processor, the processor continues as described below.

5. If the priority is equal to or less than that of the processor the processor sets the appropriate priority bit and vector bit in pending interrupt record and continues work on its current task.

When the processor in executing state decides to service the interrupt it:

1. saves the current state of process controls and arithmetic controls in an interrupt record on the stack that the processor is currently using.

2. if the execution of an instruction was suspended the processor includes a resumption record for the instruction in the current stack and sets the resume flag in the saved process controls.

3. switches to the interrupted state.

4. sets the state flag in the process controls to interrupted, its execution mode to supervisor, and its priority to the priority of the interrupt.

5. clears trace-fault-pending and trace-enable flags.

6. allocates a new frame on the interrupt stack and switches to the interrupt stack.

7. sets the frame return status field.

8. performs an implicit call-extended operation at the address specified by the interrupt table for the specified interrupt vector.

3.2.3 Am29000

The following operations are performed by the processor when an interrupt or trap is taken:

1. Instruction execution is suspended

2. Instruction fetching is suspended

3. Any in-progress load or store operation is completed. Any additional operations are cancelled in the case of load-multiple and store multiple.

4. The contents of the Current Processor Status Register are copied into the Old Processor Status Register.

5. The Current Status register is modified to indicate interrupt(trap).

6. The address of the first instruction of the interrupt or trap handler is determined.
7. The processor determines whether or not the first instruction is in instruction ROM.
8. An instruction fetch is initiated using the instruction address as determined in previous steps. At this point, normal execution resumes.

3.2.4 MIPS R2000

An interrupt exception occur as a result of hardware signal or by execution of special instructions.

1. The R2000 branches to the general exception vector for this exception.
2. the *IP* field in the *Cause* register shows which of six external interrupts are pending, and the *SW* field in the *Cause* register shows which of two software interrupts are pending. More than one interrupt can be pending at a time.
3. The R2000 saves the *Kernel/User previous, Interrupt Enable previous, Kernel/User current,* and *Interrupt Enable current* bits of the *Status* register in the *Kernel/User old, Interrupt Enable old, Kernel/User previous* and *Interrupt Enable previous* bits respectivly, and clears the *Kernel/User current* and *Interrupt Enable current* bits.

3.2.5 SPARC

An interrupt is a special case of trap condition. A trap causes the following action:

1. It disables traps
2. It copies the S field of the PSR into the PS field and then sets the S field to 1.
3. It decrements the CWP by 1 modulo 7.
4. It saves the PC and nPC into r[17] and r[18], respectively of the new window.
5. It sets the *tt* field of the TBR to the appropriate value.
6. If the trap is not a reset, it writes the PC with the contents of TBR, and the nPC with the contents of TBR+4. If the trap is a RESET, it loads the PC with 0 and the nPC with 4.

3.2.6 T800

The T800 *EventReq* and *EventAck* pins provide an asynchronous handshake interface between an external event and an internal process. When an external event (interrupt) pulls

EventReq active the external event channel (additional to the external link channels) is made ready to communicate with a process. When both the event channel and the process are ready the processor pulls *EventAck* active and the process, if waiting, is scheduled.

Only one process may use the event channel at any given time. If no process requires an event to occur *EventAck* will never be activated.

If the process is a high priority one and no other high priority process is running, the latency is typically 19 processor cycles. Setting a high priority task to wait for an event input allows the user to interrupt a transputer program running at low priority. The following functions take place:

- Sample *EventReq* at pad and synchronise.
- Edge detect the synchronised *EventReq* and form the interrupt request.
- Sample interrupt vector for microcode ROM in the CPU.
- Execute the interrupt routine for Event rather than the next instruction.

The time taken activating *EventReq* to the execution of the microcode interrupt handler in the CPU is four cycles.

3.2.7 THOR

THOR interrupt handling is described in paragraph 2.7.5. As opposed to a more general interrupt handling approach, THOR gives hardware support for synchronisation between processes running on different processors. On the other hand, in a single processor system, interrupts may be treated in a more conventional and general manner.

The hardware defined exceptions are listed in table 2.26. All of these exceptions can also be raised by software. The `Exception Register (ER)` is used when an exception is raised. It points to an Exception Information Block in the stack. This block holds the program counter for the exception handler to call, and the pointer to the next (outer scope) Exception Information Block. When a hardware generated exception is raised, the following actions occur:

- Top of stack is set to the value of `ER`,
- Stack top value, i.e address of the exception handler is popped into `PC`,
- Stack top value (now the new `ER`) is popped into `ER`,
- The exception number is pushed, according to the preceding table.

Control transfers to appropriate exception handler.

3.3 Task Switch

In a real-time environment each program under execution constitutes a *process*. Another name for a process is a *task*, both terms will used here. For each process there must exist:

- A Process Control Block (PCB) used by the operating system to maintain the process. Entries in the PCB may also be used by the process itself.
- Data Space, where the process data resides.
- Code Space, where the process code resides. May in some cases be shared by several processes.

In addition to this we must add the procesor context to fully describe a process *at any time*. A processor's context is characterised by:

- Accessible register contents
- Internal (unaccessible) register contents
- Processor internal state

During a context switch at least the processor internal state and the internal register contents must be preserved, or the processor must be allowed to proceed until a well defined state is reached. For example, the current instruction is allowed to complete. Furthermore, to allow restart of the interrupted program, the status register, stack and program counter must be saved. For a process switch, obviously the entire processor context must be saved which also includes the accessible registers.

A common method is to let the process stackpointer reside in the upper region of data space (growing downwards). The stackpointer itself, upon a process switch, is stored in the actual process PCB. That is: A minimum of operations performed to freeze a process and maintain the ability to restart it at any later time for the operating system must be:

1. Save the entire processor context by pushing it onto the stack.
2. Store stackpointer value in the PCB.

The process can be restarted simply by loading the stackpointer (from PCB) and pulling processor context from the stack.

For a complete process Switch the old process must be preserved, a new process must be selected and started. That is: at least two processor context switches and the selection contribute to the total time required. In a system with several runable processes the operating system must choose the one with highest priority. There might for example be

Processor	Processor Cycles
MC88100	148
I80960KB	136
Am29000	133
MIPSR2000	145
SPARC	144
T800	hardware implemented
THOR	hardware implemented

Table 3.1: Number of cycles required to search the PCB-list

Processor	Register file save(cycles)	Register file restore (cycles)
MC88100	62	94
I80960KB	160	238
Am29000	195	195
MIPSR2000	62	62
SPARC	272	272
T800	[a] 1	[a] 1
THOR	[a] 1	[a] 1

Table 3.2: Number of cycles required for storing/restoring processor context

[a] Special hardware support for process switch makes these abundant

processes waiting for IO, or processes waiting for synchronization with other processes in the system. In other words: Every process PCB has to be checked regarding the process status (runable or not) and priority to pick the runable process with the highest priority. The effiency of this activity is of major importance for a real time system where the overall function relies on the systems ability to respond to external events and schedule an appropriate process.

As an example of process switch in small real-time systems a simple case was analyzed for the studied processors. A real-time system with ten runable processes is considered. A complete process switch is assumed accomplished by: storing old process context - selecting a new process - load the new process context into processor registers. Table 3.1 summarises the processor cycles required to complete a search in the list of PCB:s for each processor. The number of cycles required for storing/restoring processor context is given in table 3.2. From these figures and the systems clock frequency the total time required to perform a process switch could be estimated (Table 3.3).

For THOR and T800 there is hardware support for rescheduling while for the other processors, process switch had to be programmed. Assembly language listings of these programs, and notes about the calculations giving the figures are gathered in Appendix B.

Processor	Freq. (MHz)	Total Time (mikro seconds)
MC88100	25	12.2
I80960KB	25	21.4
Am29000	40	13.1
MIPSR2000	40	6.8
SPARC	40	17.2
T800	30	less than 1
THOR	20	less than 1

Table 3.3: Total time required for a process switch (estimated)

3.4 Real Time System Support

As stated earlier in this chapter a real-time system should provide synchronisation between events. This requires data structures for wait- and delay queues and a timer function used to maintain system time and for process delay purposes. Another important issue is the problem with synchronising (local) system time with "global" time, i.e different real-time systems in cooperation should be able to use this global time for different purposes. Moreover, the system should provide an accurate delay time for processes that require it. It should be noted that we are really addressing an issue that is different from a conventional real-time clock in a work-station application.

Real-time system software needs careful debugging and testing. Traditionally, processors give support for this through a "trace"-instruction, i.e by executing one machine instruction at a time and then returning control to some debugging tool or monitor. In a real time system, which is event driven, a more extensive support would be desirable to catch transient erronous behaviour resulting from special occurances of events.

The environments in which real-time systems mostly reside and the tasks that they most often perform makes contiguous service or service during operation difficult or impossible to carry out. This makes hardware debugging facilities and fault-tolerant aspects central in real-time system design. The following paragraphs summarize support related to:

- Timer facilities
- Software/Hardware debugging
- Fault tolerance

3.4.1 MC88100

The processor can be forced to a "serial mode" by setting one bit in the status register. This, significantly reduces machine throughput but is useful for debug purposes. Besides

from that, software debugging must be accomplished by the use of general trap handling facilities.

MC88100 include comparator circuits at the output to support fault detection. There are several possible configurations possible for master/checker operation and other redundant designs.

3.4.2 i80960

To support debugging systems, the i80960 provides a mechanism for monitoring processor activity by means of trace events. The processor can be configured to detect seven different trace events, including the instruction execution, branch events, calls, supervisor calls, returns, prereturns and breakpoints. When the processor detects a trace event, it signals a trace fault and calls a fault handler.

3.4.3 Am29000

Software debug is supported by the Trace Facility which guarantees exactly one trap after the execution of any instruction in a program being tested. This allows a debug routine to follow the execution of instructions, and to determine the state of the processor and system at the end of each instruction.

The processor has a built in Timer Facility which can be configured to cause periodic interrupts. The Timer Facility consists of 2 special purpose registers , the Timer Counter and the Timer Reload registers, which are accessible only to supervisor mode programs. The Timer Facility may be used to perform precise timing of system events.

Each Am29000 output has associated logic which compares the signal on the output with the signal which the processor is providing internally to the output driver. The processor signals situations where the output of any enebled driver does not agree with its input. For a single processor, the output comparision detects short circuits in output signals, but does not detect open circuits. It is possible to connect a second processor in parallel with the first, where the second processor has its outputs disabled due to the Test mode. The second processor detects open-circuit signals, as well as providing a check of the output of the first processor.

3.4.4 R2000

The instruction set includes a BREAK instruction which causes a BREAK-trap to occur. Control is transferred to the applicable system routine.

3.4.5 SPARC

Software debugging is only supported by the means of general trap instructions.

3.4.6 T800

Software debugging is supported by a variety of instructions that affects status bits. When the processor "Analyse" pin is taken high the transputer will halt at a descheduling point.

The T800 offers the possibility to respond differently on interrupts depending on the processor's current mode.

The T800 incorporate a timer. The implementation directly supports the occam model of time. Each process can have its own independent timer which can be used for internal management or real time scheduling. Hardware redundancy is acheived by the means of multiple transputer configurations.

3.4.7 THOR

THOR has a built in real time clock to keep track of system time. Furthermore, each process has a Delay register, causing interrupt after a specified delay. This provides for an efficient implementation of a high level language (real-time) delay function since kernel software is released from polling a "delay queue" each time a scheduling is to be performed. Also the unique TASK-instructions implemented in THOR serves as a powerful support for introducing the ADA-task concept as constituting a process in a real-time system. There are instructions for scheduling and delaying tasks as well as performing "rendezvous" between tasks.

THOR provides hardware selfcheck as well as an Error Detection And Correction (EDAC) unit, for check of processor communication with memory, on chip.

3.5 Conclusions

The large register file present in several of the studied processors allows optimizing compilers to arrange for fast subprogram calls by passing parameters in registers. When a large register file is available there is a good chance that all, or most of, the parameters could be passed this way. The MC88100 and R2000 are good examples. Both architectures provide large register sets and the usage of these registers could be optimized by a compiler. The drawback here comes in the case of nested subprogram calls: only the highest program level can take full advantage of this construction. With a register window design, as in SPARC or I80960KB, it is possible to increase the number of program levels that will benefit from parameters passed in registers. However, the fundamental problem remains

since even very large register files may be exhausted. A stack architecture such as T800 or THOR provides a natural convention: stacking of all parameters. This is simple and straightforward and there are no difficulties with nested calls. Furthermore, with THOR, since the 32 bytes close to top of stack are present in on chip registers it is possible to take advantage of the rapidness with register passing without having to bother with save and restore in the case of nested calls. Am29000, finally, provides a solution similar to SPARC. The large number of registers and the use of a run-time stack made up by registers could be thought of as register windows where the calling and the called program share a set of registers.

All of the studied processors treat interrupts in a similar manner. The elapsed time between an interrupt and the point at which processing starts at the appropriate interrupt handler address can be regarded as the *interrupt latency time* and is divided into three phases:

1. Finish current instruction (does not apply to exception).
2. Check interrupt priority level versus current processor level, i.e whether the interrupt should be serviced or not.
3. Save enough processor status to be able to continue processing after the interrupt has been serviced.

Finishing current instruction causes no significant delay provided that no possible instruction (from the instruction set) may last for more than one, or a few cycles. This is true for today's RISC-architectures. Processor activities are assigned priorities determined by the type of activity. For example, reset handling has the highest priority and thus cannot be interrupted. Interrupts are assigned priorities to predetermine the behaviour when simultaneous events occur and to assure that no high priority processor activity may be interrupted. The saved processor status required to restart an interrupted program is determined by the activities required to service the interrupt. In general, the processor does not save general register contents when servicing an interrupt. The interrupt handler routine is responsible for saving and restoring register contents which might be altered by the service routine.

Since a real-time system, according to the conventions described in the Introduction of this thesis, must have *the ability to respond within a finite time*, and events, external from the system, may require immediate attention, the question of fast rescheduling becomes important. Process switches in real-time systems can be a time-consuming matter. Moreover, since processes are created and removed dynamically it becomes very difficult to predict the time spent on these activities. In analyzing the processor's ability to perform fast task-switches the important observations are:

- The register file should be reasonably sized since a task-switch (process-switch) requires the entire processor context to be exchanged.

- Hardware support for task-switches is an essential feature to reduce the time spent for rescheduling.

A large register file will delay processor context switch significantly. Therefore, a large register file, which has proved essential for increase of system performance could become a bottleneck with unpredictable consequenses. From paragraph 3.3 we may conclude that a stack architecture, such as T800 or THOR, with hardware support for process switches provides considerably better performance than any of the other processors.

In applications where speed is far beyond human control and the tolerances are small there are often needs for precise time-handling, i.e processes that require a precise delay should get that delay and nothing else. Three of the studied processors addressed these issues with on-chip timer facilities: Am29000, T800 and THOR.

Real-time systems are used to maintain surveillance and control processes where a system failure might have disastrous consequenses: Nuclear plants, aircrafts, spacecrafts just to mention a few. In the years to come we will see even more applications with steadily growing demands for reliability and security. Consequently hardware/software debugging support and fault tolerance are also important parts of real-time system design. All of the processors provide some kind of software debug support. Furthermore T800 provides facilities that makes real-time debugging possible to a limited extent. Built-in fault tolerance support such as selfcheck, memory error detection (and correction) is provided only by THOR while MC88100 and Am29000 provides support for redundant designs.

Chapter 4

System Hardware Considerations

A physical real-time system, when used in aerospace for example, must meet some important needs. It should be small in size, have low weight and low power consumption. The system should be reliable and thus only high quality components, at least military qualified, should be used. Fault tolerance support is desirable and memory errors must be detected and preferably corrected. (See [Jan90] for a thourougly description of requirements on microcomputers in critical applications.) The purpose with this chapter is to highlight how demands on system hardware impacts on system performance and dependability.

This chapter discusses six computer designs that use the Inmos T800 Transputer, the Saab-Ericsson Space THOR and the Cypress SPARC microprocessors respectively in order to evaluate hardware aspects of the three processors in two different configurations:

- A Real-time System application, called the *High Dependability Oriented configuration*, **(HDO)**. The HDO configuration should be thought of as an on board computer for a space craft.
- A general purpose (embedded) system application called the *High Speed Oriented configuration*, **(HSO)**.

The designs, which not are realised, are considered comparable at cost and analyzed to give an estimation of:

- maximum possible instruction execution rate
- required number of devices
- area of printed circuit board

- power consumtion
- failure rate

4.1 General notes on the designs

In the schematics (see appendix C), readability is emphasised. The diagrams are not complete but rather focus on devices with major impact on the configuration function and performance. For each design a description of a memory read cycle is given and analysis is carried out.

Estimations are performed using *worst case assumptions*. The designs are optimised for *the highest possible clockfrequency* i.e no attempt is made to reduce wait state penalties due to high clock frequence.

4.2 Execution Rate Estimation

The instruction mix is made up from:

- x_1 = percentage arithmetical/logical instructions
- x_2 = percentage jump/branch instructions
- x_3 = percentage load/store instructions
- x_4 = percentage floating/point instructions

as a consequense: $x_1 + x_2 + x_3 + x_4 = 1$ for a large number of executed instructions.

Parameters that describes the processor in effect are:

- X_1, the number of processor cycles required to execute an arithmetical/logical instruction
- X_2, composed by: $0.1X_{21} + 0.9X_{22}$ where
  - X_{21} is the number of processor cycles required for a "branch not taken" instruction
  - X_{22} is the number of processor cycles required for a "branch taken" instruction

  Hence, it is assumed that 90% of all conditional branches are taken.

- X_3, denotes the number of processor cycles required to execute a load/ store instruction. For simplicity these are considered equal in this sense.
- X_4, denotes the number of processor cycles required for the execution of a floating point instruction.

In order to describe wait state penalties and different instruction formats the following parameters are introduced:

- W denotes the number of wait states required for a read bus cycle, determined by the system configuration.
- U denotes the averages number of instructions that becomes available for execution as a result of one (32+8 bits) fetch. If, for example 70% of the instruction set consists of instructions encoded in 16 bits and the rest are encoded in 32 bits, then:

$$U = 0.7 * 2 + 0.3 = 1.7$$

- $Y(W, U)$ denotes average cycles required to feed the processor with one instruction. This is a function of wait state penalties and instruction format:

$$Y = \frac{1 + W}{U} \frac{cycles}{instruction}$$

Since instruction fetch and execution is performed simultaneously in a pipe-lined architecture we write:

$$Z_1 = max[X_1, Y(W, U)]$$

$$Z_2 = max[X_2, Y(W, U)]$$

$$Z_3 = X_3 + W$$

$$Z_4 = max[X_4, Y(W, U)]$$

We obtain an expression for the Execution Rate Estimation, ERE:

$$ERE = Z_1 x_1 + Z_2 x_2 + Z_3 x_3 + Z_4 x_4 (cycles)$$

where ERE denotes the average number of cycles required to execute one instruction. Including the cycle time CT in seconds, we arrive at a final expression for the execution rate:

$$ER = \frac{1}{ERE \quad CT} \frac{instructions}{second}$$

4.3 Memory Power Consumtion

The memory used in the HDO configuration, (64k nibble) Cypress CY7C194 is a 24 pin device with 35 ns access time. Memory is organized as 40 bits words (32 data and 8 check bits) thus each memory access will activate all of the ten devices.

If we define the *Average Memory Activity, (AMA)* as the fraction of processor cycles that accesses memory in an instruction mix, the memory power consumtion could be estimated as:

$$P_{average} = AMA \;\; P_{active} + (1 - AMA)\, P_{standby}$$

For this memory device:

$$P_{active} = 650\, mW$$
$$P_{standby} = 100\, mW$$

Determination of AMA is complicated by several factors. The memory device needs typically one cycle to enter standby mode after beeing accessed. Obviously, the memory power requirement depends on the instruction execution order. If, for example, load/store instructions were ordered as every other instruction rather than consecutive instructions then there would be more memory "active" cycles since we actually need two consecutive cycles that do not access memory to reach the "standby" mode. In the estimations, the instruction order as well as wait state cycles are ignored and AMA is considered a function of:

1. Instruction Fetch Rate
2. Instruction Mix
3. Instruction Execution Timing

Instruction Fetch Rate is limited by the instruction format. For example, with an instruction format of 32 bits and assuming single cycle execution of all instructions every cycle needs an instruction fetch. A shorter instruction format, i.e more dense code, will decrease the need for instruction fetches.

The *Instruction Mix* is essential since, for example, load/store instructions introduces extra memory accesses ,thus increasing AMA.

Instruction Execution Timing affects memory activity since the fact that all instructions do not execute in one cycle will reduce the need for instruction fetches. Thus the higher execution times, the lower the AMA.

Here, AMA is estimated by:

$$AMA = \frac{1}{U}\left(\frac{x_1}{X_1} + \frac{x_2}{X_2} + \frac{x_3}{X_3} + \frac{x_4}{X_4}\right)(\%)$$

4.4 Instruction Mix

The following instruction mix is assumed:

- 50% arithmetical/logical instructions
- 25% jump/branch instructions
- 10% load/store instructions
- 15% floating point instructions

4.5 Notes on the Failure Rate estimation

Failure rate estimation is carried out according to the MIL- HDBK-217-E. For temperature acceleration factor calculation the thermal resistivity factor was used whenever it was available from manufacturer's documentation. However, since such information was rare, assumptions had to be made about the junction temperature. For complex circuits, such as CPU:s and FPU a junction temperature of 110 degrees Celsius was assumed. For all others, a junction temperature of 80 degrees Celsius was assumed.

4.6 The HDO configurations

Special requirements for the HDO configuration are:

- microprocessor with 256kB primary memory
- only space qualified components
- low power consumtion
- small printed circuit board area

The HDO configuration designs consists of:

- cpu
- 256 kB of static random access memory
- error detection and correction circuitry
- real time clock

In the failure rate estimation for HDO configuration the following assumptions were made:

- Quality Factor = S (0.25)
- Voltage Factor = 1
- Application Environment Factor = Space Flight (0.9)

The T800 and SPARC designs both utilise an "error detection and correction unit" (EDAC). The introduced delay (36 ns, worst case for the EDAC in use) is inserted by the EDAC control and assures that memory "Ready" signal will not be asserted until correct data is guaranteed. THOR has a built in EDAC so there is no need for this unit in the THOR HDO configuration.

4.7 T800 HDO configuration

T800 chip running at 17.5 MHz is available in mil spec. Since the T800 has an on chip timer, no such peripheral device is required.

Component list

	Device	Qty	Power [mW]	Area [mm2]	FITS
U1	T800-G17S	1	1200(1	1451	532
U2-U5	74ACT245	4	40	220	3
U6	74ACT08	1	30	154	3
U7	74ACT244	1	40	220	3
U8,U9	74HCT373	2	38	220	3
U11	74ACT04	1	34	154	3
U12	OT05	1	100	270	27
U13,U14	54HCT393	2	30	220	3
MU1-MU10					
	CY7C194(35)	10	189(2	255	218
EU1	IDT49C460B	1	625	1944	92
EU2	CYC7C361-L66DMB	1	750	280	170
EU3	74ACT32	1	29	154	3
EU4	OT050	1	100	270	27
EU5-EU8	74ACT245	4	40	220	3
EU9	74ACT244	1	40	220	3

1) Estimated for the current application
2) Average according to AMA

4.7.1 T800 Read memory cycle (external memory)

- T1: Address setup time before address valid strobe
- T2: Address hold time after address valid strobe
- T3: Time for the bus to go to tristate on a read cycle, or to present valid data on a write cycle
- T4,T5: Time for the read or write data pulse
- T6: Time for the bus to remain in tristate after the end of read, or for data to remain valid after the end of write

For the selected device, 1 Tm = 28.5 ns.

1. Address is latched at the falling edge of T1. Address setup time is "a-8" = 20.5 ns. The 373 requires typically 5 ns, thus it is sufficient with T1 = 1 Tm.
2. Address hold after falling edge of T1 is "b-9" = 19.5 ns. The 373 needs typically 6 ns, thus T2 = 1 Tm.
3. For T3,T4 and T5, CS* is asserted at the end of T1, during a read cycle, data is latched at the falling edge of T5. Buffer propagation delay is 11 ns. T800 needs stable data 25 ns before it is latched, memory require 35 ns from CS*, the EDAC is 36 ns , Hence: (35+11+36+25) = 107 ns violates T3=T4=T5 = 1Tm (85.5 ns), and two extra Tm:s are required.

4. With T6 = 1 Tm we arrive at a total of 8 Tm, ie 228 ns for an external memory cycle.Thus a memory read bus cycle is equivalent to 228/57 = 4 processor cycles.

4.7.2 T800 HDO config execution rate

The following parameters were chosen to describe the T800 configuration:

$$X_1 = 2$$

$$X_{21} = 2, X_{22} = 4, X_2 = 3.8$$

$$X_3 = 2$$

$$X_4 = 8$$

The manufacturer claims that about 70% of executed instructions are encoded in a single byte [Inm89] p.195. From the current instruction mix we assume that 50% of the instructions are encoded in 8 bits, 30% of the instructions are encoded in 16 bits, the rest are encoded in 32 bits. This gives $U = 2$ and with $W = 3$ from the previous section we have:

$$Y(W, U) = 2$$

Thus:

$$Z_1 = X_1 = 2$$

$$Z_2 = X_2 = 3.8$$

$$Z_3 = 5$$

$$Z_4 = X_4 = 8$$

leading to:

$$ER = \frac{1}{3.65\ 57}\frac{1}{ns} = 4.8\ MmixedIPS$$

For the memory activity we obtain:

$$AMA = 0.18$$

The total memory power requirement: **189 mW/device**.

4.8 THOR HDO configuration

The THOR has on-chip timer, thus no such peripheral device. Furthermore, THOR has a built in EDAC. Thus no such peripheral device either. The chip is not yet available. Actual figures concerning the THOR chip are obtained from simulations in Genesil Silicon Compiler, from these simulations assuming components satisfying military range requirements, the clock frequency will be 15 MHz.

Component list

	Device	Qty	Power [mW]	Area [mm2]	FITS
U1	THOR	1	1500	2450	78
U2-U6	74ACT245	5	36	220	3
U7	74ACT138	1	41	220	3
U8-U10	74ACT244	3	36	220	3
U11	OTO16	1	100	270	26
U12	74ACT04	1	30	154	3
U13,U14	54HCT393	2	26	220	3
MU1-MU10					
	CY7C194(35)	10	326(*	255	218

*) Average according to AMA

4.8.1 THOR Read memory Cycle

Assuming a need for 5 ns setup before data is latched. Taking into account the delay introduced by the '138, 16 ns. Memory requires 35 ns from CS* to valid data.Data bus buffers delay data by 11 ns. Thus wee need a cycle time:

$$15 + 16 + 35 + 11 + 5 = 82ns$$

The THOR cycle time is 67 ns and therefore, one wait state is required.

4.8.2 THOR HDO configuration execution rate

The following parameters were chosen to describe the THOR configuration:

$$X_1 = 1$$

$$X_2 = 1$$

$$X_3 = 2$$

$$X_4 = 4$$

95% of THOR instructions are encoded in 16 bits, the rest are encoded in 32 bits, hence $U = 1.95$ and with $W = 1$ from previous section:

$$Y(W, U) = 1.03$$

Thus:

$$Z_1 = Y(W, U) = 1.03$$

$$Z_2 = Y(W, U) = 1.03$$

$$Z_3 = 3$$

$$Z_4 = X_4 = 4$$

leading to:

$$ER = \frac{1}{1.673\ \ 67}\frac{1}{ns} = 8.9\ MmixedIPS$$

For the memory activity

$$AMA = 0.410$$

The total memory power requirement: **326 mW/device**.

4.9 SPARC HDO configuration

The CY7C601 chip running at 25 MHz is available in mil spec.

Component list

	Device	Qty	Power [mW]	Area [mm2]	FITS
U1	CY7C601	1	1750	1998	365
U2	CY7C344	1	1000	289	170
U3(*	CY7C602	1	1750	1600	358
U4-U6	74ACT244	3	59	220	3
U11	74ACT04	1	50	154	3
U12	MC146818	1	20	255	49
MU1-MU10					
	CY7C194(35)	10	650	255	218
EU1	IDT49C460B	1	625	1944	92
EU2	CYC7C361	1	750	280	170
EU3	74ACT32	1	44	154	3
EU4	OTO50	1	100	270	27
EU5-EU8	74ACT245	4	59	220	3
EU9	74ACT244	1	59	220	3

*) Not Available in mil spec

4.9.1 SPARC Read Cycle

Delays:

- A2-A17 to CS* PLD decoder 20 ns
- memory data setup time 35 ns
- edac delay 36 ns
- data bus buffer 11 ns

Required: From stable address to data latched:

$$20 + 35 + 36 + 11 = 102ns$$

Available (3 processor cycles):

$$120 + 7 - 3 = 124ns$$

Therefore, a bus read cycle will require 3 processor cycles which implies 2 wait states.

4.9.2 SPARC HDO configuration execution rate

The following parameters were chosen to describe the SPARC configuration:

$$X_1 = 1$$

$$X_2 = 1$$

$$X_3 = 3$$

$$X_4 = 4$$

A SPARC instruction is encoded in 32 bits so $U = 1$. From the previous section $W = 2$, and:

$$Y(W, U) = 3$$

thus:

$$Z_1 = Y(W, U) = 3$$

$$Z_2 = Y(W, U) = 3$$

$$Z_3 = 5$$

$$Z_4 = X_4 = 4$$

leading to

$$ER = \frac{1}{3.35 \;\; 40} \frac{1}{ns} = 7.5 \; MmixedIPS$$

The memory power-down facility may not be used since it is not possible to deassert memory chip-select during interlocks and so the total memory power requirement is **650 mW/device**

4.10 The HSO configurations

The HSO configuration is intendeded to estimate peak performance for a computer system with 1 MByte of memory. It consists of:

- microprocessor with 1 MByte of static random access memory

4.11 General Notes on the HSO configurations

The HSO configuration is accomplished by eliminating the EDAC circuitry and changing the memory devices from the HDO configuration. Glue logic, except from address decoding and bus buffers is implemented using macro cells. The memory is built from eight 64k*16 bit, 25 ns static rams. Address decoding is performed by high speed PAL devices, eliminating any address bus skew which otherwise may arise in high clock frequency systems. Failure Rate Estimations assumes commercial quality components and a "Ground, benign" environment.

4.12 T800 HSO configuration

Component list

	Device	Qty	Power [mW]	Area [mm2]	FITS
U1	T800-G30S	1	1200	1451	13907
U2	CY7C343	1	775	311	4527
U3-U7	74ACT245	5	71	220	490
U8-U11	74ACT244	4	71	220	490
MU1-MU8	CYM1624	8	2750	442	11242
MU9-MU10	CY7C338	2	750	226	3398

4.12.1 T800 HSO configuration execution rate

From the T800 read cycle diagram, and with the chosen configuration, we conclude that an external memory read cycle may be performed without wait state penalty. This also implies that there is nothing to gain from a cache memory. It should, however, be emphasised that the T800 internal memory (4 kByte) is not considered.

Hence $W = 2$, $U = 2$ leading to $Y(W, U) = 1.5$ and:

$$Z_1 = 2$$

$$Z_2 = 3.8$$

$$Z_3 = 4$$

$$Z_4 = 8$$

The HSO T800 configuration runs at 30 MHz and thus:

$$ER = \frac{1}{3.55 \;\; 33} \frac{1}{ns} = 8.5 \; MmixedIPS$$

4.13 THOR HSO configuration

Component list

	Device	Qty	Power [mW]	Area [mm2]	FITS
U1	THOR	1	1500	2450	78
U2	CY7C343	1	775	311	4527
MU1-MU8	CYM1624	8	2750	442	11242
MU9-MU10	CY7C338	2	750	226	3398
MU11-MU14	74ACT245	4	35	220	490
MU15-MU17	74ACT244	3	60	220	490

4.13.1 THOR HSO config execution rate

In the proposed configuration, THOR (25 MHz) does not require any wait state so: $W = 0$, $U = 1.95$ leading to $Y(U, W) = 0.51$ and:

$$Z_1 = 1$$

$$Z_2 = 1$$

$$Z_3 = 2$$

$$Z_4 = 4$$

finally:

$$ER = \frac{1}{1.75 \;\; 40} \;\; \frac{1}{ns} = 14.3 \; MmixedIPS$$

4.14 SPARC HSO configuration

Component list

	Device	Qty	Power [mW]	Area [mm2]	FITS
U1	CY7C601	1	3250	1998	14063
U2	CY7C602	1	2250	1600	13979
U3-U4	CY7C157	2	1250	397	11303
U5	CY7C604	1	3250	2554	14116
U6	CY7C343	1	775	311	4527
MU1-MU8	CYM1624	8	2750	442	11242
MU9-MU10	CY7C338	2	750	226	3398
MU11-MU14	74ACT245	4	95	220	490
MU15-MU17	74ACT244	3	95	220	490

4.14.1 SPARC HSO configuration execution rate

The SPARC configuration utilises a 64 kByte cache memory. Experience has shown that for a cache of this size, a hit rate of 90 % is probable. Denoting a 32-bit word fetched from the cache $Z_x(C)$ we write:

$$ERE = (Z_1 x_1 + Z_2 x_2 + Z_3 x_3 + Z_4 x_4)\, 0.10 +$$

$$(Z_1(C) x_1 + Z_2(C) x_2 + Z_3(C) x_3 + Z_4(C) x_4)\, 0.9$$

Timing analysis (carried out as in 4.9.1) shows that a cache miss will cost one wait state. An access whithin cache may be done without wait state. Hence:

$$Z_1 = 2$$
$$Z_2 = 2$$
$$Z_3 = 4$$
$$Z_4 = 4$$

and:

$$Z_1(C) = 1$$
$$Z_2(C) = 1$$
$$Z_3(C) = 3$$
$$Z_4(C) = 4$$

The HSO configuration runs at 40 MHz and from this:

$$ER = \frac{1}{1.735 \;\; 25} \;\; \frac{1}{ns} = 23 \;\; MmixedIPS$$

4.15 Summary of Results

As shown in table 4.2, the designs that were intended to show maximum performance clearly favours the SPARC. This is not very suprising. The SPARC cpu is available in a 40 MHz version and offers an architecture designed for single cycle execution of instructions. The figures of power requirement and the required board area indicates the price for this superior performance.

Table 4.1 however, gives another picture. The restrictions made on the real-time system configuration degrades total SPARC system performance notably, here it is comparable with both THOR and T800. The explanation lies in the absence of cache memory. and the presence of an EDAC which prevents the system from gaining from the benefits that the SPARC architecture offers. At the same time the expected failure rate and the total board area required are considerably larger than for THOR. The power requirement more than doubled compared to both T800 and THOR.

4.16 Conclusions

The system hardware considerations shows that in a real-time system design there is not very much to gain with a modern, general purpose RISC design such as SPARC. On the contrary, while the estimated performance for SPARC was just about the level of THOR, the board area became approximatly 40% larger, the power consumption 70% more and the expected failure became 45 % greater.

T800	THOR	SPARC	
17.5	15	25	Clock Frequency (MHz)
4.8	8.9	7.5	Mixed instruction execution rate (MmixedIPS)
32	24	27	Number of required devices
10307	7844	11254	Total area for devices (mm2)
5294	5271	13061	Total power requirement (mW)
3079	2320	3392	Failure Intensity (FITS)

Table 4.1: Summary: real-time system configuration

T800	THOR	SPARC	
30	25	40	Clock Frequency (MHz)
8.5	14.3	23.0	Mixed instruction execution rate (MmixedIPS)
21	19	23	Number of Required Devices
7730	8289	12785	Total area for devices (mm2)
26114	26020	36190	Total Power Requirement (mW)
119576	104767	169453	Failure Intensity (FITS)

Table 4.2: Summary: general purpose system configuration

Chapter 5

Concluding Remarks

Several descisions has to be made during the design of a new computer architecture. These descisions are based upon the designers experience as well as the systems requirements. From RISC-design concepts, several high performance microprocessors has been constructed.

In this thesis, we have studied how seven different microprocessors could perform in real-time systems. Four of these processors are general purpose RISC processors: Motorola 88100, Intel 80960kb, MIPS R2000 and Cypress SPARC, while three processors: AMD 29000, Inmos T800 and Saab-Ericsson Space THOR are targeted for real-time systems. From observations in this study we may conclude that important real-time requirements such as fault tolerance, precise time handling and rapid response on external events (process switch) and debug facilities has not had a major influence on the design of the general purpose processors. Rather, they are optimized for highest possible execution rate.

A real-time system requirement such as fault-tolerance places several restrictions on the system hardware design. It turns out that a high execution rate cannot be maintained due to the fact that memory devices for these applications are to slow. Moreover, since the communication between processor and memory must be checked (by dedicated logic) the memory bandwith is further reduced.

Precise time handling is essential for the control of several processes in real-time system applications. The general purpose processors relies on timer-functions provided by other devices in the system and this is probably not sufficient.

The ability to respond within a finite time on an external event is dependent of the processors support for a software process switch. Minimizing the latency of switch between to processes requires hardware support for this event. The general purpose processors do not provide such support.

Debug capabilities of hardware as well as software are necessary for the design of high dependable systems such as real-time systems. The general purpose processor's do not provide extensive support for debugging of a real-time system.

Am29000, despite that the manufacturer claims it to be designed for real-time systems, is similar to the general purpose processors.

T800 has several features which support real-time systems while THOR is the only, of the studied processors, that seems to be dedicated for use in real-time systems.

Bibliography

[Adv88] Advanced Micro Devices. *Am29000 streamlined instruction processor*, 1988.

[Bir85] Birnbaum J.S. , Worley W.S. Beyond risc: High precision architecture. *Hewlett Packard Journal*, vol 36(no 8):pp 4–10, August 1985.

[Hen84] Hennessy J.L. Vlsi processor architecture. *IEE Transactions on Computers*, vol C-33(no 12):pp 1221–1246, December 1984.

[Hen90] Hennessy J.L.,Pattersson D.A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann publishers, San Mateo, California, 1990.

[Hil85] Hill M.D. et alt. Spur: A vlsi multiprocessor workstation. Technical report, Computer Science Division, University of California, Berkeley, December 1985.

[Hil86] Hill M.D. et alt. Design decisions in spur. *IEE Computer*, vol 19(no 11):pp 8–22, November 1986.

[Hin86] Hindin H.J. Ibm risc workstation features 40-bit addressing. *ComputerDesign*, pages pp 28–30, February 1986.

[Inm89] Inmos limited. *Transputer databook*, second edition, 1989.

[Int88] Intel Corporation. *80960KB programmer's reference manual*, 1988.

[Jan90] Jan Torin. Characterisation of microcomputers for embedded real time systems - directions and basic criteria. Technical report, Department of Computer Engineering, Chalmers University of Technology, 1990.

[Mil83] Milutinović V.M., editor. *High Level Languages in Computer Architecture*. Computer Science Press Inc, Oxford, 1983.

[MIP87] MIPS Computer Systems Inc. *MIPS R2000 RISC architecture*, 1987.

[Mot90] Motorola Inc. *MC88100 RISC microprocessor user's manual*, second edition, 1990.

[Pat82] Patterson D.A.,Séquin C.H. A vlsi risc. *Computer*, pages pp 8–22, September 1982.

[Rad83] Radin G. The ibm 801 minicomputer. *IBM Journal R&D*, vol 27(no 3):pp 237–246, May 1983.

[ROS90] ROSS technology, Inc. *SPARC RISC user's guide*, 1990.

[Saa92] Saab Ericsson Space. *Stack RISC microprocessor instruction set architecture for prototype chip*, 1992.

[Sie82] Siewiorek D.P.,Bell C.G.,Newell A. *Computer Structures: Principles and Examples*. McGraw-Hill, Singapore, 1982.

[Smi83] Smith J.E.,Pleszkun A.R.,Katz R.H.,Goodman J.R. Pipe: A high performance vlsi architecture. *Proceedings of IEE International Workshop on computer systems organisation*, March 1983.

[Tab87] Tabak D. *RISC Architecture*. John Wiley & Sons Inc, New York, 1987.

[You82] Young S.J. *Real Time Languages: Design and Development*. Ellis Horwood, Chichester, 1982.

Appendix A

Instruction set summaries

A.1 MC88100 instruction set summary

Instruction	Operands	Name
ADD	rD,rS1,IMM16	integer add
ADD.CAR	rd,rS1,rS2	
ADDU.CAR	rD,rS1,IMM16	unsigned integer add
	rD,rS1,rS2	
CMP	rD,rS1,IMM16	integer compare
	rD,rS1,rS2	
DIV	rD,rS1,IMM16	integer divide
	rD,rS1,rS2	
DIVU	rD,rS1,IMM16	integer unsigned divide
	rD,rS1,rS2	
MUL	rD,rS1,IMM16	integer multiply
	rD,rS1,rS2	
SUB	rD,rS1,IMM16	integer subtract
SUB.CAR	rD,rS1,rS2	
SUBU	rD,rS1,IMM16	integer unsigned subtract
SUBU.CAR	rD,rS1,rS2	

Table A.1: MC88100 Integer Arithmetic Instructions

Instruction	Operands	Name
AND.U	rD,rS1,IMM16	logical and
AND.C	rD,rS1,S2	logical and
MASK.U	rD,rS1,IMM16	logical mask immediate
OR.U	rD,rS1,IMM16	logical or
OR.C	rD,rS1,rS2	logical or
XOR.U	rD,rS1,IMM16	logical exclusive or
XOR.C	rD,rS1,rS2	logical exclusive or

Table A.2: MC88100 Logical Instructions

Instruction	Operands	Name
JMP.N	rS2	unconditional jump
JSR.N	rS2	jump to subroutine
BB0.N	B5,rS1,D16	branch on bit clear
BB1.N	B5,rS1,D16	branch on bit set
BCND.N	M5,rS1,D16	branch on condition met
BR.N	D26	unconditional branch
TB0	B5,rS1,VEC9	trap on bit clear
TB1	B5,rS1,VEC9	trap on bit set
TBND	rS1,IMM16	trap on bounds check
	rS1,rS2	
TCND	M5,rS1,VEC9	conditional trap
RTE		return from exeption

Table A.3: MC88100 Flow Control Instructions

Instruction	Operands	Name
FADD.FSZ	rD,rS1,rS2	floating point add
FCMP.FSZ	rD,rS1,rS2	floating point compare
FDIV.FSZ	rD,rS1,rS2	floating point divide
FLDCR	rD,,fcrS	load from floating point control register
FLT.FSZ	rD,rS2	convert integer to floating point
FMUL.FSZ	rD,rS1,rS2	floating point multiply
FSTCR	rD,fcrD	store to floating point control register
FSUB.FSZ	rD,rS1,rS2	floating point subtract
FXCR	rD,rS,fcrS/D	exhange floatin point control registers
INT.FSZ	rD,rS2	round floating point to integer
TRNC.FSZ	rD,rS2	truncate floating point

Table A.4: MC88100 Floating Point Instructions

Instruction	Operands	Name
CLR	rD,rS1,IMM10	clear bit-field
	rD,rS1,rS2	
EXT	rD,rS1,IMM10	extract bit-field
	rD,rS1,rS2	
EXTU	rD,rS1,IMM10	extract unsigned bit-field
	rD,rS1,rS2	
FF0	rD,rS2	find first bit clear
FF1	rD,rS2	find first bit set
MAK	rD,rS1,IMM10	make bit-field
	rD,rS1,rS2	
ROT	rD,rS1,IMM10	rotate register (only 5 bits of IMM10 used)
	rD,rS1,rS2	
SET	rD,rS1,IMM10	set bit-field
	rD,rS1,rS2	

Table A.5: MC88100 Bit-Field Instructions

Instruction	Operands	Name
LD.SZ	rD,rS1,IMM16	load register rD from memory at address rS1+IMM16
LD.SZ.USR	rD,rS1,rS2	load from address rS1+rS2 or rS1+(rS2[scale]
	rD,rS1,(rS2)	Scale might be 0,1,2 or 3
LDA.SZ	rD,rS1,IMM16	load address
	rD,rS1,rS2	
	rD,rS1,(rS2)	
LDCR	rD,crS	load from control register
ST.SZ	rD,rS1,IMM16	store contents of rD in memory rS1+IMM16
ST.SZ.USR	rD,rS1,rS2	store in rS1+rS2 or rS1+(rS2[Scale]
	rD,rS1,(rS2)	
STCR	rD,crD	store to control register
XMEM.BU	rD,rS1,IMM16	exhange register with memory
XMEM.BU.USR	rD,rS1,rS2	
	rD,rS1,(rS2)	
XCR	rD,rS,crS/D	exhange control register

Table A.6: MC88100 Load/Store/Exchange Instructions

A.2 I80960 KB instruction set summary

Instruction	Operands	Name
LD	src,dst	load
LDOB	src,dst	load ordinal byte
LDOS	src,dst	load ordinal short
LDIB	src,dst	load integer byte
LDIS	src,dst	load integer short
LDL	src,dst	load long
LDT	src,dst	load triple
LDQ	src,dst	load quad
LDA	src,dst	load address
ST	src,dst	store
STOB	src,dst	store ordinal byte
STOS	src,dst	store ordinal short
STIB	src,dst	store integer byte
STIS	src,dst	store integer short
STL	src,dst	store long
STT	src,dst	store triple
STQ	src,dst	store quad

Table A.7: I80960KB Load/Store instructions

Instruction	Operands	Name
ADDI	src1,src2,dst	add integer
ADDO	src1,src2,dst	add ordinal
SUBI	src1,src2,dst	subtract integer
SUBO	src1,src2,dst	subtract ordinal
MULI	src1,src2,dst	multiply integer
MULO	src1.src2,dst	multiply ordinal
DIVI	src1,src2,dst	divide integer
DIVO	src1,src2,dst	divide ordinal
ADDC	src1,src2,dst	add ordinal with carry
SUBC	src1,src2,dst	subtract ordinal with carry
EMUL	src1,src2,dst	extended multiply
EDIV	src1,src2,dst	extended divide
REMI	src1,src2,dst	remainder integer
REMO	src1,src2,dst	remainder ordinal
MODI	src1,src2,dst	modulo integer

Table A.8: I80960KB Integer arithmetic instructions

Instruction	Operands	Name
MOV	src,dst	move
MOVL	src,dst	move long
MOVT	src,dst	move triple
MOVQ	src,dst	move quad

Table A.9: I80960KB Move instructions

Instruction	Operands	Name
SHLO	len,src,dst	shift left ordinal
SHRO	len,src,dst	shift right ordinal
SHLI	len,src,dst	shift left integer
SHRI	len,src,dst	shift right integer
SHRDI	len,src,dst	shift right dividing integer
AND	src1,src2,dst	A and B
ANDNOT	src1,src2,dst	A and (not B)
NOTAND	src1,src2,dst	(not A) and B
OR	src1,src2,dst	A or B
NOR	src1,src2,dst	(not A) and (not B)
XOR	src1,src2,dst	not (A=B)
XNOR	src1,src2,dst	A=B
NOT	src1,src2,dst	not A
NOTOR	src1,src2,dst	(not A) or B
ORNOT	src1,src2,dst	A or (not B)
NAND	src1,src2,dst	(not A) or (not B)

Table A.10: I80960KB Shift, rotate and logical instructions

Instruction	Operands	Name
CMPI	src1,src2	compare integer
CMPO	src1,src2	compare ordinal
CONCMPI	src1,src2	conditional compare integer
CONCMPO	src1,src2	conditional compare ordinal
CMPINCI	src1,src2,dst	compare and increment integer
CMPINCO	src1,src2,dst	compare and increment ordinal

Table A.11: I80960KB Compare, conditional compare instructions

Instruction	Operands	Name
B	targ	branch
BX	targ	branch extended
BAL	targ	branch and link
BALX	targ,dst	branch and link extended
BE	targ	branch if equal
BNE	targ	branch if not equal
BL	targ	branch if less
BLE	targ	branch if less than or equal
BG	targ	branch if greater
BGE	targ	branch if greater or equal
BO	targ	branch if ordered
BNO	targ	branch if unordered

Table A.12: I80960KB Branch instructions

Instruction	Operands	Name
CMPIBE	src1,src2,targ	compare integer, branch if equal
CMPIBNE	src1,src2,targ	compare integer, branch if not equal
CMPIBL	src1,src2,targ	compare integer, branch if not less
CMPIBLE	src1,src2,targ	compare integer, branch if not less or equal
CMPIBG	src1,src2,targ	compare integer, branch if greater
CMPIBGE	src1,src2,targ	compare integer, branch if greater
CMPIBO	src1,src2,targ	compare integer, branch if ordered
CMPIBNO	src1,src2,targ	compare integer, branch if unordered
CMPOBE	src1,src2,targ	compare ordinal, branch if equal
CMPOBNE	src1,src2,targ	compare ordinal, branch if not equal
CMPOBL	src1,src2,targ	compare ordinal, branch if not less
CMPOBLE	src1,src2,targ	compare ordinal, branch if not less or equal
CMPOBG	src1,src2,targ	compare ordinal, branch if greater
CMPOBGE	src1,src2,targ	compare ordinal, branch if greater
BBS	bitpos,src,targ	check bit, branch if set
BBC	bitpos,src,targ	check bit, branch if clear

Table A.13: I80960KB Compare and branch instructions

Instruction	Operands	Name
SETBIT	bitpos,src,dst	set bit
CLRBIT	bitpos,src,dst	clear bit
NOTBIT	bitpos,src,dst	not bit (bit toggle)
CHKBIT	bitpos,src	check bit
ALTERBIT	bitpos,src2,dst	alter bit
SCANBIT	src,dst	scan for bit
SPANBIT	src,dst	span over bit
EXTRACT	bitpos,len,src/dst	extract bits
MODIFY	mask,src,src/dst	modify bit

Table A.14: I80960KB Bit, bitfield instructions

Instruction	Operands	Name
CALL	targ	call a new precedure
CALLS	targ	call a system procedure
CALLX	targ	call extended
RET		return from procedure

Table A.15: I80960KB Call/return instructions

Instruction	Operands	Name
FAULTE		fault if equal
FAULTNE		fault if not equal
FAULTL		fault if less
FAULTLE		fault if less or equal
FAULTG		fault if greater
FAULTGE		fault if greater or equal
FAULTO		fault if ordered
FAULTNO		fault if unordered

Table A.16: I80960KB Conditional fault instructions

Instruction	Operands	Name
MODTC	mask,src,dst	modify trace controls
MARK		generate breakpoint trace-event
FMARK		force mark
MODPC	src,mask,src/dst	modify process controls
FLUSHREG		flush local registers
MODAC	mask,src,dst	modify arithmetic control
TESTE	dst	test for equal
TESTNE	dst	test for not equal
TESTL	dst	test for less
TESTLE	dst	test for less or equal
TESTG	dst	test for greater
TESTGE	dst	test for greater or equal
TESTO	dst	test for ordered
TESTNO	dst	test for unordered

Table A.17: I80960KB Processor management instructions

Instruction	Operands	Name
SYNCF		synchronize faults
SYNLD	src,dst	synchronize load
SYNMOV	dst,src	synchronous move
SYNMOVL	dst,src	synchronous move long
SYNMOVQ	dst,src	synchronous move quad

Table A.18: I80960KB Synchronous load and move instructions

Instruction	Operands	Name
ADDR	src1,src2,dst	add real
ADDL	src1,src2,dst	add long real
ATADD	src/dst,src,dst	atomic add
ATANR	src1,src2,dst	arctangent real
ATANRL	src1,src2,dst	arctangent long real
ATMOD	src,mask,src/dst	atomic modify
CLASSR	src	classify real
CLASSRL	src	classify long real
CMPOR	src1,src2	compare ordered real
CMPORL	src1,src2	compare ordered long real
CMPR	src1,src2	compare real
CMPRL	src1,src2	compare long real
COSR	src,dst	cosine real
COSRL	src,dst	cosine long real
CPYRSRE	src1,src2,dst	copy sign real extended
CPYSRE	src1,src2,dst	copy reversed sign real extended
CVTILR	src,dst	convert long integer to real
CVTIR	src,dst	convert integer to real
CVTRI	src,dst	convert real to integer
CVTRIL	src,dst	convert real to integer long
CVTZRI	src,dst	convert truncated real to integer
CVTZRIL	src,dst	convert truncated real to long integer
DIVR	src1,src2,dst	divide real
DIVRL	src1,src2,dst	divide long real
EXPR	src,dst	exponent real
EXPRL	src,dst	exponent long real
LOGBNR	src,dst	log binary real
LOGBNRL	src,dst	log binary long real
LOGEPR	src1,src2,dst	log epsilon real
LOGEPRL	src,1src2,dst	log epsilon long real
LOGR	src1,src2,dst	log real
LOGRL	src1,src2,dst	log long real
MOVR	src,dst	move real
MOVRL	src,dst	move long real
MOVRE	src,dst	move extended real
MULR	src1.src2,dst	multiply real
MULRL	src1.src2,dst	multiply long real
REMR	src1,src2,dst	remainder real
REMRL	src1,src2,dst	remainder long real
ROUNDR	src,dst	round real
ROUNDRL	src,dst	round long real
SCALER	src1,src2,dst	scale real
SCALERL	src1,src2,dst	scale long real

Table A.19: I80960KB Floating point instructions

Instruction	Operands	Name
SINR	src,dst	sine real
SINRL	src,dst	sine long real
SQRT	src,dst	square root real
SQRTRL	src,dst	square root long real
SUBQ	src1,src2,dst	subtract ordinal with carry
SUBR	src1,src2,dst	subtract real
SUBRL	src1,src2,dst	subtract long real
TANR	src,dst	tangent real
TANRL	src,dst	tangent long real

Table A.20: I80960KB Floating point instructions (continued)

Instruction	Operands	Name
DMOVT	src,dst	decimal move and test
DSUBC	src1,src2,dst	decimal subtract with carry
DADDC	src1,src2,dst	decimal add with carry

Table A.21: I80960KB Decimal arithmetic instructions

Instruction	Operands	Name
SCANBYTE	src1,src2	scan byte for equality
ROTATE	len,src,dst	rotate bits
CMPDECI	src1,src2,dst	compare and decrement integer
CMPPDECO	src1,src2,dst	compare and decrement ordinal

Table A.22: I80960KB Miscellanous instructions

A.3 Am29000 instruction set summary

Instruction	Operands	Comments
ADD	rc,ra,[rb—const8]	add
ADDS	rc,ra,[rb—const8]	signed add
ADDC	rc,ra,[rb—const8]	add with carry
ADDCS	rc,ra,[rb—const8]	signed add with carry
ADDCU	rc,ra,[rb—const8]	unsigned add with carry
SUB	rc,ra,[rb—const8]	subtract
SUBC	rc,ra,[rb—const8]	subtract with carry
SUBCS	rc,ra,[rb—const8]	subtract with carry, signed
SUBCU	rc,ra,[rb—const8]	subtract with carry, unsigned
SUBR	rs,ra,[rb—const8]	subtract reverse
SUBRC	rs,ra,[rb—const8]	subtract reverse with carry
SUBRCS	rs,ra,[rb—const8]	subtract reverse with carry, signed
SUBRCU	rs,ra,[rb—const8]	subtract reverse with carry, unsigned
SUBRS	rs,ra,[rb—const8]	subtract reverse signed
SUBRU	rs,ra,[rb—const8]	subtract reverse unsigned
SUBS	rs,ra,[rb—const8]	subtract signed
SUBU	rs,ra,[rb—const8]	subtract unsigned
MULTIPLU	rc,ra,rb	integer multiply unsigned
MULTIPLY	rc,ra,rb	integer multiply signed
MUL	rc,ra,[rb—const8]	multiply step
MULL	rc,ra,[rb—const8]	multiply last step
MULU	rc,ra,[rb—const8]	multiply step unsigned
DIV	rc,ra,[rb—const8]	divide step
DIVIDE	rc,ra,rb	integer divide, signed
DIVIDU	rc,ra,rb	integer divide, unsigned
DIV0	rc,[rb—const8]	divide initialize
DIVL	rc,ra,rb	divide last step
DIVREM	rc,ra,[rb—const8]	divide remainder

Table A.23: Am29000 Integer arithmetic instructions

Instruction	Operands	Comments
CPBYTE	rc,ra,[rb—const8]	compare bytes
CPEQ	rc,ra,[rb—const8]	compare equal to
CPGE	rc,ra,[rb—const8]	compare greater than or equal to
CPGEU	rc,ra,[rb—const8]	compare greater than or equal to,unsigned
CPGT	rc,ra,[rb—const8]	compare greater than
CPGTU	rc,ra,[rb—const8]	compare greater than, unsigned
CPLE	rc,ra,[rb—const8]	compare less than or equal to
CPLEU	rc,ra,[rb—const8]	compare less than or equal to, unsigned
CPLT	rc,ra,[rb—const8]	compare less than
CPLTU	rc,ra,[rb—const8]	compare less than, unsigned
CPNEQ	rc,ra,[rb—const8]	compare not equal to
ASEQ	vn,ra,[rb—const8]	assert equal to
ASGE	vn,ra,[rb—const8]	assert greater than or equal to
ASGEU	vn,ra,[rb—const8]	assert greater than or equal to, unsigned
ASGT	vn,ra,[rb—const8]	assert greater than
ASGT	vn,ra,[rb—const8]	assert greater than,unsigned
ASLE	vn,ra,[rb—const8]	assert less than or equal to
ASLEU	vn,ra,[rb—const8]	assert less than or equal to,unsigned
ASLT	vn,ra,[rb—const8]	assert less than
ASLTU	vn,ra,[rb—const8]	assert less than,unsigned
ASNEQ	vn,ra,[rb—const8]	assert not equal to

Table A.24: Am29000 Compare instructions

Instruction	Operands	Comments
AND	rc,ra,[rc—const8]	and logical
ANDN	rc,ra,[rb—const8]	and not logical
NAND	rc,ra,[rb—const8]	nand logical
NOR	rc,ra,[rb—const8]	nor logical
OR	rc,ra,[rb—const8]	or logical
XOR	rs,ra,[rb—const8]	exclusive or logical
XNOR	rs,ra,[rb—const8]	exclusive nor logical
SLL	rc,ra,[rb—const8]	shift left logical
SRA	rc,ra,[rb—const8]	shift right arithmetic
SRL	rc,ra,[rb—const8]	shift right logical
EXTRACT	rc,ra,[rb—const8]	extract word, bit-aligned

Table A.25: Am29000 Logical/shift instructions

Instruction	Operands	Comments
LOAD	ce,cntl,ra,[rb—const8]	load
LOADL	ce,cntl,ra,[rb—const8]	load and lock
LOADM	ce,cntl,ra,[rb—const8]	load multiple
LOADSET	ce,cntl,ra,[rb—const8]	load and set
STORE	ce,cntl,ra,[rb—const8]	store
STOREL	ce,cntl,ra,[rb—const8]	store and lock
STOREM	ce,cntl,ra,[rb—const8]	store multiple
EXBYTE	rc,ra,[rb—const8]	extract byte
EXHW	rc,ra,[rb—const8]	extract half-word
EXHWS	rc,ra	extract half-word, sign extended
INBYTE	rc,ra,[rb—const8]	insert byte
INHW	rc,ra,[rb—const8]	insert half word
MFSR	rc,spid	move from special register
MFTLB	rc,ra	move from translation look-aside buffer register
MTSR	spid,rb	move to special register
MTSRIM	spid,const16	move to special register immediate
MTTLB	ra,rb	move to translation look aside buffer register

Table A.26: Am29000 Data movement instructions

Instruction	Operands	Comments
CONST	ra,const16	constant
CONSTH	ra,const16	constant high
CONSTN	ra,const16	constant negative

Table A.27: Am29000 Constant instructions

Instruction	Operands	Comments
CALL	ra,target	call subroutine
CALLI	ra,rb	call subroutine, indirect
JMP	target	jump
JMPF	ra,target	jump false
JMPFDEC	ra,target	jump false and decrement
JMPFI	ra,rb	jump false indirect
JMPI	rb	jump indirect
JMPT	ra,target	jump true
JMPTI	ra,rb	jump true indirect

Table A.28: Am29000 Branch instructions

Instruction	Operands	Comments
DADD	rc,ra,rb	floating point add, double precision
DDIV	rc,ra,rb	floating point division, double precision
DEQ	rc,ra,rb	floating point equal to, double precision
DGE	rc,ra,rb	f.p greater than or equal to, d.p
DGE	rc,ra,rb	f.p greater than d.p
DMUL	rc,ra,rb	f.p multiply, d.p
DSUB	rc,ra,rb	f.p subtract, d.p
FADD	rc,ra,rb	f.p add, single precision
FDIV	rc,ra,rb	f.p divide, s.p
FEQ	rc,ra,rb	f.p equal to, s.p
FGE	rc,ra,rb	f.p greater than or equal to, s.p
FGT	rc,ra,rb	f.p greater than, s.p
FMUL	rc,ra,rb	f.p multiply, s.p
FSUB	rc,ra,rb	f.p subtract, s.p

Table A.29: Am29000 Floating-point instructions

Instruction	Operands	Comments
EMULATE	vn,ra,rb	trap to software emulation routine
HALT		enter halt mode
INV		invalidate
IRET		interrupt return
IRETINV		interrupt return and invalidate
SETIP	rc,ra,rb	set indirect pointers
CLZ	rc,[rb—const8]	count leading zeros
CONVERT	rc,ra,[conversion]	convert data format

Table A.30: Am29000 Miscellaneous instructions

A.4 R2000 instruction set summary

Instruction	Operands	Comments
LB	rt,offset(base)	load byte offset addr signed
LBU	rt,offset(base)	load byte offset addr unsigned
LH	rt,offset(base)	load halfword offset addr signed
LHU	rt,offset(base)	load halfword offset addr usigned
LW	rt,offset(base)	load word offset addr signed
LWCz	rt,offset(base)	load word to coprosessor
LWL	rt,offset(base)	load word left
LWR	rt,offset(base)	load word right
SB	rt,offset(base)	store byte
SH	rt,offset(base)	store halfword
SW	rt,offset(base)	store word
SWCz	rt,offset(base)	store word from coprocessor z
SWL	rt,offset(base)	store word left

Table A.31: R2000 Load/Store instructions

Instruction	Operands	Comments
ADD	rd,rs,rt	signed add,trap on overflow
ADDI	rt,rs,immediate	signed immediate add,trap on overflow
ADDIU	rt,rs,immediate	unsigned immediate add
ADDU	rd,rs,rt	unsigned add
SLT	rd,rs,rt	set on less than
SLTI	rt,rs,immediate	set on less than immediate
SLTIU	rt,rs,immediate	set on less than immediate unsigned
SLTU	rd,rs,rt	set on less than unsigned
AND	rd,rs,rt	logical and
ANDI	rt,rs,immediate	logical and immediate
LUI	rt,immediate	load upper word immediate
OR	rd,rs,rt	logical OR
ORI	rt,rs,immediate	logical OR immediate
XOR	rd,rs,rt	logical exclusive or
XORI	rt,rs,immediate	logical exclusive or immediate
SUB	rd,rs,rt	subtract
SUBU	rd,rs,rt	subtract unsigned
NOR	rd,rs,rt	logical NOR

Table A.32: R2000 Computational instructions

Instruction	Operands	Comments
SLL	rd,rt,amount	shift left logical
SLLV	rd,rt,rs	shift left logical variable
SRA	rd,rt,amount	shift right arithmetic
SRAV	rd,rt,rs	shift right arithmetic variable
SRL	rd,rt,amount	shift right logical
SRLV	rd,rt,rs	shift right logical variable

Table A.33: R2000 Shift instructions

Instruction	Operands	Comments
BCzF	offset	branch if false, coprocessor z condition is tested
BCzT	offset	branch if true, coprocessor z condition is tested
BEQ	rs,rt,offset	branch if equal
BGEZ	rs,offset	branch on greater than/equal to zero
BGEZAL	rs,offset	branch on greater than/equal to zero
BGTZ	rs,offset	branch on greater than zero
BLEZ	rs,offset	branch on less than/ equal to zero
BLTZ	rs,offset	branch on less than zero
BLTZAL	rs,offset	branch on less than/ equal to zero
BNE	rs,rt,offset	branch on not equal
BREAK		breakpoint trap
J	target	unconditional jump
JAL	target	unconditional jump and link
JALR	rs	jump and link register
JALR	rd,rs	jump and link register
JR	rs	jump register

Table A.34: R2000 Jump/branch instructions

Instruction	Operands	Comments
MULT	rs,rt	multiply
MULTU	rs,rt	unsigned multiply
DIV	rs,rt	signed divide
DIVU	rs,rt	unsigned divide
MFLO	rd	move from register LO
MFHI	rd	move from register HI
MTLO	rs	move to register LO
MTHI	rs	move to register HI

Table A.35: R2000 Multiply/divide instructions

Instruction	Operands	Comments
MFC0	rt,rd	move from system control coprocessor
MFCz	rt,rd	move from coprocessor z
MTC0	rt,rd	move to system control coprocessor
MTCz	rt,rd	move to coprocessor
RFE		restore from exeption
SYSCALL		system call
TLBP		probe TLB for matching entry
TLBR		read indexed TLB entry
TLBWI		write indexed TLB entry
TLBWR		write random TLB entry
CFCz	rt,rd	move control from coprocessor z
COPz	cofun	coprocessor operation
CTCz	rt,rd	move control to coprocessor z

Table A.36: R2000 Special/coprocessor instructions

A.5 SPARC CY7C601 instruction set summary

Instruction	Operands	Comments
ADD	*rs1,rs2/imm,rd*	integer add
ADDcc	*rs1,rs2/imm,rd*	integer add, modify icc
ADDX	*rs1,rs2/imm,rd*	integer add with carry
ADDXcc	*rs1,rs2/imm,rd*	integer add with carry, modify icc
TADDCC	*rs1,rs2/imm,rd*	tagged add and modify icc
TADDCCTV	*rs1,rs2/imm,rd*	tagged add, modify icc and trap on overflow
AND	*rs1,rs2/imm,rd*	logical and
ANDcc	*rs1,rs2/imm,rd*	logical and, modify icc
ANDN	*rs1,rs2/imm,rd*	logical and not
ANDNcc	*rs1,rs2/imm,rd*	logical and not, modify icc
SUB	*rs1,rs2/imm,rd*	subtract integer
SUBcc	*rs1,rs2/imm,rd*	subtract integer, modify icc
SUBX	*rs1,rs2/imm,rd*	subtract with carry
SUBXcc	*rs1,rs2/imm,rd*	subtract with carry, modify icc
TSUBCC	*rs1,rs2/imm,rd*	tagged subtract and modify icc
TSUBCCTV	*rs1,rs2/imm,rd*	tagged subtract, modify icc and trap on overflow
MULSCC	*rs1,rs2/imm,rd*	multiply step
OR	*rs1,rs2/imm,rd*	inclusive or
ORCC	*rs1,rs2/imm,rd*	inclusive or, modify icc
ORN	*rs1,rs2/imm,rd*	inclusive or not
ORNCC	*rs1,rs2/imm,rd*	inclusive or not, modify icc
XOR	*rs1,rs2/imm,tbr*	exclusive or
XORCC	*rs1,rs2/imm,tbr*	exclusive or and modify icc
XNOR	*rs1,rs2/imm,tbr*	exclusive nor
XNORCC	*rs1,rs2/imm,tbr*	exclusive nor and modify icc
SLL	*rs1,rs2/imm,rd*	shift left logical
SRL	*rs1,rs2/imm,rd*	shift right logical
SRA	*rs1,rs2/imm,rd*	shift right arithmetic
SETHI	*const,rd*	zero least sign 10 bits, replace high order bits

Table A.37: SPARC Arithmetic/Logical/Shift instructions

Instruction	Operands	Comments
LDSB	*[address],rd*	load signed byte
LDSBA	*[address]asi,rd*	load signed byte from alternate space
LDSH	*[address],rd*	load signed halfword
LDSHA	*[address]asi,rd*	load signed halfword from alternate space
LDUB	*[address],rd*	load unsigned byte
LDUBA	*[address]asi,rd*	load unsigned byte from alternate space
LDUH	*[address],rd*	load unsigned halfword
LDUHA	*[address]asi,rd*	load unsigned halfword from alternate space
LD	*[address],rd*	load word
LDA	*[address]asi,rd*	load word from alternate space
LDD	*[address],rd*	load doubleword
LDDA	*[address]asi,rd*	load doubleword from alternate space
LDF	*[address],frd*	load floating-point register
LDDF	*[address],frd*	load double floating-point register
LDFSR	*[address],fsr*	load floating-point state register
LDC	*[address],creg*	load coprocessor register
LDDC	*[address],creg*	load double coprocessor register
LDCSR	*[address],creg*	load coprocessor state register
LDSTUB	*[address],rd*	atomic load-store unsigned byte
LDSTUBA	*[address]asi,rd*	atomic load-store unsigned byte from alternate space
STB	*rd,[address]*	store byte
STBA	*rd,[address] asi*	store byte into alternate space
STH	*rd,[address]*	store halfword
STHA	*rd,[address] asi*	store halfword into alternate space
ST	*rd,[address]*	store word
STA	*rd,[address] asi*	store word into alternate space
STD	*rd,[address]*	store doubleword
STDA	*rd,[address] asi*	store doubleword into alternate space
STF	*frd,[address]*	store floating-point
STDF	*frd,[address]*	store double floating-point
STFSR	*fsr,[address]*	store floating-point state register
STDFQ	*fq,[address]*	store double floating-point queue
STC	*creg,[address]*	store coprocessor
STDC	*creg,[address]*	store double coprocessor
STCSR	*csr,[address]*	store coprocessor state register
STDCQ	*cq,[address]*	store double coprocessor queue
SWAP	*[source],rd*	swap register with memory
SWAPA	*[regsource]asi,rd*	swap register with alternate space memory

Table A.38: SPARC Load/Store instructions

Instruction	Operands	Comments
SAVE	*rs1,rs2/imm,rd*	save callers window
RESTORE	*rs1,rs2/imm,rd*	restore callers window
RETT	*address*	return from trap
BA	*label*	branch always
BN	*label*	branch never
BNE	*label*	branch on not equal
BE	*label*	branch on equal
BG	*label*	branch on greater
BLE	*label*	branch on less or equal
BGE	*label*	branch on greater or equal
BL	*label*	branch on less
BGU	*label*	branch on greater unsigned
BLEU	*label*	branch on less or equal unsigned
BCC	*label*	branch on carry clear
BCS	*label*	branch on carry set
BPOS	*label*	branch on positive
BNEG	*label*	branch on negative
BVC	*label*	branch on overflow clear
BVS	*label*	branch on overflow set
FBA	*label*	floating point branch always
FBN	*label*	floating point branch never
FBU	*label*	floating point branch on unordered
FBG	*label*	floating point branch on greater
FBUG	*label*	floating point branch on unordered or greater
FBL	*label*	floating point branch on less
FBUL	*label*	floating point branch on unordered or less
FBLG	*label*	floating point branch on less or greater
FBNE	*label*	floating point branch on not equal
FBE	*label*	floating point branch on equal
FBUE	*label*	floating point branch on unordered or equal
FBGE	*label*	floating point branch on greater or equal
FBUGE	*label*	floating point branch on unordered or greater or equal
FBLE	*label*	floating point branch on less or equal
FBULE	*label*	floating point branch on unordered or less or equal
FBO	*label*	floating point branch on unordered
CBA	*label*	branch always (on coprocessor condition)
CBN	*label*	branch never (on coprocessor condition)
CBx	*label*	branch on coprocessor x condition
CBxy	*label*	branch on coprocessor x or y condition
CBxyz	*label*	branch on coprocessor x or y or z condition
CALL	*label*	call subroutine
JMPL	*address,rd*	jump and link
TA	*address*	trap always
TN	*address*	trap never

Table A.39: SPARC Control Transfer instructions (continued)

Instruction	Operands	Comments
TNE	*address*	trap on not equal
TE	*address*	trap on equal
TG	*address*	trap on greater
TLE	*address*	trap on less or equal
TGE	*address*	trap on greater or equal
TL	*address*	trap on less
TGU	*address*	trap on greater unsigned
TLEU	*address*	trap on less or equal unsigned
TCC	*address*	trap on carry clear
TCS	*address*	trap on carry set
TPOS	*address*	trap on positive
TNEG	*address*	trap on negative
TVC	*address*	trap on overflow clear
TVS	*address*	trap on overflow set

Table A.40: SPARC Control Transfer instructions

Instruction	Operands	Comments
RDY	*y,rd*	read y register
RDPSR	*psr,rd*	read processor state register
RDWIM	*wim,rd*	read window invalid mask register
RDTBR	*tbr,rd*	read trap base register
WRY	*rs1,rs2/imm,y*	write y register
WRPSR	*rs1,rs2/imm,psr*	write processor state register
WRWIM	*rs1,rs2/imm,wim*	write window invalid mask register
WRTBR	*rs1,rs2/imm,tbr*	write trap base register

Table A.41: SPARC Read/Write control register operations

Instruction	Operands	Comments
CPop		coprocessor operations
FPop		coprocessor operations
UNIMP	*const22*	unimplemented instruction
IFLUSH	*address*	flush instruction cache

Table A.42: SPARC Miscellaneous instructions

A.6 T800 instruction set summary

Instruction	Operand	Comments
J	*adress*	jump
LDLP	*constant*	load local pointer
PFIX	prefix	
LDNL	*constant*	load non local
LDC	*constant*	load constant
LDNLP	*constant*	load non local pointer
NFIX	negative prefix	
LDL	*constant*	load local
ADC	*constant*	add constant
CALL	*adress*	call subroutine
CJ	*adress*	conditional jump
AJW	*constant*	adjust workspace
EQC	*constant*	equals constant
STL	*constant*	store local
STNL	*constant*	store non local
OPR	operate	

Table A.43: T800 Function codes

Instruction	Comments
AND	logical and
OR	logical or
XOR	logical xor
NOT	bitwise not
SHL	shift left
SHR	shift right
ADD	add
SUB	subtract
MUL	multiply
FMUL	fractional multiply
DIV	div
REM	remainder
GT	greater than
DIFF	difference
SUM	sum
PROD	product for positive(negative) register A

Table A.44: T800 Arithmetic/Logical operations

Instruction	Comments
LADD	long add
LSUB	long sub
LSUM	long sum
LDIFF	long diff
LMUL	long multiply
LDIV	long divide
LSHL	long shift left
LSHR	long shift right
NORM	normalise

Table A.45: T800 Long arithmetic operations

Instruction	Comments
REV	reverse
XWORD	extend to word
CWORD	check word
XDBLE	extend to double
CSNGL	check single
MINT	minimum integer
DUP	duplicate top of stack

Table A.46: T800 General operations

Instruction	Comments
MOVE2DINIT	initialise data for 2D block move
MOVE2DALL	2D block copy
MOVE2DNONZERO	2D block copy non-zero bytes
MOVE2DZERO	2D block copy zero bytes

Table A.47: T800 2D block move operations

Instruction	Comments
CRCWORD	calculate crc on word
CRCBYTE	calculate crc on byte
BITCNT	count bits set in word
BITREVWORD	reverse bits in word
BITREVNBITS	reverse bottom n bits in word

Table A.48: T800 CRC and bit operations

Instruction	Comments
BSUB	byte subscript
WSUB	word subscript
WSUBDB	word double word subscript
BCNT	byte count
WCNT	word count
LB	load byte
SB	store byte
MOVE	move message

Table A.49: T800 Indexing/array operations

Instruction	Comments
LDTIMER	load timer
TIN	timer input
TALT	timer alt start
TALTWT	timer alt wait
ENBT	enable timer
DIST	disable timer

Table A.50: T800 Timer handling operations

Instruction	Comments
IN	input message
OUT	output message
OUTWORD	output word
OUTBYTE	output byte
ALT	alt start
ALTWT	alt wait
ALTEND	alt end
ENBS	enable skip
DISS	disable skip
RESETCH	reset channel
ENBC	enable channel
DISC	disable channel

Table A.51: T800 Input/Output operations

Instruction	Comments
RET	return
LDPI	load pointer to instruction
GAJW	general adjust workspace
GCALL	general call
LEND	loop end

Table A.52: T800 Control operations

Instruction	Comments
STARTP	start process
ENDP	end process
RUNP	run process
LDPRI	load current priority

Table A.53: T800 Scheduling operations

Instruction	Comments
CSUB0	check subscript from 0
CCNT1	check count from 1
TESTERR	test error and clear
STOPERR	stop on error
SETERR	set error
CLRHALTERR	clear halt-on-error
SETHALTERR	set halt-on-error
TESTHALTERR	test halt-on-error

Table A.54: T800 Error handling operations

Instruction	Comments
TESTPRANAL	test processor analysing
SAVEH	save high priority registers
SAVEL	save low priority registers
STHF	store high priority front pointer
STHB	store high priority back pointer
STLF	store low priority front pointer
STLB	store low priority back pointer
STTIMER	store timer

Table A.55: T800 Processor initialisation operations

Instruction	Comments
FPLDNLSN	fp load non-local single
FPLDNLDB	fp load non-local double
FPLDNLSNI	fp load non-local indexed single
FPLDNLDBI	fp load non-local indexed double
FPLDZEROSN	fp load zero single
FPLDZERODB	fp load zero double
FPLDNLADDSN	fp load non-local and add single
FPLDNLADDDB	fp load non-local and add double
FPLDNLMULSN	fp load non-local and multiply single
FPLDNLMULDB	fp load non-local and multiply double
FPSTNLSN	fp store non-local single
FPSTNLDB	fp store non-local double
FPSTNLI32	fp store non-local int32

Table A.56: T800 Floating point Load/Store operations

Instruction	Comments
FPENTRY	floating point unit entry
FPREV	floating point reverse
FPDUP	floating point duplicate

Table A.57: T800 Floating point general operations

Instruction	Comments
FPURN	set rounding mode to round nearest
FPURZ	set rounding mode to round zero
FPURP	set rounding mode to round positive
FPURM	set rounding mode to round minus

Table A.58: T800 Floating point rounding operations

Instruction	Comments
FPCHKERROR	check fp error
FPTESTERROR	test fp error false and clear
FPUSETERROR	set fp error
FPUCLEARERROR	clear fp error

Table A.59: T800 Floating point error operations

Instruction	Comments
FPGT	fp greater than
FPEQ	fp equality
FPORDERED	fp orderability
FPNAN	fp not a number
FPNOTFINITE	fp not finite
FPUCHKI32	check in range of type int32
FPUCHKI64	check in range of type int64

Table A.60: T800 Floating point comparison operations

Instruction	Comments
FPUR32TOR64	real 32 to real 64
FPUR64TOR32	real 64 to real 32
FPRTOI32	real to int 32
FPI32TOR32	int 32 to real 32
FPI32TOR64	int 32 to real 64
FPB32TOR64	bit 32 to real 64
FPUNOROUND	real 64 to real 32, no round
FPINT	round to floating integer

Table A.61: T800 Floating point conversion operations

Instruction	Comments
FPADD	floating-point add
FPSUB	floating-point subtract
FPMUL	floating-point multiply
FPDIV	floating-point divide
FPUABS	floating-point absolute
FPREMFIRST	floating-point remainder first step
FPREMSTEP	floating-point remainder iteration
FPUSQRTFIRST	floating-point square root first step
FPUSQRTSTEP	floating-point square root step
FPUSQRTLAST	floating-point square root end
FPUEXPINC32	multiply by 2 EE 32
FPUEXPDEC32	divide by 2 EE 32
FPUMULBY2	multiply by 2
FPUDIVBY2	divide by 2

Table A.62: T800 Floating point arithmetic operations

A.7 THOR instruction set summary

Instruction	Operands	Comments
ADD	*expr*	add integer
ADDF	*expr*	add float
ADDI	*expr*	add immediate
ADDU	*expr*	add unsigned
DIV	*expr*	divide integer
DIVF	*expr*	divide float
MOD	*expr*	modulus
MUL	*expr*	multiply integer
MULF	*expr*	multiply float
MULI	*expr*	multiply immediatly
MULL	*expr*	multiply long
MULU	*expr*	multiply unsigned
SUB	*expr*	subtract
SBR	*expr*	subtract reversed
SUBF	*expr*	subtract float
SBRF	*expr*	subtract reversed float
SUBU	*expr*	subtract unsigned
SBRU	*expr*	subtract reversed unsigned
ABS		convert to absolute value
INT		convert float to integer
FLT		convert signed integer to float

Table A.63: THOR Arithmetic instructions

Instruction	Operands	Comments
PSH	*expr*	push value onto stack
PSHI	*expr*	push immediate
PSHR	*reg[,expr]*	push register
PSHX	*expr*	push indexed
POP	*expr*	pop value from stack
POPR	*reg[,expr]*	pop register
POPX	*expr*	pop indirect
LDX	*expr*	load indirect

Table A.64: THOR Move instructions

Instruction	Operands	Comments
AND	*expr*	logical and
ANDI	*expr*	logical and immediate
FBC		first bit changed
NOT		logical not
OR	*expr*	logical or
ORI	*expr*	logical or immediate
XOR	*expr*	logical exclusive or

Table A.65: THOR Logical instructions

Instruction	Operands	Comments
SL	*expr*	shift left
SLD	*expr*	shift left dynamic
SR	*expr*	shift right
SRA	*expr*	shift right arithmetic
SRAD	*expr*	shift right arithmetic dynamic
SRD	*expr*	shift right dynamic
SRDL	*expr*	shift right dynamic long

Table A.66: THOR Shift instructions

Instruction	Operands	Comments
CLL	*expr*	compare lower limit
CMP	*expr*	compare
CMPF	*expr*	compare float
CMPU	*expr*	compare unsigned
CUL	*expr*	compare upper limit

Table A.67: THOR Compare instructions

Instruction	Operands	Comments
CALL	*expr*	call subprogram
CALLP	*expr*	call protected
CLRF	*expr*	clear flags
FLUSH		flush cache
HLT		enter halt mode
JR	*expr*	jump relative
JREQ	*expr*	jump relative on equal
JRGE	*expr*	jump relative on greater than or equal
JRGT	*expr*	jump relative on greater than
JRLE	*expr*	jump relative on less than or equal
JRLT	*expr*	jump relative on less than
JRNE	*expr*	jump relative on not equal
JRX	*expr*	jump relative indirect
MTOS	*expr*	move top of stack
NOP		no operation
RET		return
RETU		return to user mode
SETF	*expr*	set flags
TEST	*expr*	test signed integer
RAISE		raise exception
TREG		change TCB
TA		task accept
TAE		task accept end
TAS		task accept start
TCA		task conditional accept
TCE	*expr*	task conditional entrycall
TDLY		task delay
TE	*expr*	task entrycall
TEE		task entrycall end
TPTR		task pointer
TSCH		task schedule

Table A.68: THOR Control instructions

Appendix B

Processor Context Switch

Figure B.1 describes the Process Control Block structure. The PCB:s search may be accomplished by the following (formal) scheme: (Figures within curly brackets denotes number of times each instruction are executed for a complete search).

```
; PCB search (generic) , exits with task identification
; number (T.ID) in r4,  task priority (T.PRI) in r3,
; ptr to highest process tasks PCB in r5
        move    PCB0PTR,r2        address of first PCB in r2 {1}
        move    r2,r5             ptr to hi priority task {1}
        move    10,r1             number of PCB:s to search {1}
        move    0,r3              initial priority (lowest) {1}
        move    0,r4              initial PCB ID (undefined) {1}
.L1:    cmp     (r2)T.PRI,r3      check PCB priority {10}
        jmple   .L2               branch if previous is greater {10}
        move    r2)T.PRI,r3       substitute new priority {1}
        move    (r2)T.ID,r4       remember task ID {1}
        move    r2,r5             remember PCB ptr {1}
.L2:    move    (r2)T.NEXT,r2      get next PCB pointer  {10}
        sub     1,r1              exit ... {10}
        cmp     0,r1              .. when .. {10}
        jmpne   .L1               .. all PCB:s searched {9}
```

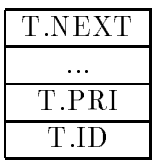

Figure B.1: Process Control Block structure

In the following paragraphs, the generic code will be translated to assembly code for the respective processors. The total amount of required machine cycles used to perform the

PCB search will be approximated. Register names are generalised to increase readability, thus the register naming conventions proposed by each manufacturer are not always used. It is assumed that "r0" is a "hard-wired-zero" register. It is further assumed that only one substitution of PCB is needed. Figures within curly brackets denotes the assumed number of processor cycles with respect to possible pipeline penalties. The code is not tested and not aimed for practical use.

The number of clock cycles required for storing/restoring processor context is estimated by considering a multiple store as well as a multiple load sequence. Since we are interested in the architectures impact only, we assume no wait state penalty from slow memory devices.

B.1 MC88100

B.1.1 PCB search

```
; PCB search, exits with task identification number (T.ID) in r4,
; task priority (T.PRI) in r3,
; ptr to highest process tasks PCB in r5
        lda.h   r2,r0,PCB0PTR   address of first PCB in r2 {1}
        add     r5,r0,r2        ptr to hi priority task  {1}
        add     r1,r0,10        number of PCB:s to search {1}
        add     r3,r0,0         initial priority (lowest) {1}
        add     r4,r0,0         initial PCB ID (undefined) {1}
.L1:    ld.b    r6,r2,T.PRI     priority to r6 (memory access) {40}
        cmp     r7,r3,r6        compare priorities, result in r7 {10}
        bb1     HS.BIT,r7,.L2   branch if previous is greater  {19}
        add     r3,r0,r6        substitute new priority  {1}
        lda.h   r4,r2,T.ID      remember task ID (memory access) {4}
        add     r5,r0,r5        remember PCB ptr  {1}
.L2:    lda.h   r2,r2,T.NEXT     get next PCB pointer (memory access) {40}
        sub     r1,r1,1         exit ...  {10}
        bcnd    gt0,r1,.L1      .. when all PCB:s searched {18}
```

B.1.2 Register Store

Figure B.2 outlines pipe-line occupation during multiple store. cycles 4-6 are memory data accesses that prevents instruction fetch, therefore MC88100 will finish 3 stores within every sixth cycle and so storing 31 registers will use (31*6/3) 62 cycles.

Register Restore

From figure B.3 we conclude: cycles 4-6 are memory data accesses that prevents instruction fetch, therefore MC88100 will finish 3 loads within every tenth cycle. During the last cycle, a prefetch of next instruction is possible, thus, loading 31 registers will be accomplished within ((31*9)/3)+1 cycles.

B.2 I80960KB

B.2.1 PCB search

Assuming Normal case execution time. Register "moves" are word sized.

```
; PCB search, exits with task identification number (T.ID) in r4,
; task priority (T.PRI) in r3,
; ptr to highest process tasks PCB in r5
        lda     PCB0PTR,r2   address of first PCB in r2  {1}
        move    r2,r5        ptr to hi priority task  {1}
        move    10,r1        number of PCB:s to search {1}
        move    0,r3         initial priority (lowest) {1}
        move    0,r4         initial PCB ID (undefined) {1}
.L1:    ldl     T.PRI(r2),r6 (memory access) {40}
# cmpibge has to wait for r6 ...
        cmpibge r3,r6,.L2    branch if previous is greater {30}
        move    r6,r3        substitute new priority {1}
        ldl     T.ID(r2),r4  remember task ID (memory access) {2}
        move    r2,r5        remember PCB ptr {1}
.L2:    ldl     T.NEXT(r2),r2 get next PCB pointer (memory access) {20}
        subo    r1,1,r1      exit ... {10}
        cmpobg  r1,r0,.L1    .. when all PCB:s searched {27}
```

B.2.2 Register Store

Cycles 4-6 (figure B.4) are memory data accesses that prevents instruction fetch, therefore I80960KB will finish 3 stores within every sixth cycle and so storing 80 registers will use (80*6)/3) 160 cycles.

B.2.3 Register Restore

Cycles 4-9 are memory data accesses that prevents instruction fetch, therefore I80960 will finish 3 loads within every tenth cycle. During the last cycle, a prefetch of next instruction

is possible, thus, loading 80 registers will be accomplished within ((79*9)/3)+1 cycles.

B.3 Am29000

B.3.1 PCB search

```
        ....
        ....
; PCB search, exits with task identification number (T.ID) in r4,
; task priority (T.PRI) in r3,
; ptr to highest process tasks PCB in r5
        const   r2,( PCB0PTR & 0xFFFF)  {1}
        consth  r2,( ( PCB0PTR >> 16 ) & 0xFFFF )  {1}
; load immediate into r2 done
        add     r5,r2,0      ptr to hi priority task {1}
        const   r1,10        number of PCB:s to search {1}
        const   r3,0         initial priority (lowest) {1}
        const   r4,0         initial PCB ID (undefined) {1}
.L1:    add     r7,r2,T.PRI  compute address of priority in r7 {10}
# feedforward, no penality for r7
        load    0,CNTL,r8,r7 get priority into r8(memory access) {30}
# wait for r8
        cplt    r9,r3,r8     compute boolean into r9 {10}
        jmpf    r9,.L2       branch if previous greater {2}
        nop                  always executed .. {10}
        add     r3,r8,0      remember new priority {1}
        add     r7,r2,T.ID   compute address of new task ID into r7 {1}
        load    0,CNTL,r4,r7 remember task ID (memory access) {1}
        add     r5,r2,0      remember PCB ptr {1}
.L2:    add     r7,r2,T.NEXT  compute address of next PCB ptr {10}
        load    0,CNTL,r2,r7 get next PCB pointer (memory access) {10}
        sub     r1,r1,1      one more ... {1}
        cpeq    r9,r1,0      compute boolean into r9 {10}
        jmpf    r9,.L1       continue until done {20}
        nop                  always executed {10}
        ....
        ....
```

B.3.2 Register Store/Restore

The "Load Multiple" and "Store Multiple" instructions allows the entire register file to be restored or saved in a single instruction. Thus loading as well as storing (192 registers) will be accomplished within 4+191 cycles.

B.4 MIPS R2000

B.4.1 PCB search

```
; PCB search, exits with task identification number (T.ID) in r4,
; task priority (T.PRI) in r3,
; ptr to highest process tasks PCB in r5
        lui     r2,(PCB0PTR >> 16 )      {1}
        ori     r2,r2,(PCB0PTR & 0x FFFF)     {1}
; load immediate into r2 done
        or      r5,r0,r2        copy into r5 {1}
        ori     r1,r0,9         number of PCB:s-1 to search  {1}
        ori     r3,r0,0         initial priority (lowest) {1}
        ori     r4,r0,0         initial PCB ID (undefined)  {1}
.L1:    lb      r8,T.PRI(r2)    priority (memory access) {10}
        nop                     delay slot {10}
        sltu    r9,r3,r8        compare priorities, result in r9 {10}
        nop                     delay slot {10}
        blez    r9,.L2          branch if previous is greater {10}
        nop                     delay slot {10}
        ori     r3,r8,0         substitute new priority  {1}
        lb      r4,T.ID(r2)     remember task ID (memory access) {1}
        ori     r5,r2,0         remember PCB ptr  {1}
.L2:    lhu     r6,T.NEXT(r2)   PCB pointer(high) (memory access) {10}
        lh      r7,T.NEXT+2(r2) PCB pointer(low) (memory access) {10}
        addi    r1,r1,-1                              {10}
        or      r2,r6,r7        move result into r2  {10}
        sltu    r9,r1,r0        compute bool into r9  {10}
        nop                     delay slot {10}
        blez    r9,.L1          exit when all PCB:s searched {9}
        nop                     (delayed branch) {9}
        ....
        ....
```

B.4.2 Register Store/Restore

Pipeline stalls while data is read from memory, or stored in memory (see figure B.6) since this prevents the processor from fetching the next instruction. Thus R2000 loads (or stores) 3 registers within 6 cycles which makes a total of 31*6/3 cycles.

B.5 SPARC

B.5.1 PCB search

```
        ....
        ....
; PCB search, exits with task identification number (T.ID) in r4,
; task priority (T.PRI) in r3,
; ptr to highest process tasks PCB in r5
        sethi   (PCB0PTR >> 10),r2
add r2,( PCBPTR & 0x3FF ),r2
; load immediate into r2 done ...
        add     r2,0,r5         ptr to hi priority task  {1}
        add     r0,10,r1        number of PCB:s to search {1}
        add     r0,0,r3         initial priority (lowest) {1}
        add     r0,0,r4         initial PCB ID (undefined) {1}
.L1:    ldub    r2+T.PRI,r6     r6 temp hold, priority (memory access) {1}
        sub     r6,r3,r7        compare priorities, result in r7 {1}
        ble,a   .L2             branch if previous is greater  {1}
        add     r0,r6,r3        substitute new priority  {1}
        ldub    r2+T.ID,r4      remember task ID (memory access) {1}
        add     r0,r2,r5        remember PCB ptr  {1}
.L2:    ld      r2+T.NEXT,r2     get next PCB pointer (memory access) {1}
        sub     r1,1,r1         exit ...  {1}
        bne,a   .L1             .. when all PCB:s searched {1}
        ....
        ....
```

B.5.2 Register Store/Restore

The SPARC pipeline is similar to the R2000 and the same pipeline stalls occurs (figure B.6. Thus loading as well as storing the entire SPARC register file will use 136*6/3 cycles.

B.6 T800 PCB search

For the T800 there is no need for a software process scheduler since there is hardware support for this in the processor. The T800 can run several processes concurrently. Processes may be assigned either high or low priority and there may be any number of each.

The processor has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. At any time, a concurrent process may be:

- *Active*
  - Being executed
  - On a list waiting to be executed
- *Inactive*
  - Ready to input
  - Ready to output
  - Waiting until a specified time

The scheduler operates in such a way that inactive processes do not consume any processor time. It allocates a portion of the processors time to each process in turn. Active processes waiting to be executed are held in two linked lists of process workspace, one of high priority processes and one of low priority processes. Each list is implemented using two registers, one of which points to the first process in the list, the other to the last.

Each process runs until it has completed its action, but is descheduled whilst waiting for communication from another process or transputer, or for a time to complete. In order for several processes to operate in parallel, a low priority process is only permitted to run for a maximum of two time slices before it is forcibly descheduled at the next descheduling point. The time slice period is approximately 1 ms.

A process can only be descheduled on certain instructions, known as descheduling points. As a result, en expression evaluation can be guarenteed to execute without the process being timesliced part way through.

Whenever a process is unable to proceed, its instruction pointer is saved in the process workspace and the next process taken from the list. Process scheduling pointers are updated by instructions which cause scheduling operations, and should not be altered directly. Actual process switch times are less than 1 micro second, as little state needs to be saved and its not necessary to save the evaluation stack on rescheduling.

The T800 supports two levels of priority. Priority 1 (low priority) processes are executed whenever there are no active priority 0 (high priority) processes. High priority processes are expected to execute for a short time. If one or more high priority processes are able to proceed, then one is selected and runs until it has to wait for a communication, a timer input or it completes processing. If no process at high priority is able to proceed, but one or more processes at low priority are able to proceed, then one is selected. If there are n low priority processes, then the maximum latency from the time at which a low priority process becomes active to the time when it starts processing is $2n$-2 timeslice periods. It is then able to execute for between one and two timeslice periods, less any time taken by high priority processes. This assumes that no process monopolises the transputer time; that is: has a distribution of descheduling points.

B.7 THOR PCB search

THOR , like the T800, facilitates hardware support for task switching. There are 6 different "Signal In" pins (SI0-SI5) which functionality equals ordinary interrupt signal lines. There are further four different SIGNAL OUT (SO0-SO3). Each SIGNAL IN is corresponding to a specific task, so that, when a SIGNAL IN occurs the hardware will ensure that the corresponding task will be scheduled next. This mechanism provides for a very rapid response to external events, and indeed supports multiprocessor configurations where different tasks may run in separate processors and the synchronisation between these tasks is accomplished throug the SIGNAL OUT and SIGNAL IN pins.

Fast software taskscheduling is accomplished by hardware. The chip include registers aimed to hold *task related data* i.e PCB. The mechanism insures that the highest priority process will be scheduled next. Priorities range between 1-32. It further insures that a delayed task receives immediate attention att the end of the delay. THOR, thus, do not need a software kernel to perform process scheduling.

Due to the stack architecture of THOR there are very little context to be saved and so it is reasonably to assume a process switch time below 1 microsecond.

Pipeline occupation cycle by cycle						
fetch 1	fetch 2	fetch 3	stall	stall	stall	fetch 4
	dec 1	dec 2	dec 3			
		exe 1	exe 2	exe 3		
			addr1	addr2	addr3	
			data1	data2	data3	

Figure B.2: MC88100 multiple store sequence

Pipeline occupation cycle by cycle									
fetch 1	fetch 2	fetch 3	stall	stall	stall	stall	stall	stall	fetch 4
	dec 1	dec 2	dec 3						
		exe 1	exe 2	exe 3					
			addr1		addr2		addr3		
				data1		data2		data3	
					writ1		writ2		writ3

Figure B.3: MC88100 multiple load sequence

Pipeline occupation cycle by cycle						
fetch 1	fetch 2	fetch 3	stall	stall	stall	fetch 4
	dec 1	dec 2	dec 3			
		exe 1	exe 2	exe 3		
			addr1	addr2	addr3	
			data1	data2	data3	

Figure B.4: I80960KB multiple store sequence

Pipeline occupation cycle by cycle									
fetch 1	fetch 2	fetch3	stall	stall	stall	stall	stall	stall	fetch 4
	dec 1	dec 2	dec 3						
		effadd1	effadd2	effadd3					
			addr1		addr2		addr3		
				data1		data2		data3	
					writ1		writ2		writ3

Figure B.5: I80960KB multiple load sequence

Pipeline occupation cycle by cycle						
fetch 1	fetch 2	fetch 3	stall	stall	stall	fetch 4
	dec 1	dec 2	dec 3			
		exe 1	exe 2	exe 3		
			write1	write2	write3	

Figure B.6: MIPS R2000 multiple load (store) sequence

Appendix C

Schematics

Figure C.1: T800 HDO-configuration

Figure C.2: THOR HDO-configuration

Figure C.3: SPARC HDO-configuration

Figure C.4: T800 and SPARC EDAC

Figure C.5: T800,THOR and SPARC memory

Figure C.6: T800 HSO-configuration

Figure C.7: THOR HSO-configuration

Figure C.8: SPARC HSO-configuration