Detailed Architecural Survey

Roger Johansson
Departement of Computer Engineering
Chalmers University of Technology
Göteborg

December 13, 1990

Abstract

During the past years the aspects of computer architecture has undergone rapid and revolutionary changes. Development of Reduced Instruction Set Computer (RISC) architectures has introduced a number of new concepts in computer architecture and design. This paper discuss RISC design methodologies in general and focusing on RISC processors in embedded Real-Time Systems. It is as a part of the ESTEC "RISC evaluation study".

Contents

Chapter 1

Introduction

This is a work on Reduced Instruction Set Computer design. It can, for short, be divided into three parts:

- A historical background, fundamentals of RISC.
- RISC processor study:
  - "Motorola MC 88100"
  - "Intel Iapx80960".
  - "Advanced Micro Devices Am 29000"
  - "MIPS R2000"
  - "Sun SPARC version 7"
  - "Inmos T800 transputer"
  - "Acorn Risc Machine (VL86C010)"
  - "Saab THOR"
- RISC processors behaviour in real-time environments.

A major purpose with this report is to provide an overview of the fundamentals in RISC design. Another important aim is to feature the RISC design concepts mainly by exemplifying with existent processor designs, discussing how application of different concepts may lead to quite different microprocessor designs.

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 to manage the central processing. One, for example, is the data-flow machine, another is the von Neumann- machine. A von Neumann-machine does information processing by sequentially executing algoritms which are ordered as programs and stored in a memory. The programs details interpretation and processing of information coded as data and stored in the same memory. The von Neumann-machine consists consequensly of at least a processor that sequentially interpret instructions in the program and a primary memory that store program and data. These architectures may degrade performance from the so called "von Neumann bottleneck" that is; 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 precense of data lacks 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 and so has been done. To distinguish between different von Neumann-architectures we more generally speak 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.

That is, the processors instruction set, its registers, and other details 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 instruction exhibits 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.

The Amdahl definition of computer architecture is not the only one but it serves well its purpose in this work.

1.2 Trends in computer architectures

To gain understanding about 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 has 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 execute. 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 has been added, the number of instructions has doubled, new functions has been added for instruction caches and coprocessors. Furthermore the instructions complexity has grown a tremendously.

The general trend towards modern CISC (Complex Instruction Set Computer) is a result of several factors. New models, within a computer family is got to be compatible with their predeccesors. As a result the number of functional units in the processor increases. By this new functions can be added in new machines without wasting of earlier software development efforts. Several efforts has been done in order 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 the tendency to be 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 instruction requires 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 lengthed. 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 encourage complex instructions. It is a structured way to implement, create and modify those algoritms that controls 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 exhibit the shortest execution times. This has been so due to the fact that assembly language programmers know the computer architecture well and is capable of taking every advantage of it. It is difficult to accomplish this in an automatic manner and for general cases as is required in compiler generated code. However, assembly language programming, as a way of increasing program performance lacks of 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 got 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 can't make use all of the sophisticated instructions that the architecture offers. Therefore, the compiler using simpler instructions, generating larger code making programs run slower, and wasting primary memory in a way that shouldn't 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 hypothesized that increased performance should be possible through a streamlined design and instruction set simplicity, hence a Reduced Instruction Set Computer.

A more analythical way to put these considerations may be as follows:

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 circiumstances. There must be at least three different ways to minimize 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 execute simultanously, 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, pipeline is a way to reduce C.

2. To keep T low requires 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. To minimize I requires complex instructions.

As we can see, there is no way of meeting all of these requirements at the same time. In fact, there is several contradictions in the requirements such as 1) and 3), 2) and 3), 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. Therefor, 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 "Reduce P = C T I, at constant cost" , described above can be considered as the generic RISC design concept. This besides other experiences has evolved in todays microprocessors. It may prove helpful to look at some consequenses of design decisions made on a basic RISC criteria. Figure 1.1 below is an attempt to sort things out. It is not a complete sketch but covers most of the RISC design consideration interactions that shows up in this paper.

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 execution were pipelined. Cray realized that all operations must be simplified for maximal perfomance. One bottleneck in processing may cause all other operations to degrade perfomance.

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

The first major university RISC research project was at the University of California, Berkeley. David Patterson, Carlos Sequin and a group of graduate students investigated the effective use of VLSI in microprocessor design. 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 also were investigated. This research resulted in a RISC processor for VLSI that is commonly referred to as "the Stanford MIPS".(Microprocessor without Interlocked Pipeline Stages).

Figure 1.1: A Risc Design Decision Graph

1.6 A brief overwiev of some RISC projects

The SOAR (Smalltalk On A RISC) project designed a computer to execute Smalltalk-80 faster than existing VLSI systems.

Berkeley SPUR (Symbolic Processing Using RISC) is a multiprocessor research machine for investigations in paralell processing.

Stanford MIPS-X is a successor to the Stanford MIPS design.

University of Wisconsin PIPE (Parallel Instructions and Pipelined Execution) project was an attempt to reduce three commonn processor bottlenecks with a reduced architecture. In the PIPE, programs are decomposed in separate address and computation tasks. Two independent identical processors peforms 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. The research group saw that the performance gains through concurrency have the potential beeing much more significant than performance gains throuh increased device speeds.

Ben-Gurion University MODHEL RISC study was an attempt to investigate the space between RISC and SIC (Single Instruction Computer) design to determine the optimal instruction mix.

Hewlett-Packard has developed a family of computers based upon RISC design. Two of these computers, the Series 930 and the Series 950 are implementations of the Precision Architecture.

IBM ROMP/MMU Processor represents one of the commercial spinoffs from the IBM 801 research project.

As one of the early commercial RISCs , the Pyramid 90x is a large, general purpose minicomputer intended to be competitive with VAX 11/780.

The Ridge 32s is one member of a family of 32-bit RISC workstations.

References:
32-Bit Microprocessors,
Mitchell, H J,
William Collins Sons & Co,1986

High Level Language computer architecture,
Milutinović (Editor),
Computer Science Press, 1989

RISC Architecture,
Tabak, Daniel,
Research Studies Press, 1987

Chapter 2

Description of RISC architectures

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

- Data types
- Instruction formats
- Addressing Modes
- CPU register description
- Instruction Set
- Processor states

The varying ways of implementing floating point support, memory management etc, will only be mentioned short and no detailed description will be given.

The main purpose with this chapter are:*Using common accessible information, give a standardised description of selected RISC processors architecture.*

The literature selected are throughly each manufacturers **Users Manual** (or equvivalent).

In some cases, descriptions are not of the standardised form that is desired. This may be due to the fact that *the information is not available from the Users Manual* or *Differences between the manuals do not allow a standardised description.* Finally, attempts to avoid repetitions may lead to discrepancies between the ways of describing two different processors.

In cases where none or insufficient information is available from the documentation, paragraphs may be entirely omitted or there might be a note about it.

2.1 The 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 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-extented. Data is placed in the least significant part while remaining bits is 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.

2.1.2 MC88100 instruction formats and 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 instructions references are made by the instruction unit. Data memory access instruction 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.

- Register to Register Instructions

  There are four addressing modes for register to register instructions.

  1. Triadic Register Addressing Mode
     Uses three five-bit fields to specify two source registers and a destination register. Some instructions do not use all three register selection fields. Unused fields should be zero. For arithmetic and logical instructions there is a subopcode field wich specifies the full operation
     Instruction Format (Floating Point):

```
31        26 25    21 20      16 15        5 4       0
1 0 0 0 0 1     D        S1     SUBOPCODE     S2
```

Instruction Format (Non Floating Point):

```
31        26 25    21 20      16 15        5 4       0
1 1 1 1 0 1     D        S1     SUBOPCODE     S2
```

D The D-field specifies the destination register.

S1 Specifies the source 1 operand register or zero.

SUBOPCODE Identifies the particular instruction.

S2 Specifies the source 2 operand register

2. **Register with 10-bit immediate addressing**

   This mode is used in bit-field instructions. The 10- bit immediate field serves as two 5-bit fields specifying the width and offset of the S1 operand field.

   Instruction Format:

```
31        26 25    21 20      16 15       10 9       0
1 1 1 1 0 0    D         S1     SUBOPCODE   IMM10
```

   D Specifies destination register

   S1 Specifies source 1 operand register

   SUBOPCODE identifies the particular instruction

   IMM10 Contains a 10-bit value where: bits 9-5 is a 5-bit width bits 4-0 is a 5-bit offset

3. **Register with 16-bit immediate addressing**

   This form is used by arithmetic and logical instructions requiring a 16-bit immediate source value.

   Instruction Format:

```
31        26 25    21 20      16 15                  0
 OPCODE        D        S1            IMM16
```

   OPCODE identifies the particular instruction

   D Specifies destination register

   S1 Specifies source 1 operand register

   IMM16 contains an unsigned immediate value.

4. **Control Register Addressing**

   This mode is used to reference the general control and FPU control registers. It applies to user as well as supervisor programming modes.

   Instruction Format:

```
31        26 25    21 20      16 15 14 13  11 10      5 4   0
1 0 0 0 0 0    D        S1        OP    SFU   CRS/CRD    S2
```

   D Specifies destination register

   S1 In case of store/exhange instructions this field specifies the source register. For load-instructions this field is ignored.

   OP identifies the particular instruction

SFU specifies the special function unit accessed by the instruction.

CRS/CRD specifies the control register, might be source as well as destination depending on instruction

S2 must contain the same value as the S1 field and serves the same purposes.

- Data Memory Access Instructions

1. Register Indirect with zero-extended immediate index

   The contents of rS1 are added to the 16-bit zero- extended immediate index contained in the I16 field of the instruction. The result is a data memory address used to load or store data. For a load instruction the data is loaded into the register specified by the D field. For a store or a exhange instruction the data in the register specified by the D field is stored to memory.

   Instruction Format:

```
31       26 25    21 20     16 15                        0
  OPCODE       D        S1                I16
```

   OPCODE identifies the particular instruction

   D Specifies destination register

   S1 specifies the source 1 operand register usd in the address calculation.

   I16 contains a 16-bit index.

2. **Register indirect with index**

   The contents of rS1 is added to the contents of rS2 resulting in a data memory address used to load or store data.

   Instruction Format:

```
31       26 25    21 20     16 15             5 4        0
1 1 1 1 0 1    D        S1         SUBOPCODE      S2
```

   D Specifies destination register

   S1 specifies the source 1 operand register usd in the address calculation.

   SUBOPCODE identifies the particular instruction

   S2 specifies the source 2 operand.

3. **Register indirect with scaled index**

   The contents of rS2 is scaled by the size of the access and then added to the contents of rS1 resulting in a data memory address used to load or store data.

   Instruction Format:

```
31       26 25    21 20     16 15             5 4        0
1 1 1 1 0 1    D        S1         SUBOPCODE      S2
```

   D Specifies destination register

   S1 specifies the source 1 operand register usd in the address calculation.

   SUBOPCODE identifies the particular instruction including the scaling factor.

   S2 specifies the source 2 operand.

- Flow Control Instructions

1. Triadic Register Addressing

   This form is used to specify the target of a jump instruction or the operands of a trap-on-bound instruction.

   Instruction Format:

```
31        26 25    21 20     16 15            5 4       0
0 1 1 1 1 0 1    D         S1         SUBOPCODE       S2
```

   D This field is ignored

   S1 This field is ignored

   SUBOPCODE identifies the particular instruction

   S2 specifies the source 2 register

2. **Trap generating bounds check instruction (tbnd)**

   The data in rS1 and rS2 is compared and a trap is taken if the source 1 data is greater than the source 2 data (unsigned). If the trap is taken execution transfers to the bounds check exception as follows: the 20-bit address in the VBR is concatenated with the bounds check exception vector and three trailing zeros to form the 30-bit instruction address. The result is placed in the FIP, and program execution begins from that address.

   Instruction Format:

```
31        26 25    21 20     16 15            5 4        0
1 1 1 1 0 1    D         S1         SUBOPCODE       S2
```

   D This field is ignored

   S1 This field specifies the source 1 operand register

   SUBOPCODE identifies the particular instruction (tbnd).

   S2 specifies the source 2 register

3. **Register with 9-bit vector table index**

   For bit test trap instructions (tb0, tb1 and tcnb) the bit in rS1 specified by the B5 field is tested for either a set or clear condition. For conditional trap instructions, the source 1 register is tested for the condition(s) specified in the M5 field. In either case, if the test condition is true, the 20-bit address in the in the VBR is concatenated with the VEC9 field of the instruction and three trailing zeros to form the 30-bit instruction address. Exception processing begins,and the vector is fetched from the resulting address.

   Instruction Format:

```
31        26 25    21 20     16 15            9 8        0
1 1 1 1 0 0   B5/M5      S1         SUBOPCODE       VEC9
```

   B5/M5 For bit-test the B5 field specifies the bit to be tested in the register specified by the S1 field. For conditional tests the M5 field specify which conditions to test out of four possible conditions.

   S1 This field specifies the source 1 operand register

   SUBOPCODE identifies the particular instruction (tb0,tb1 and tcnb)

   S2 specifies the source 2 register

   VEC9 contains the 9-bit vector numbert

4. Register with 16-bit displacement/immediate

   This form is used by branch and trap instructions for target address and test condition generation. For bit-test branch instructions, the bit in rS1 specified by the B5 field is tested for either a set or clear condition. For condition-test branch instructiuons, rS1 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 is shifted left two positions and sign-extended to 32 bits. The two least significant bits are cleared to force word alignement. This value is added to the execute instruction pointer (XIP), and the result is loaded into the FIP. Program execution is transferred to that address.

   Instruction Format:

```
31        26 25    21 20     16 15                         0
   OPCODE     B5/M5      S1                D16
```

   OPCODE this field identifies the particular instruction (bb0 ,bb0.n ,bb1,bb1.n ,bcnd, bcnd.n).

   B5/M5 For bit-test the B5 field specifies the bit to be tested in the register specified by the S1 field. For conditional tests the M5 field specify which conditions to test out of four possible conditions.

   S1 This field specifies the source 1 operand register

   D16 specifies a signed 16-bit displacement.

5. Trap generating Bounds Check Instruction(tbnd)

   The data in rS1 is compared to the specified immediate operand, and a trap is taken if the register data is greater than the immediate operand (unsigned). If the trap is taken,the bounds check vector number is combined with the VBR, and the result is concatenated with three trailing zeros and loaded into the FIP. Exception processing begins for the bounds check exception.

   Instruction Format:

```
31        26 25    21 20     16 15                         0
   OPCODE       D        S1              IMM16
```

   OPCODE this field identifies the particular instruction(bb0,bb0.n,bb1,bb1.n,bcnd,bcnd.

   D This field is not used.

   S1 This field specifies the source 1 operand register

   IMM16 specifies a 16-bit immediate operand for the tbnd-instruction.

6. 26-bit branch displacement

   This form is used to specify the branch target instruction in unconditional branch instructions (br,bsr). These instructions 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.

   Instruction Format:

```
31      26 25                                       0
   OPCODE                    D26
```

OPCODE This field identifies the particular instruction (br,br.n,bsr,bsr.n).
D26 This field specifies the displacement to the target instruction.

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. They are essential for understanding of exeption processing, delayed branches etc. The following will briefly describe these registers.

1. General Purpose registers

   r0-r31 contain program data. All of these registers with the exeption of r0 (constant zero) has read/write access. A write operation to r0 has no effect.

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

2. Floating-point operation registers

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

3. Control Registers

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

4. Internal Registers

   - **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. This will be further discussed below.**

2.1.4 MC88100 instruction set

The assembler mnemonics may have extensions with the following interpretations:

- .u denotes upper half word
- .c second operand is ones complemented before it is used in the operation
- .n delayed branch option
- .car (carry) may be
  1. .ci carry in, include PSR carry in operation
  2. .co carry out, affect PSR carry by operation
  3. .cio both .ci and .co operations

- .sz size (default is ”word”) may be
  1. .b byte
  2. .bu unsigned byte
  3. .h half word
  4. .hu unsigned half word
  5. .s single word (32 bits)
  6. .d double word (64 bits)
- .fsz floating point operand size
  1. .s single precision
  2. .d double precision

  Floating point operations support mixed operand sizes.
- .usr user memory option allows user memory to be accessed while in the supervisor mode

Instruction/ Options	Operands	Comments
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
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	
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	
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
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
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

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. This levels define 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 signaled
- 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 signaled 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 88000 Pipelining

There are four separate execution units which allow MC88000 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 one 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 paralellism. Instruction decode and source operand fetches from the registers are performed simultanously. Branch instruction decode and branch target address calculation are performed in paralell with the next instruction fetch. Three internal register buses allow three simultanously register accesses.

References:
MC88100 Risc Processor User's Manual,Second Edition
Prentice Hall, 1990

2.2 Intel 80960KB

The 80960 is a 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. 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.

2.2.1 The 80960 data formats

The 80960 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 represents grouping of bits or bytes that the processor can operate on as 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.

2.2.2 80960 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 instructions 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 is for operations that are performed on data contained in the global, local or floating point registers.

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

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 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 is used primarily for control-and- branch-instructions. The opcode field is 8 bits.

```
31     24 23      19 18   14 13 12                2 1 0
  OPCODE     SRC1     SRC2  M1      DISPLACEMENT    0 0
```

  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, twos 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 comparision. 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 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.

```
31     24 23                                      2 1 0
  OPCODE             DISPLACEMENT                   0 0
```

  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.

- MEM format

  The MEM(A or B) 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

     ```
     31      24 23       19 18    14 13 12            0
       OPCODE   SRC/DST   ABASE  MD  0    OFFSET
     ```

     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 loade 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. Using 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.

     ```
     31      24 23         19 18    14 13   10 9      7 6 5 4      0
       OPCODE     SRC/DST   ABASE   MODE   SCALE  0 0  INDEX
     ```

     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, twos complement value.

2.2.3 80960 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 instructionsfor 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.

2.2.4 80960 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 do not use registers in that group, the processor, in some instances is 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 not is known.

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.

2.2.5 80960 Instruction Set

Instruction	Operands	Options
ADDC	src1,src2,dst	add ordinal with carry
ADDI	src1,src2,dst	add integer
ADDO	src1,src2,dst	add ordinal
ADDR	src1,src2,dst	add real
ADDL	src1,src2,dst	add long real
ALTERBIT	bitpos,src2,dst	alter bit
AND	src1,src2,dst	logical and
ANDNOT	src1,src2,dst	logical and, src1 inverted
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
BAL	targ	branch and link
BALX	targ,dst	branch and link extended
B	targ	branch
BX	targ	branch extended
BBC	bitpos,src,targ	check bit, branch if clear
BBS	bitpos,src,targ	check bit, branch if set
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
CALL	targ	call a new precedure
CALLS	targ	call a system procedure
CALLX	targ	call extended
CHKBIT	bitpos,src	check bit
CLASSR	src	classify real
CLASSRL	src	classify long real
CLRBIT	bitpos,src,dst	clear bit
CMPI	src1,src2	compare integer
CMPO	src1,src2	compare ordinal
CMPDECI	src1,src2,dst	compare and decrement integer
CMPPDECO	src1,src2,dst	compare and decrement ordinal
CMPINCI	src1,src2,dst	compare and increment integer
CMPINCO	src1,src2,dst	compare and increment ordinal
CMPOR	src1,src2	compare ordered real
CMPORL	src1,src2	compare ordered long real
CMPR	src1,src2	compare real
CMPRL	src1,src2	compare long real
COMPIBE	src1,src2,targ	compare integer, branch if equal
COMPIBNE	src1,src2,targ	compare integer, branch if not equal
COMPIBL	src1,src2,targ	compare integer, branch if not less
COMPIBLE	src1,src2,targ	compare integer, branch if not less or equal
COMPIBG	src1,src2,targ	compare integer, branch if greater
COMPIBGE	src1,src2,targ	compare integer, branch if greater
COMPIBO	src1,src2,targ	compare integer, branch if ordered
COMPIBNO	src1,src2,targ	compare integer, branch if unordered
COMPOBE	src1,src2,targ	compare ordinal, branch if equal
COMPOBNE	src1,src2,targ	compare ordinal, branch if not equal
COMPOBL	src1,src2,targ	compare ordinal, branch if not less
COMPOBLE	src1,src2,targ	compare ordinal, branch if not less or equal
COMPOBG	src1,src2,targ	compare ordinal, branch if greater
COMPOBGE	src1,src2,targ	compare ordinal, branch if greater
CONCMPI	src1,src2	conditional compare integer
CONCMPO	src1,src2	conditional compare ordinal
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
DADDC	src1,src2,dst	decimal add with carry
DIVI	src1,src2,dst	divide integer

DIVO	src1,src2,dst	divide ordinal
DIVR	src1,src2,dst	divide real
DIVRL	src1,src2,dst	divide long real
DMOVT	src,dst	decimal move and test
DSUBC	src1,src2,dst	decimal subtract with carry
EDIV	src1,src2,dst	extended divide
EMUL	src1,src2,dst	extended multiply
EXPR	src,dst	exponent real
EXPRL	src,dst	exponent long real
EXTRACT	bitpos,len,src/dst	extract bits
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
FLUSHREG		flush local registers
FMARK		force mark
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
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
MARK		generate breakpoint trace-event
MODAC	mask,src,dst	modify arithmetic control
MODI	src1,src2,dst	modulo integer
MODIFY	mask,src,src/dst	modify bit
MODPC	src,mask,src/dst	modify process controls
MODTC	mask,src,dst	modify trace controls
MOVE	src,dst	move
MOVL	src,dst	move long
MOVT	src,dst	move triple
MOVQ	src,dst	move quad
MOVR	src,dst	move real
MOVRL	src,dst	move long real
MOVRE	src,dst	move extended real
MULI	src1,src2,dst	multiply integer
MULO	src1.src2,dst	multiply ordinal

MULR	src1.src2,dst	multiply real
MULRL	src1.src2,dst	multiply long real
NAND	src1,src2,dst	bitwise nand
NOR	src1,src2,dst	bitwise or
NOT	src1,src2,dst	bitwise not
NOTAND	src1,src2,dst	bitwise notand
NOTBIT	bitpos,src,dst	not bit (bit toggle)
NOTOR	src1,src2,dst	not or
OR	src1,src2,dst	logical or
ORNOT	src1,src2,dst	logical or complemented
REMI	src1,src2,dst	remainder integer
REMO	src1,src2,dst	remainder ordinal
REMR	src1,src2,dst	remainder real
REMRL	src1,src2,dst	remainder long real
RET		return from procedure
ROTATE	len,src,dst	rotate bits
ROUNDR	src,dst	round real
ROUNDRL	src,dst	round long real
SCALER	src1,src2,dst	scale real
SCALERL	src1,src2,dst	scale long real
SCANBIT	src,dst	scan for bit
SCANBYTE	src1,src2	scan byte equal
SETBIT	bitpos,src,dst	set bit

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
SINR	src,dst	sine real
SINRL	src,dst	sine long real
SPANBIT	src,dst	span over bit
SQRT	src,dst	square root real
SQRTRL	src,dst	square root long real
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
SUBQ	src1,src2,dst	subtract ordinal with carry
SUBI	src1,src2,dst	subtract integer
SUBO	src1,src2,dst	subtract ordinal
SUBR	src1,src2,dst	subtract real
SUBRL	src1,src2,dst	subtract long real
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
TANR	src,dst	tangent real
TANRL	src,dst	tangent long real
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
XNOR	src1,src2,dst	exclusive nor
XOR	src1,src2,dst	exclusive or

2.2.6 i80960 processor states

The i80960 recognizes two different kinds of events which may be considered as disturbance of normal instruction execution. Thus the processor may be in on of three states:

- Normal Instruction Execution
- Interrupts
- Faults

Interrupts

An interrupt is a temporary break in the control stream of a program so that the processor can handle a different chore. Interrupts are generally requested from an external source. The interrupt request either contains a vector number or else, points to a vector that tells the processor what chore to do while in the interrupted state. When the processor has finished servicing the interrupt it generally returns to the program that was interrupted and resumes execution where it left off.

The processor provides a mechanism for servicing interrupts, which uses an implicit procedure call to a selected interrupt handling procedure called the interrupt handler. To use the processors interrupt handling facilities, software must provide the following items in memory

- Interrupt Table
- Interrupt Handler Routines
- Interrupt Stack

These items are generally established in memory as part of the initialization procedure. Once these items are present in memory and pointers to them have been entered in the appropriate system data sttructures the processor then handles interrupts automatically and independently from software.

Each interrupt vector is 8 bits in length which allows up to 256 unique vectors to be defined. Vectors 0-7 cannot be used and vectors 244 thru 251 are reserved. Each vector has a predefined priority, which is defined by the expression: PRIORITY = VECTOR/8. The processor uses the priority to determine whether or not to service the interrupt immediatly or to delay service. A priority 31 is always serviced immediatly. The lowest program priority allowed is 0, thats why vectors 0-7 cannot be used since interrupts with these prioritys never would be accepted.

Fault Handling

The processor is able to detect various conditions in code or its internal state called "fault conditions". Such conditions could cause the processor to deliever incorrect or inappropriate results or could cause it to follow an undesirable control path. A fault may be detected while the processor is executing a program, an interrupt handler or a fault handler. When the procesor detects a fault it handles the fault immediatly and independently of the program or handler it is currently working on, using a mechanism similar to that used to service interrupts.

All of the faults that the processor detects are predefined. These faults are divided into types and subtypes, each of which is given a number. The processor uses the type number to select a fault handler. The fault handler then uses the subtype number to select a specific fault-handling procedure.

It is possible for multiple fault conditions to occur simultaneously. For certain fault types, bit positions in the fault-subtype field are used to indicate the occurence of multiple faults of the same type. As a general rule, however, the processor does not indicate situations where multiple faults occur. Instead it records one of the faults and does not report on the faults that were not recorded. Faults occuring while the processor executng a fault handling routine causes unpredictable operation by the processor.

Faults are grouped into the categories precise, imprecise and asynchronous. Precise faults are those tat are intended to be recoverable by software. Imprecise faults are those that in some instances are allowed to occur and not be signaled or be signaled out of order. Asynchronous faults are those whose occurence has no direct relationship to the instruction pointer.

2.2.7 i80960 Pipelining

Pipelined execution is accomodated by the use of separate control units:

- Bus Control Logic *BCL*
- Instruction Fetch Unit with instruction cache *IFU*
- Instruction Decoder *ID*
- Microinstruction sequencer *MIS*
- Instruction Execution Unit *IEU*

The Instruction Fetch Unit acts as a buffer for the Instruction Decoder . The IFU maintains a 512 byte, direct-mapped instruction cache. While other units in the processor are executing instructions the IFU looks ahead in instruction flow stored in the cache. If a cache miss is detected the IFU issues a prefetch request to the BCL. The IFU works closely with the ID in handling branch conditions. The ID informs the IFU of any branch operation that is to take place. When the IFU is notified of a branch, it checks for a cache hit on the desired instruction. If the instruction is not present, the IFU begins fetching instructions for the new control path. The ID may send a special signal to the IFU whenever instructions are required immediatly. The IFU then passes the fetched instructions to the ID directly, rather than writing them to the cache and reading them back again. This technique is called instruction-cache- bypassing.

The ID decodes the instructions it receives from the IFU and routes them to the appropriate execution unit. Instructions are decoded into four groups, according to how the instructions are executed:

- Simple Instructions
- Floating Point and branch instructions
- Complex instructions
- Load/Store Instructions

The MIS is a multipurpose unit designed to help in the execution of instructions that use microcode. When the ID receives a complex instruction, the MIS supplies the microcode to the IEU. The MIS also supplies microcode for floating-point instructions the power-up and self-test performed during processor initialisation; interrupt handling and fault handling.

The IEU contains the Arithmetic Logic Unit ALU and the mechanisms for register and condition-code scoreboarding. The IEU handles the reading and writing of global registers. It also handles the allocation of local register sets on each procedure calls. The IEU allocates a new set of registers on each procedure call. If all four register sets become allocated, the IEU automatically flushes the oldest frame to the stack on the next procedure call. The IEU also automatically retrieves any local register frame from the stack when required by a return operation.

The register scoreboard provides scoreboarding for the global and the local registers. When, one or more registers are being used in an operation they are marked as in use. The mechanism allows the processor to continue executing subsequent instructions as long as those instructions do not require the contents of the scoreboard registers.

References:
80960KB Programmer's Manual,
Intel Corporation, 1988

2.3 The 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 has 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 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 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 respectivly, in the most significant bit of a word.

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

2.3.2 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.

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

- *BITS 31-22*

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

A 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.

M 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

- *BITS 21-16*
  - 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..I18 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
- *BITS 15-8*
  - RA the RA-field contains a global or local register number
  - SA the SA-field contains a special register number
- *BITS 7-0*
  - 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.

2.3.3 Am29000 register description

The Am29000 has three classes of registers which are accesible 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.

The Am29000 incorporates 192 general-purpose registers organised as follows:

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

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 accsesses. 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 protected special purpose registers are defined as follows:

   The Special Purpose Registers are organised as follows:

Special Purpose Register Number	PROTECTED REGISTER
0	Vector Base Address
1	Old Processor Status
2	Current Processor Status

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

VECTOR AREA BASE 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 superviseor 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.

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 provideing information on the least recently used entry of the TLB when a TLB miss occurs.

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 set

Assembler syntax in a short form:
ce, determines the coprocessor enable (CE) bit of a load or store instruction.
cntl, determines the 7-bit control field in a load or store operation.
const8, specifies a constant which can be expressed by 8 bits.
const16, specifies a constant which can be expressed by 16 bits.
ra,rb,rc, represents general purpose registers.
spid, represents a special purpose register
target, is a symbolic label for a jump or call instruction.
vn, specifies a trap vector number.

Instruction	Operands	Comments
ADD	rc,ra,[rb—const8]	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
AND	rc,ra,[rc—const8]	and logical
ANDN	rc,ra,[rb—const8]	and not logical
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
CALL	ra,target	call subroutine
CALLI	ra,rb	call subroutine, indirect
CLZ	rc,[rb—const8]	count leading zeros
CONST	ra,const16	constant
CONSTH	ra,const16	constant high
CONSTN	ra,const16	constant negative
CONVERT	rc,ra,[conversion]	convert data format
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
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
DIV	rc,ra,[rb—const8]	divide step
DIV0	rc,[rb—const8]	divide initialize
DIVIDE	rc,ra,rb	integer divide, signed
DIVIDU	rc,ra,rb	integer divide, unsigned
DIVL	rc,ra,rb	divide last step
DIVREM	rc,ra,[rb—const8]	divide remainder
DMUL	rc,ra,rb	f.p multiply, d.p
DSUB	rc,ra,rb	f.p subtract, d.p
EMULATE	vn,ra,rb	trap to software emulation routine
EXBYTE	rc,ra,[rb—const8]	extract byte
EXHW	rc,ra,[rb—const8]	extract half-word
EXHWS	rc,ra	extract half-word, sign extended
EXTRACT	rc,ra,[rb—const8]	extract word, bit-aligned
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
HALT		enter halt mode
INBYTE	rc,ra,[rb—const8]	insert byte
INHW	rc,ra,[rb—const8]	insert half word
INV		invalidate
IRET		interrupt return
IRETINV		interrupt return and invalidate
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
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
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
MUL	rc,ra,[rb—const8]	multiply step
MULL	rc,ra,[rb—const8]	multiply last step
MULTIPLU	rc,ra,rb	integer multiply unsigned
MULTIPLY	rc,ra,rb	integer multiply signed
MULU	rc,ra,[rb—const8]	multiply step unsigned
NAND	rc,ra,[rb—const8]	nand logical
NOR	rc,ra,[rb—const8]	nor logical
OR	rc,ra,[rb—const8]	or logical
SETIP	rc,ra,rb	set indirect pointers
SLL	rc,ra,[rb—const8]	shift left logical
SRA	rc,ra,[rb—const8]	shift right arithmetic
SRL	rc,ra,[rb—const8]	shift right logical
STORE	ce,cntl,ra,[rb—const8]	store
STOREL	ce,cntl,ra,[rb—const8]	store and lock
STOREM	ce,cntl,ra,[rb—const8]	store multiple
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
XNOR	rs,ra,[rb—const8]	exclusive nor logical
XOR	rs,ra,[rb—const8]	exclusive or logical

2.3.5 Am29000 Interrupts

Normal program flow may be preempted by an interrupt or trap for which the processor is enabled. The eeffect on the processor is identical for interrupts and traps; the distinction is in the different mechanisms by which the interrupt and traps are enabled. It is intended that interrupts be used for suspending current program execution and causing another program to execute, while traps are 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 processors 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), "Disble 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 differnt 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 eeither predefined or specified by an instruction causing the trap.

Exception vectors 0-21 determines the following exceptions:

(a) Illegal Opcode

(b) Unaligned Address

(c) Out of Range

(d) Coprocessor Not Present

(e) Coprocessor Exception

(f) Instruction Access Violation

(g) Data Access Violation

(h) User Mode Instruction TLB-miss

(i) User Mode Data TLB-miss

(j) Supervisor Mode Instruction TLB-miss

(k) Supervisor Mode Data TLB-miss

(l) Instruction TLB protection violation

(m) Data TLB protection violation

(n) Timer

(o) Trace

(p) INTR0

(q) INTR1

(r) INTR2

(s) INTR3

(t) TRAP0

(u) TRAP1

Exception vectors 22-63 are either reserved, reserved for instruction emulation or associated with Floating Point instructions.

Exception vectors 64-255 are user defined and thus system dependent.

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 thenext 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.

References: Am29000 Streamlined Instruction Processor,
Advanced Micro Devices, 1988

2.4 The 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.

2.4.1 MIPS 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.

2.4.2 MIPS instruction format

Every R2000 instruction consists of a single word (32 bits) aligned on a word boundary. There are three instruction formats:

```
I-TYPE (Immediate)

31  26 25  21 20  16 15           0
  OP     RS     RT      IMMEDIATE


J-TYEPE (Jump)

31  26 25                         0
  OP      TARGET

R-TYPE (Register)

31  26 25  21 20  16 15  11 10      6 5      0
  OP     RS     RT     RD     SHAMT    FUNCT
```

where:

- 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

- IMMEDATE 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

2.4.3 MIPS 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.
- 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 a revision level of the Processor and System Control Coprocessor.

2.4.4 MIPS R2000 instruction set

Instruction	Operands	Comments
ADD	rd,rs,rt	signed immediate add,trap on overflow
ADDI	rt,rs,imm	signed immediate add,trap on overflow
ADDIU	rt,rs,imm	unsigned immediate add
ADDU	rd,rs,rt	unsigned add
AND	rd,rs,rt	logical and
ANDI	rt,rs,imm	logical and immediate
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
CFCz	rt,rd	move control from coprocessor z
COPz	cofun	coprocessor operation
CTCz	rt,rd	move control to coprocessor z
DIV	rs,rt	signed divide
DIVU	rs,rt	unsigned divide
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
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
LUI	rt,immediate	load upper word immediate
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
MFC0	rt,rd	move from system control coprocessor
MFCz	rt,rd	move from coprocessor z
MFHI	rd	move from register LO
MTC0	rt,rd	move to system control coprocessor
MTCz	rt,rd	move to coprocessor
MTHI	rs	move to register HI
MTLO	rs	move from register LO
MULT	rs,rt	multiply
MULTU	rs,rt	unsigned multiply
NOR	rd,rs,rt	logical NOR
OR	rd,rs,rt	logical OR
ORI	rt,rs,immediate	logical OR immediate
RFE		restore from exeption
SB	rt,offset(base)	store byte
SH	rt,offset(base)	store halfword
SLL	rd,rt,amount	shift left logical
SLLV	rd,rt,rs	shift left logical variable
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
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

SUB	rd,rs,rt	subtract
SUBU	rd,rs,rt	subtract unsigned
SW	rt,offset(base)	store word
SWCz	rt,offset(base)	store word from coprocessor z
SWL	rt,offset(base)	store word left
SYSCALL		system call
TLBP		probe TLB for matching entry
TLBR		read indexed TLB entry
TLBWI		write indexed TLB entry
TLBWR		write random TLB entry
XOR	rd,rs,rt	logical exclusive or
XORI	rt,rs,immediate	logical exclusive or immediate

2.4.5 R2000 Exceptions

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 preceding 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 addressesor 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* Twos 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 Instruction Pipeline

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

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.

RD Read any required operands from CPU-registers while decoding the instruction.

ALU Perform the required operation on instruction operands.

MEM Access memory (D-Cache) if required(for Load/Store instructions)

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 immediatly following one of these "delayed" instructions. Instead of having the processor deal with pipeline delays, the R2000 turns over the responibility for dealing with delayed instructions to software.

References:
MIPS R2000 RISC Architecture,
Kane, Gerry,
Prentice Hall, 1987

2.5 SPARC V7

The SPARC (Scalable Processor ARChitecture), designed by Sun Microsystems is an open computer architecture. It is a development from the Berkeley RISC-II and the SOAR (Smalltalk On A Risc). The SPARC architecture has been implemented by several vendors.

2.5.1 SPARC V7 data types

The SPARC architecture defines 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.

2.5.2 SPARC V7 instruction formats and addressing modes

The SPARC instructions are classified into three major formats, simply called *format1, format 2* and *format 3.* Two of these, includes subformats.

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

```
op                          disp30

31  29                                                     0
```

(b) The *format 2* is used by SETHI and branch-instructions:

```
op   rd        op2                       imm22

op  a  cond     op2                     disp22

31  29  28    24    21                                     0
```

- *op* This field places the instruction into one of 3 major formats(format1,format2 and format3).
- *op2* This field comprises bits 24 through 22 of format 2 instructions and selects one of the instructions: UNIMP,Bicc,SETHI,FBfcc,CBccc.
- *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.
- *a* The "a" bit means "annul" in format 2 instructions. This bit changes the behaviour of the instruction encountered immediatly after a control transfer.
- *cond* This field selects the condition code for format 2 instructions.
- *imm22* This field is a 22-bit constant value used by the SETHI instruction.
- *disp22* This field is a 22-bit sign-extended word displacement for branches.

(c) Remaining instruction uses *format 3*:

```
op    rd      op3      rs1     i       asi     rs2

op    rd      op3      rs1     i         simm13

op    rd      op3      rs1         opf          rs2

31 29     24        18          13 12          4       0
```

- *op3* The op3 field selects one of the format 3 opcodes.
- *i* The i-bit selects the type of the second ALU-operand for non-FPop instructions. If i=0, the second operand is r[rs2]. If i=1, the second operand is sign-extended *simm13*.
- *asi* This 8-bit field is the *address space identifier* generated by load/store alternate instructions. The address space identifier generated by the processor is made available to the external system to distinguish up to 256 address spaces.
- *rs1* This 5-bit field selects the first source operand from either the *r registers* or the *f registers*.
- *rs2* This 5-bit field selects the second source operand from either the *r registers* or the *f registers*.
- *simm13* This field is a sign-extended 13-bit immediate value used as the second ALU operand when i = 1.
- *opf* This 9-bit field identifies a floating point operate(FPop) instruction or a coprocessor operate (CPop) instruction.

2.5.3 SPARC V7 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*. The *globals* may be sddressed when any window is active. For any active window the registers are addressed as:

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*

Each window shares its *ins* and *outs* with adjacent windows. The register overlap such 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 yilds 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.

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 below shows 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 occur.

The 32-bit PSR contains various fields describing the state of the IU.

```
 impl    ver     icc    reserved    EC  EF   PIL    S PS ET CWP
31 28 27   24 23   20 19         14 13  12 11     8 7  6  5 4  0
```

- *impl* Bits 31 through 28 identify the implementation number of the processor. This field can not be modified by software.
- *ver* Bits 27 through 24 contains a constant; the meaning of this constant depends on the value of the *impl* field. Can not be modified by software.
- *icc* Bits 23 through 20 contains the IU's condition codes. These bits are modified by dedicated instructions and by the WRPSR (write processor status register) instruction.
- *reserved* Bits 19 through 14 are reserved. This field should only be written to 0 (for future compatibility).
- *EC* This bit determines whether or not the coprocessor is enabled.
- *EF* This bit determines whether or not the FPU is enabled.
- *PIL* Bits 11 through 8 identify the processor interrupt level. The processor only accepts interrupts whose interrupt level is greater than the value in *PIL*.
- *S* Bit 7 determines whether the processor is in supervisor mode or not. Supervisor mode can only be entered by a software or hardware trap.

```
  previous window
r[31]
 .   ins
r[24]
r[23]
 .   locals
r[16]                 active window
r[15                r[31]
 .   outs            .   ins
r[8]                r[24]
                    r[23]
                     .   locals
                    r[16]               next window
                    r[15]           r[31]
                     .   outs        .   ins
                    r[8]            r[24]
                                    r[23
                                     .   locals
                                    r[16]
                                    r[15]
                                     .   outs
                                    r[8]

                    r[7]
                     . globals
                    r[0]
```

Figure 2.1: Three overlapping windows and globals

- *PS* Bit 6 contains the value of the S bit at the time of the most recent trap.
- *ET* Bit 5 is the Trap Enable bit. When 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* Bits 4 through 0 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. Bit 0 corresponds to window 0, bit 1 corresponds to window 1 etc. 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.

```
w31 w30 w29               ....             w2 w1 w0
31  30  29                                 2  1  0
```

The Trap Base register (TBR) contains three fields that generate the address of the trap handler when a trap occur. These are:

```
          TBA                   tt          zero
31                   12 11             4 3      0
```

- *TBA* Bits 31 through 12 comprise 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.
- *tt* Bits 11 through 4 comprise the trap type (*tt*) field. This 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.
- *zero* Bits 3 through 0 are zero. The WRTBR instruction does not affect this field.

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

2.5.4 SPARC V7 instruction set

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
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
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
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
CPOP1		coprocessor operate
CPOP2		coprocessor operate
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

FITOS	*frs2,frd*	convert integer to floating point
FITOD	*frs2,frd*	convert integer to double
FITOX	*frs2,frd*	convert integer to extended
FSTOI	*frs2,frd*	convert single to integer
FDTOI	*frs2,frd*	convert double to integer
FXTOI	*frs2,frd*	convert extended to integer
FSTOD	*frs2,frd*	convert single to double
FSTOX	*frs2,frd*	convert single to extended
FDTOS	*frs2,frd*	convert double to single
FDTOX	*frs2,frd*	convert double to extended
FXTOS	*frs2,frd*	convert extended to single
FXTOD	*frs2,frd*	convert extended to double
FMOVS	*frs2,frd*	move between floating point registers
FNEGS	*frs2,frd*	negate
FABS	*frs2,frd*	absolute value
FSQRTS	*frs2,frd*	square root single
FSQRTD	*frs2,frd*	square root double
FSQRTX	*frs2,frd*	square root extended
FADDS	*frs1,frs2,frd*	add single
FADDD	*frs1,frs2,frd*	add double
FADDX	*frs1,frs2,frd*	add extended
FSUBS	*frs1,frs2,frd*	subtract single
FSUBD	*frs1,frs2,frd*	subtract double
FSUBX	*frs1,frs2,frd*	subtract extended
FMULS	*frs1,frs2,frd*	multiply single
FMULD	*frs1,frs2,frd*	multiply double
FMULX	*frs1,frs2,frd*	multiply extended
FDIVS	*frs1,frs2,frd*	divide single
FDIVD	*frs1,frs2,frd*	divide double
FDIVX	*frs1,frs2,frd*	divide extended
FCMPS	*frs1,frs2*	compare single
FCMPD	*frs1,frs2*	compare double
FCMPX	*frs1,frs2*	compare extended
FCMPES	*frs1,frs2*	compare single and exception if unordered
FCMPED	*frs1,frs2*	compare double and exception if unordered
FCMPEX	*frs1,frs2*	compare extended and exception if unordered
IFLUSH	*address*	flush instruction cache
JMPL	*address,rd*	jump and link
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
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
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
RESTORE	*rs1,rs2/imm,rd*	restore callers window
RETT	*address*	return from trap
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

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
SLL	*rs1,rs2/imm,rd*	shift left logical
SRL	*rs1,rs2/imm,rd*	shift right logical
SRA	*rs1,rs2/imm,rd*	shift right arithmetic
SAVE	*rs1,rs2/imm,rd*	save callers window
SETHI	*const,rd*	zero least sign 10 bits, replace high order bits
TADDCC	*rs1,rs2/imm,rd*	tagged add and modify icc
TADDCCTV	*rs1,rs2/imm,rd*	tagged add, modify icc and trap on overflow
TSUBCC	*rs1,rs2/imm,rd*	tagged subtract and modify icc
TSUBCCTV	*rs1,rs2/imm,rd*	tagged subtract, modify icc and trap on overflow
TA	*address*	trap always
TN	*address*	trap never
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
UNIMP	*const22*	unimplemented instruction
WDY	*rs1,rs2/imm,y*	write y register
WDPSR	*rs1,rs2/imm,psr*	write processor state register
WDWIM	*rs1,rs2/imm,wim*	write window invalid mask register
WDTBR	*rs1,rs2/imm,tbr*	write trap base register
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

2.5.5 SPARC V7 traps and exceptions

SPARC V7 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:

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

References:

The SPARC Architecture Manual Version 7,
Sun Microsystems,Inc. 1987

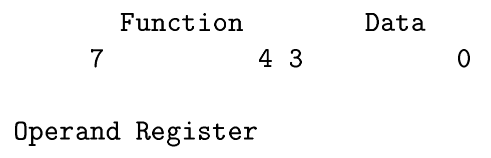

Figure 2.2: Instruction format

2.6 The 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 (a detailed description will be given in chapter 3) and furthermore, special communication links supporting large multiprocessor systems. These, among other things, makes it difficult to treat the Transputer as other microprocessors descirbed in this work. Hence, the following paragrafs will differ slightly from previous chapters.

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 instruction formats and addressing modes

In the T800 there is only a single instruction format. 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.

Since most of the instructions operates on the evaluation stack there is no need for complicated addressing modes. There are not any either.

2.6.2 The T800 registers

In contrast to other RISC architectures there are only six CPU-registers:

- 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 forms 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.

2.6.3 The T800 instruction set

Instruction	Operands	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
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
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
REV	reverse
XWORD	extend to word
XDBLE	extend to double
CSNGL	check single
MINT	minimum integer
DUP	duplicate top of stack
MOVE2DINIT	initialise data for 2D block move
MOVE2DALL	2D block copy
MOVE2DNONZERO	2D block copy non-zero bytes
MOVE2DZERO	2D block copy zero bytes
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
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
LDTIMER	load timer
TIN	timer input
TALT	timer alt start
TALTWT	timer alt wait
ENBT	enable timer
DIST	disable timer
IN	input message
OUT	output message
OUTWORD	output word
OUTBYTE	output byte
ALT	alt start
ALT	alt wait
ALTEND	alt end
ENBS	enable skip
DISS	disable skip
RESETCH	reset channel

ENBC	enable channel
DISC	disable channel
RET	return
LDPI	load pointer to instruction
GAJW	general adjust workspace
GCALL	general call
LEND	loop end
STARTP	start process
ENDP	end process
RUNP	run process
LDPRI	load current priority
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
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
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
FPENTRY	floating point unit entry
FPREV	floating point reverse

FPDUP	floating point duplicate
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
FPCHKERROR	check fp error
FPTESTERROR	test fp error false and clear
FPUSETERROR	set fp error
FPUCLEARERROR	clear fp error
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
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
FPADD	floating-point add
FPSUB	floating-point subtract
FPMUL	floating-point multiply
FPDIV	floating-point divide
FPABS	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

References:

The Tranputer Databook,
Inmos Limited, 1989

The transputer Applications Notebook,
Architecture and software,
Inmos Limited, 1989

2.7 Acorn

The VL86C010 Acorn RISC Machine is a full 32-bit general-purpose microprocessor designed using RISC methodologies. The processor is targeted for the microcomputer, graphics, industrial and controller market for use in stand-alone or embedded systems.

2.7.1 Acorn data types

The processor supports two data types: eight bit bytes and 32-bit words, where words must be aligned on four byte boundaries. Data operations are only performed as word quantities.

2.7.2 Acorn instruction formats and addressing modes

There are six instruction formats, each describing the possible addressing modes.

(a) Branch instruction:

```
31   28 27    25 24 23                                   0
 Cond     101     L              offset
```

- Cond field is a condition (described below)
- L field: 0 = branch, 1= branch and link
- offset is the branch target offset from PC

(b) Data processing Type:

```
31   28 27 26 25 24     21 20 19   16 15   12 11         0
 Cond    0    I    Opcode    S    Rn      Rd      Operand2
```

- I field determines if the operand2 is an immediate 8-bit value, or if it is a register
- S, if set, the condition codes are set according to the instruction
- Rn is the first operand register
- Rd is the destination register
- Operand 2 specifies the second operand. When this operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the shift field in the instruction. This field indicates the type of shift to be performed.

  ```
  11                       4 3       0
               shift                  Rm
  ```

  shift applied to Rm, Rm is the second operand register.

  ```
  11             8 7              0
                 shift          imm
  ```

  shift applied to immediate value, imm is the unsigned 8-bit immediate value shift applied to Rm, Rm is the second operand register.

(c) Multiply and Multiply-accumulate:

```
31   28 27     22 21 20 19     16 15    12 11     8 7    4 3   0
 Cond    00.0      A   S     Rd      Rm       Rs     1001   Rm
```

- A, accumulate bit, 0 = multiply, 1 = multiply and accumulate
- S, if set, the condition codes are set according to the instruction

- Rd,Rs and Rm is operand registers

(d) Single Data Transfer

```
31   28 27 26 25 24 23 22 21 20 19   16 15   12 11         0
 Cond    0  1  I  P  U  B  W  L   Rn      Rd       offset
```

- I , I=0 if offset is an immediate value

  ```
  11                           0
       unsigned 12 bit offset
  ```

  I = 1

  ```
  11                4 3        0
     shift appl to..     Rm
  ```
- P, pre/post, 0= add offset after transfer, 1 = add offset before transfer
- U, up/down, 0=subtract offset from base 1 = add offset to base
- B, byte/word, 0=word, 1=byte
- W, 0=no write back, 1= write address into base
- L, 0= store to memory, 1= load from memory
- Rn, base register
- Rd, source/destination register
- offset as described by I-field

(e) Block Data Transfer:

```
31   28 27   25 24 23 22 21 20 19   16 15              0
 Cond     100    P  U  S  W  L   Rn      register list
```

- P, pre/post, 0= add offset after transfer, 1 = add offset before transfer
- U, up/down, 0=subtract offset from base 1 = add offset to base
- S, PSR and force user bit, 0= do not load PSR, force user mode 1 = load PSR or force user mode
- W, 0=no write back, 1= write address into base
- L, 0= store to memory, 1= load from memory
- Rn, base register
- register list specifies registers in the transfer

(f) Software Interrupt

```
31  28 27   24 23                              0
 Cond   1111     Comment field ignored by proc.
```

2.7.3 Acorn registers

The processor provides 27 registers of wwhich 16 are accessible in any mode of operation. In User Mode, registers R0 to R13 are accessible as general-purpose registers. R14,the User-mode return- link register is specific to the user mode. It's contents are mapped with those of other return-link registers as the mode is changed. The return-link register is used by the branch and link instruction in a procedure call sequence but may be used as a general-purpose register at other times. The least significant two bits of the processor status word (PSW) define the current mode of operation.

Seven registers are dedicated to the FIRQ mode and overlie user-mode registers R8-R14 when the fast interrupt request is serviced. The registers R8FIRQ to R14FIRQ are local to the fast interrupt service routine and are used instead of the registers R8-R13. Register R14FIRQ holds the address used to restart the interrupted program instead of pushing it onto a stack at the expense of another memory cycle.

To registers are dedicated to the IRQ mode and overlie user mode registers R13 and R14. R14IRQ holds the restart address and R13IRQ is general purpose dedicated to the IRQ-mode.

To registers are dedicated to the supervisor mode and overlay registers R13 and R14 when a supervisor mode switch is made using a software interrupt instruction. Operation of these two registers are similar to the IRQ mode.

One register, R15, contains the processor status word and program counter and is shared by all modes of operation. The upper six bits are processor status, the next 24 bits are the program counter (word address) and the last two bits indicate the mode.

2.7.4 Acorn instruction set

The processor supports five basic types of instructions. These are: Data processing, data transfer, block data transfer, branches and software interrupt. All instructions contain a 4-bit conditional execution field that can cause an instruction to be skipped if the condition specified is false. The condition may be one of:

`EQ`	`equal`
`NE`	`not equal`
`CS`	`unsigned higher or same`
`CC`	`unsigned lower`
`MI`	`negative`
`PL`	`positive or zero`
`VS`	`overflow`
`VC`	`no overflow`
`HI`	`unsigned higher`
`LS`	`unsigned lower or same`
`GE`	`greater or equal`
`LT`	`less than`
`GT`	`greater than`
`LE`	`less than or equal to`
`AL`	`always`
`NV`	`never`

Instruction	Operands	Comments
B{Cond}	label	branch
BL{Cond}	label	branch and link

AND{Cond}	Rd,Rn,Operand2	bitwise logical and
EOR{Cond}	Rd,Rn,Operand2	bitwise logical exclusive or
SUB{Cond}	Rd,Rn,Operand2	subtract operand2 from operand1
RSB{Cond}	Rd,Rn,Operand2	subtract operand1 from operand2
ADD{Cond}	Rd,Rn,Operand2	add operands
ADC{Cond}	Rd,Rn,Operand2	add operands with carry
SBC{Cond}	Rd,Rn,Operand2	subtract operand2 from operand1 with carry
RSC{Cond}	Rd,Rn,Operand2	subtract operand1 from operand2 with carry
TST{Cond}	Rd,Rn,Operand2	as AND but result not written
TEQ{Cond}	Rd,Rn,Operand2	as EOR but result not written
CMP{Cond}	Rd,Rn,Operand2	as SUB but result not written
CMN{Cond}	Rd,Rn,Operand2	as ADD but result not written
ORR{Cond}	Rd,Rn,Operand2	bitwise logical OR of operands
MOV{Cond}	Rd,Rn,Operand2	move operand 2 (operand 1 igored)
BIC{Cond}	Rd,Rn,Operand2	bit clear
MVN{Cond}	Rd,Rn,Operand2	move NOT operand 2 (operand 1 igored)

The operand2 may be shifted before it is used in an operation. This is accomplished by adding a second mnemonic:

LSL	op	logical shift left
ASL	op	arithmetic shift left
LSR	op	logical shift right
ASR	op	arithmetic shift right

MUL{Cond}{S}	Rd,Rm,Rs	Multiply Rs by Rm store result in Rd
MULA{Cond}{S}	Rd,Rm,Rs,Rn	

Multiply Rs by RM finally add Rn and store result in Rd

LDR{Cond}{B}{T}	Rd,address	load register from memory
SDR{Cond}{B}{T}	Rd,address	store in memory from register

LDM{addr mode}	Rn,rlist	load multiple registers from memory
STM{addr mode}	Rn,rlist	store multiple registers in memory
SDR{Cond}{B}{T}	Rd,address	store in memory from register

2.7.5 Acorn processing states

Normal execution may be preempted by an external device (IRQ or FIRQ), an internally raised exception or an exception raised by software (SWI). The processor supports a partially overlapping register set so that when interrupts are taken, the contents of the register array do not have to be saved before new operations can begin.

Acorn exception handling

The processor exception map is:

Address	Function	Priority level
0	Reset	0
4	Undefined Instruction	6
8	Software Interrupt	7
C	Abort(prefetch)	5
10	Abort(data)	2
14	Address Exception	1
18	Normal Interrupt(IRQ)	4
1C	Fast Interrupt(FIRQ)	3

'0' denotes the highest priority. These vector addresses normally will contain a branch instruction to the associated service routine except for the FIRQ entry. In order to reduce latency, the FIRQ service routinee may begin at address 001Ch if software designer so chooses.

Whenever the processor enters the supervisor mode, whether from an SWI, address exceptiion, undefined instruction trap, prefetch or data abort the IRQ is disabled and FIRQ unchanged.

References:
VL86C010 Risc Family Data Manual
VLSI Technology Inc, 1987

2.8 THOR

The Saab-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.

2.8.1 THOR data types

Different instruction operates on one (or more) of the following data types: 32-bit integer (unsigned/signed), 64-bit signed integer, 32-bit IEE-754 single precision floating point and a special long-precision floating-point format:

```
Sign and Mantissa (high 32 bits)

Mantissa (low 32 bits)

Exponent, 32-bits two's complement
```

2.8.2 THOR instruction formats and addressing modes

There are five different instruction formats. The format determines the instruction length (in bytes) an how to interpret any parameter:

A 16-bit encoded instruction designated "2".

```
16        8  7             0
   opcode      ext. opcode
```

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.

```
16        8 7              0
   opcode          P
```

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 $-2^{23}$to$2^{23}-1$.

```
31    24 23                      0
 opcode              P
```

The format "4b" is identical with "4a" except from 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. The following summarises the available addressing modes:

Stack Relative

The Operand Effective Address is calculated relative to the TOS, either implicit or by adding the parameter to TOS.

Program Counter Relative

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)

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

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.

2.8.3 THOR registers

The processor maintains the following on-chip registers.

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

- **Configuration Register** is used for hardware specific parameters:

```
 0 0 0  DC1  WS1  0 0 0  DC0  WS0  0  S  RM CC DW   CLK
31  29    26   24     21   19    16 15 14  12 10 8         0
```

- **CLK** Clock Frequency is used to set a division factor (1 to 255) of the chip clock to get frequency, nominally 1 MHz. Clocks are stopped when this field is zero.
- **DW** Sets the *Data bus Width*, 8 bits, 16 bits or 32 bits can be used.
- **CC** *Cache Control* controls the use of data and instruction caches.
- **RM** Controls the IEE-754 floating point *Rounding Mode.*
- **S** Determines the *Scheduling Mode* used.
- **WS0** *Waitstate 0*, sets the number of waitstates in the first 512 MBytes of memory. From 0 up to 6 waitstates can be used. Setting this field to 7 indicates use of the *Ready* signal.
- **DC0** *Data Check 0* sets the data error checking mode in the first 512 MBytes of memory. Mode may be one of: *Parity, EDAC* or disabled.
- **WS1** *Waitstate 1*, sets the number of waitstates in the second 512 MBytes of memory. From 0 up to 6 waitstates can be used. Setting this field to 7 indicates use of the *Ready* signal.
- **DC1** *Data Check 1* sets the data error checking mode in the second 512 MBytes of memory. Mode may be one of: *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 byte address.

• 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 **Real-Time-Clock** (RTL,RTM) is a 63 bit value read as two 32-bit registers. Incrementation of this register is due to contentsin the Configuration Register.

• The **Task Pointer** (TP) value defines the currently executing task, its value can renge from 1 to 7.

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

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

Mnemonic	Name	Size(bits)
PC	Program Counter	31
RR	Result Register	32
SR	Status Register	32
TOP	Top Register	32
TOS	Top of Stack	29
AR	Accept Register	7
DR	Delay Register	32
ER	Exception Register	31

PR	Priority Register	6
BOS	Beginning of Stack	29
EOS	End of Stack	29

- The **Program Counter** (PC) holds the address of the last instruction read from memory. This address is a halfword address.
- The **Result Register** (RR) holds the least significant half of arithmetic instructions that yuilds 64-bit results.
- The **Status Register** (SR) holds condition codes, hardware exception numbers and Ada support information

```
 0 .. 0   RZ   UM  TSI  QE  RT  EC  AW  DLY  RDY  I C N Z   E
31    19 18  16 15  14   13  12  11  10   9    8   7 6 5 4 3   0
```

  - The **Exception Number Field** (E) is set to the exception number when a hardware exception occurs.
  - The **Negative Flag** (N), **Zero Flag** (Z) **Carry Flag** (C) is set according to arithmetic conditions.
  - The **Inexact Flag** (I) is set when a calculated floating-point result requires rounding.
  - The **Ready Flag** (RDY) is set when this task is ready to execute.
  - The **Delay Flag** (DLY) is set when this task is delayed.
  - The **Accept Wait** (AW) is set when this task is waiting for an accept statement.
  - The **Entry Call Flag** (EC) is set when this task is performing an entry call.
  - The **Remote Task Flag** (RT) is set when this task is doing rendezvous with a remote task.
  - The **Queued Entry Flag** (QE) is set when queued calls exist for an entry in this task.
  - 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 **Rendezvous Field** (RZ) is set to the calling task number when a rendezvous with this task starts. If this field is zero there is no rendezvous in progress.
- The **TOP** register holds the word at the stack top (pointed at by TOS).
- The **TOS** register points at the word on top of stack.
- **Accept Register** (AR). When an enrty call has been performed the bit corresponding to the calling task is set.
- 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) a scheduling is perfomed.
- The **Exception Register** (ER) points to the exception information block in the stack. ER is a word pointer.
- The **Priority Register** (PR) is used to determine th next task to execute when scheduling occurs. 32 priorities can be defined.

- **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.

2.8.4 THOR instruction set

Instruction	Operands	Comments
ADD	*expr*	add integer
ADDC	*expr*	add with carry, integer
ADDF	*expr*	add float
ADDI	*expr*	add immediate
ADDU	*expr*	add unsigned
AND	*expr*	logical and
CALL	*expr*	call subprogram
CALLP	*expr*	call protected
CLL	*expr*	compare lower limit
CLRF	*expr*	clear flags
CMP	*expr*	compare
CMPF	*expr*	compare float
CUL	*expr*	compare upper limit
DIVF	*expr*	divide float
DIVS	*expr*	divide short
FBC		first bit changed
FBCL		first bit changed, long
FLSH		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
MTOS	*expr*	move top of stack
MUL	*expr*	multiply
MULF	*expr*	multiply float
MULI	*expr*	multiply immediatly
MULS	*expr*	multiply short
MULU	*expr*	multiply unsigned

NOP		no operation
NOT		logical not
OR	*expr*	logical or
POP	*expr*	pop value from stack
POPR	*reg[,expr]*	pop register
POPX	*expr*	pop indexed
PSH	*expr*	push value onto stack
PSHI	*expr*	push immediate
PSHR	*reg[,expr]*	push register
PSHX	*expr*	push indexed
REMS	*expr*	remainder short
RET		return
RETU		return to user mode
SETF	*expr*	set flags
SL	*expr*	shift left
SLD	*expr*	shift left dynamic
SLDL	*expr*	shift left dynamic long
SR	*expr*	shift right
SRA	*expr*	shift right arithmetic
SRAD	*expr*	shift right arithmetic dynamic
SRADL	*expr*	shift right arithmetic dynamic long
SRD	*expr*	shift right dynamic
SRDL	*expr*	shift right dynamic long
SUB	*expr*	subtract
SUBB	*expr*	subtract with borrow
SUBF	*expr*	subtract float
SUBU	*expr*	subtract unsigned
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
XOR	*expr*	logical exclusive or

2.8.5 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 differant 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 , but they can also be raised by software, remaining exception numbers are user defined.

THOR hardware defined exception numbers

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

References:

Stack Risc Microprocessor Instruction Set
Architecture For Prototype Chip,
Saab-Space 1990.

Stack Risc Microprocessor Assembler User's Guide
Saab-Space 1990

Chapter 3

RISCs in a real-time environment

The design of reduced instruction set computers is guided by a design philosophy. It really does not rely upon inclusion of a set of required features. There is no strict definition of what constitutes a RISC-design however some common features may be observed in several examples of RISC-designs.

In this chapter, differences and similarities between the studied architectures will be focused. Some High Level Language aspects will be discussed and topics as interrupts, context switch, task switch (from a HLLs point of view), subprogram calls, interlocking etc will be treated.

3.1 Common RISC features

As a result of RISC design efforts one may observe some common features:

- Only LOAD/STORE instructions may access memory. This is the key to single cycle execution of instructions. Operations on register contents are faster than operations on memory contents. References to cached or buffered operands may be as rapid as register references if the operand is in the cache and the cache is on the CPU-chip.
- 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, complex 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.

MPU	LOAD/ STORE	INSTR	INSTR FORMATS	REGISTERS
MC 88100	yes	77	14	32(4)
I 80960KB	yes	184	4	80(4)
Am 29000	yes	112	1	192
MIPS R2000	yes	119	3	32
SPARC V 7	yes	81	3	122(32)
T800	yes	162	1	0
Acorn	yes	44	6	27
THOR	yes	70	5	0

Note:
The number of instructions with unique op-codes
as listed in the available documentation.

Table 3.1: RISC features

Table 3.1 shows how the studied processors conform to some typical RISC features.

- Is it a LOAD/STORE architecture? To be, there must not be a single instruction besides the dedicated LOAD/STOREs that access memory.
- The number of instructions.
- The number of instruction formats.
- The number of general purpose registers. Number of FPU registers within parenthesis

It had been desirable also to include an addressing modes count figure. However it has not been possible to distinguish formal (hardware supported) from assembler directed (synthesisised) addressing modes in the available documentation. Therefore theese figures are omitted.

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, it might be:

- 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 and prevents 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 such that the elimination of the cause and restarting 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 provides a method of implementing operating system calls etc. A trap may be conditional such as TRAP on OVERFLOW and used in conjunction with arithmetic operations.

When such an event occur, the processor performs the appropriate handler routine. This has been shortly described in the previous chapter. These events introduces preemptiom of normal execution that sometimes may violate time constraints in real time systems. The ability to respond to these events is essential, and more: the RISC design philosophy introduces several difficulties in this procedure.

The typical processor behaviour when it recognizes an event that breaks the normal flow of instruction execution can be described as:

(a) Finish current instruction (does not apply to exception).

(b) Check interrupt priority level versus current processor level. I.e whether the interrupt should be serviced or not.

(c) Save enough processor status to be able to continue processing after the interrupt has been serviced.

(d) Fetch the appropriate interrupt vector.

(e) Execute the interrupt service routine

(f) Restore the interrupted status

(g) Continue normal processing.

If we assume that there is no delay between the time from which the interrupt was asserted until the time the processor recognizes the interrupt then items a) through d) together constitutes the "interrupt latency time"

In the following paragraphs each, of the studied processors, is analyzed with respect to the actions taken while servicing an event.

MC 88100

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

(a) Finish current instruction (synchronize)

(b) Freeze all pipelines except the data unit

(c) Allow data unit to complete (or fault)

(d) Freeze all shadow registers and copy the PSR to the TPSR.

(e) Set new PSR to indicate exception processing

(f) Generate vector

(g) Prefetch vector and vector+4

This is carried out within 6 clock cycles. A typical instruction will require a maximum of 4 clock cycles to complete. We assume no wait for the data unit and thus the MC88100 will start the service routine in 10 clock cycles upon the interrupt.

I80960KB

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

(a) It temporarily stops work on its current task, whether it is working on a program or another interrupt procedure.

(b) It reads the interrupt vector.

(c) It compares the priority of the vector with the processors current priority.

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

(e) 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:

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

(b) 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.

(c) switches to the interrupted state.

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

(e) clears trace-fault-pending and trace-enable flags.

(f) allocates a new frame on the interrupt stack and switches to the interrupt stack.

(g) sets the frame return status field.

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

Am29000

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

(a) Instruction execution is suspended

(b) Instruction fetching is suspended

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

(d) The contents of the Current Processor Status Register are copied into the Old Processor Status Register.

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

(f) The address of the first instruction of the interrupt or trap handler is determined.

(g) The processor determines whether or not the first instruction is in instruction ROM.

(h) An instruction fetch is initiated using the instruction address as determined in previous steps. At this point, normal execution resumes.

MIPS R2000

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

(a) The R2000 branches to the general exception vector for this exception.

(b) the *IP* field in the *Cause* register show 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.

(c) The R2000 saves the *KUp,IEp,KUc,* and *IEc* bits of the *Status* register in the *KUo, IEo,KUp* and *IEp* bits respectivly, and clears the *KUc* and *IEc* bits.

SPARC V 7

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

(a) It disables traps

(b) It copies the S field of the PSR into the PS field and then sets the S field to 1.

(c) It decrements the CWP by 1 modulo 7.

(d) It saves the PC and nPC into r[17] and r[18], respectively of the new window.

(e) It sets the *tt* field of the TBR to the appropriate value.

(f) 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.

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.

Acorn

Upon recognition of a FIRQ the processor:

- Save R15 in R14IRQ
- Force processor mode bits into FIRQ mode
- Force PC to 01Ch

The interrupt latency time is, most of all, due to the interrupted instruction execution time.

3.3 Context Switch

A processors context is characterised by:

- Accessible register contents
- Unaccessible (internal) register contents
- Processor internal state, i.e pipe-line state, control-unit state, integer-unit state etc.

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 (pre-) defined state is reached. For example, when an interrupt occurs, the current instruction is allowed to complete and a return address is saved in a register or on a stack. However this describes a minimum context to be preserved. If the accessible register contents are to be exchanged (entire context switch) then a small register file obviusly will benefit compared to a large one.

Thus, the benefits from a large register file (less memory traffic) must be related to the penality when a context swith occurs. Processors with large register files may provide optional use of it to accomplish fast context switches.

3.4 Task Switch

In a multi-tasking environment each program under execution constitutes a *process.* 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 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 process PCB. That is: A minimum of operations performed to freeze a process and maintain the ability to restart it at any later time the operating system must be:

(a) Save the entire processor context by pushing it onto the stack.

Figure 3.1: PCB organisation in memory

(b) 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. In a real case, however, there is significantly more "housekeeping" activities that has to be carried out.

For a complete Task Switch the old process must be preserved, a new task must be selected and started. That is: at least two processor context switches and the selection contributes to the total time required. In a system with several runnable processes (tasks) the operating system must choose the one with highest priority. There might for example be processes waiting for IO in the system, 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(runnable or not) and priority to pick the runnable process with the highest priority. This type of "housekeeping" can be quit time consuming and may, in some cases, violate time constraints laid upon the operating system.

In order to gain some knowledge about a processors ability to handle a task switch, let us construct a hypotethic case. Assume a small system with exactly 10 tasks. The tasks may be either runnable or waiting and they have dynamic priorities. Figure 3.1 describes the PCB:s organisation in memory. The time required to perform a simple (and fast) task switch may be described as:

```
  Context Switch - Time spent searching x 10 PCB:s - Context Switch
```

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, 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)N.PCB,r2       get next PCB pointer  {10}
        sub     1,r1               exit ... {10}
        cmp     0,r1               .. when .. {10}
        jmpne   .L1                .. all PCB:s searched {9}
        ....
        ....
```

In the following paragraphs, this formal 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 is not 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 penalities. The code is not tested, and thus not highly reliable...

MC88100 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      r6 temp hold, priority (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,N.PCB      get next PCB pointer (memory access) {40}
        sub     r1,r1,1          exit ...  {10}
        bcnd    gt0,r1,.L1       .. when all PCB:s searched {18}
        ....
        ....
```

I80960KB 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     N.PCB(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}
        ....
        ....
```

Am29000 PCB search

Note: Instruction timing not available from the documentation. Figures are estimations based on related descriptions.

```
        ....
        ....
; 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,N.PCB  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}
        ....
        ....
```

MIPS R2000 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,N.PCB(r2)   PCB pointer(high) (memory access) {10}
        lh      r7,N.PCB+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}
        ....
        ....
```

SPARC V 7 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+N.PCB,r2     get next PCB pointer (memory access) {1}
        sub     r1,1,r1         exit ...  {1}
        bne,a   .L1             .. when all PCB:s searched {1}
        ....
        ....
```

T800 PCB search

For the T800 there is really no need for a software process scheduler, 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. In the linked process list (Figure 3.2) process S is executing and P, Q and R are active awaiting execution. The figure shows the low priority process queue. The high priority process queue acts similar.

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

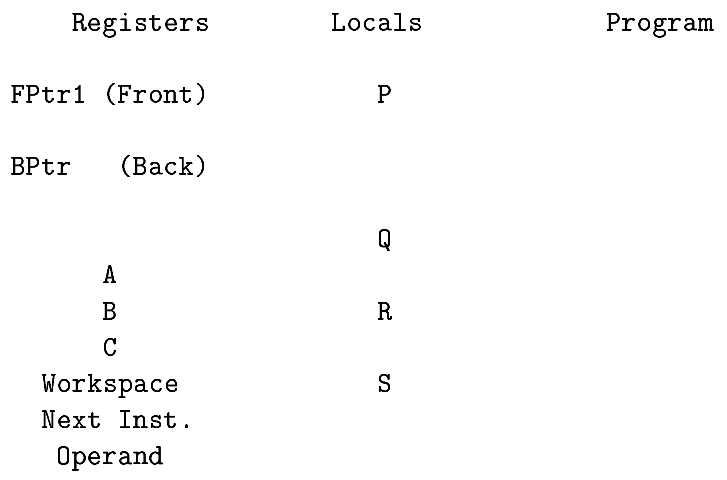

Figure 3.2: Linked Process List

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.

Despite the fact that the T800 do not support dynamic priorities and that there may be only high or low priority processes, this is an elegant way of handling a task switch. It do not need a software kernel to perform process scheduling, its a feature of the architecure.

Acorn 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
mov r2,#PCB0PTR
mov r5,r2          ptr to hi priority task  {1}
        mov     r1,#10          number of PCB:s to search {1}
        mov     r3,#0           initial priority (lowest) {1}
        mov     r4,#0           initial PCB ID (undefined) {1}
.L1:    ldr     r6,(r2,#T.PRI)   r6 temp hold, priority (memory access) {30}
        cmp     r3,r6           compare priorities, result in r7 {10}
        bhi     .L2             branch if previous is greater  {30}
        mov     r3,r6           substitute new priority  {1}
        ldr     r4+T.ID,r4      remember task ID (memory access) {1}
        add     r0,r2,r5        remember PCB ptr  {1}
.L2:    ldr     r2,(r2,#N.PCB)  get next PCB pointer (memory access) {30}
        sub     r1,#1           exit ...  {10}
        bne     .L1             .. when all PCB:s searched {27}
        ....
        ....
```

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.

Conclusions

The PCB searches may be summarised as:

```
Processor       Processor Cycles (approx )
MC88100         148
I80960KB        136
```

Am29000	133
MIPSR2000	145
SPARC V7	144
T800	hardware implemented
Acorn	144
THOR	hardware implemented

Task switches in real-time systems are a time-consuming matter. Moreover, since processes are created and removed dynamically the time spent for these activities become very difficult to predict. Some general 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.

3.5 Subprogram Calls

A subprogram call is a result of a HLL function (procedure) call statement. Consider the following:
callsub(p1,p2 ... ,pn);
The compiler 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 of wich each one may prevent the processor from single cycle execution. Thus, its preferable to hold and pass the parameters in registers. This may be accomplished by conventions for register usage, that is directives for the compiler writer of how to dispose the register set. The register usage conventions is a matter of the processor architecture and these conventions will be described in the next paragraphs.

MC 88100 register conventions

The outline of the MC88100 general purpose registers is described in paragraph 2.1.3, page 15 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, if these conventions are adapted, the architecture supports subprogram calls with eight (or fewer) parameters well.

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, thus 15 parameters could conviently be passed to (or from) a subprogram.

Am29000 register conventions

The Am29000 utilises a large, on chip register set. The following will concentrate on "subprogram calls/returns" and does not cover the entire proposed usage of the register set.

When a subprogram is called, a new activation record, or "stack frame" is allocated on the run-rime stack. This record includes local variables, arguments to the subprogram and a return address. Figure 3.5 shows an example of the run-time-stack during a subprogram call.

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.

```
                       out args X
                       in args to A          higher memory addresses

Activation
Record A               locals A


                       out args A
                       in args B

                                             Activation
                       locals B              Record B


                       out args B
                                             Top of Stack
```

Figure 3.3: Am29000 Run-time stack example

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.

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. 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** contins the return address. Used for expression evaluation.

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

SPARC V 7 register conventions

The organisation of SPARC V7 register windows was described in paragraph 2.5.3, page 48. Figure 2.1,(page 50) 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, less parameters can be assumed thus providing additional local registers.

3.6 Data Interlocking and Delayed Branches

A data interlock occurs as a result from data dependency between operations on registers which concurrently occupies the pipeline. For example the instruction sequence:

```
b = 2*a + b;
```

may generate code, assuming "a" in *R2* "b" in *R3*, such as:

```
shl     R2              ; 2*a
add     R2,R3,R3        ; b = 2*a + b
```

Now, the add-instruction must not execute before the shift instruction is through and have produced valid data in register R2. This exemplifies pipeline penality associated with data interlock.

There are different strategies of handling this in hardare. One might be to introduce a dedicated register (often called "scoreboard register") to keep track of registers while they are operated on. The hardware may thus detect critical situations and for a case like this delay execution of the add-instruction until the shift-instruction has completed.

A quite different strategy, is to let the processor execute the next instruction as fast as it possibly can under the assumption that the compiler has rearranged code in a manner to prevent all interlock situations.

Another pipeline penality is associated with changes in program flow. For a *branch* or *jump* instruction the pipeline has to be flushed and refilled with the instructions at the destination address . To reduce this penality one may incorporate *delayed branching*, that is before the branch is performed the next instruction is executed. Thus the compiler has to rearrange code so that a useful instruction is inserted *immediatly after* the branch or jump instruction.

MC88100 handles data dependencies internally by the use of a scoreboard register. I.e execution pipeline stalls while the preceeding instruction finish the write-back- stage if there is a data dependency. Delayed branches are provided as an option. The compiler may choose to use or not. A facility that probably reduce code size.

I80960 provides scoreboarding for global and local registers. If a register used in an operation is marked in-use the processor can not continue execution of that instruction. This typical apply to load-operations. *Register Bypassing* is a mechanism that allows instructions that would ordinarily require source operandsd to be placed in registers to be executed without accessing one or both of the source registers. Register bypassing occurs in either of two circumstances. First, when the IEU executes an instruction with two source operands, register bypassing occurs if one or both of the operands are literals. Second, register bypassing will also occur when the second of two source operands is the result of the previous instruction. The net result of register bypassing is the saving of one clock cycle. On conditional branch instructions, the ID uses a condition code scoreboard to streamline the branching process.

Am29000 detect data dependencies and provides a mechanism that detects:

(a) If one of the source registers matches the destiation register of the immediatly previous instruction.

(b) If one of the source registers matches a register destinated for an outstanding load-operation.

In the first case, the result of the execute stage is selected as an operand, thus bypassing the write-back stage, (data forwarding). In the second case the processor may enter the Pipeline Hold mode if the load has not completed.i The effect of jump and call instructions is delayed by one cycle. Jump and call instructions are collectively referred to as delayed branches.

The R2000 always perform delayed branches, assuming the compiler to arrange for proper operation. Data dependencies are treated in a similar manner, i.e no hardware mechanism detects data dependencies.

SPARC V7 behaves similar to MC88100 in the case of delayed branches. An "annual-bit" is used in the branch op-codes from which the processor determines whether it is to execute the next instruction or not.

In order to decrease pipeline penalities **THOR** provides hardware detection and actions for some situations. In general though, the responsibility for handling this is laid upon the compiler.

3.7 Real Time System Support

Am29000

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.

T800

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.

THOR

THOR has a built in real time clock to keep track of system time. Furthermore, each task 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.

3.8 Conclusions

The goal of a reduced instruction set is to make *a simple hard-wired processor* and to carry out as *many functions as possible in software.* This demands for *efficient compiling techniques.* The RISC approach has had a major influence on development of compilers during the last years. Studies made within RISC projects often concentrates around a predetermined *well defined environment.* I.e medium size, multitasking, multiuser operating systems. A commonly goal seems to be to acheive a *general purpose processor.* Benchmark studies have shown that RISC processors outperform conventional CISC processors. RISC processor performance is often measured relative to a VAX 11 (1978) or a Motorola MC68020 (1984). RISC designs employs:

- a large register file
- a reduced intruction set
- pipelined execution
- few addressing modes
- fixed instruction format

These features is essential to *single cycle execution.* This ability is certainly favouring the RISC designs. The best performance, from this ability is acheived in an environment with "patiant" I/O requests. Running a benchmark is a good example of such environments. However, this is not true in an *embedded real time environment.* On the contrary, theese systems includes functions that require *extremely rapid responses on external events.* These events may *occur frequently.*

References:
Reduced Instruction Set Computers, Stallings W,
Computer Organisation and Architecture, 1986, pp 431-435

Reduced Instruction Set Computers, Patterson, D A
Communications of the ACM, January 1985, pp 8-21

VLSI Processor Architecture, Hennessy J L,
IEEE transactions on Computers, December 1984, pp 1221-1246

RISC: Back to the Future?, Bell, C
Datamation, June 1986, pp 96-108