6. Interrupts

6.1 Examples and Alternate Names

Interrupts are events that require a change in the control flow, other than jumps or branches. They appeared as an efficient way to manage I/O devices: instead of continuously checking the state of a peripheral device (polling), the CPU does something else until it is announced by the peripheral (with an interrupt) that it is ready; at this moment CPU suspends its regular operation and take appropriate measures to handle the interrupt. Note that polling is not only very inefficient, in that the CPU does nothing but testing the status of some peripheral device, it may lead to a system that get stuck in the case the awaited event never occurs. Even if it eventually resumes the computation it is no guarantee that other events have not been lost. Such an approach can be found only in very simple processing units.

Interrupts make the Control Unit specification and implementation extremely difficult; they must be considered from an early stage of the design. Here are some of the situations in which interrupts are used:

- I/O device request
- arithmetic overflow/underflow
- hardware malfunction (with the most frequent case of memory error)

- real-time clock;
- power failure;
- wrong instruction;
- unaligned memory access (if the system requires alignment);
- page fault (object not in memory);
- user defined breakpoints;
- others.

There is no unique name for this kind of change in control flow which we usually call interrupt.

- In IBM-360 and Intel 80x86 every event is called an interrupt.
- In Motorola 68000 family every event is called an exception.
- The VAX divides events in interrupts and exceptions:

- interrupts may be: device, software, urgent;
- exceptions may be: faults, traps, aborts.

While there are different names to describe the same event, it is important to keep in mind that they all describe the same thing; actions in every case might be different, but what is common is that they all require a transfer in the flow of control.

6.2 A Classification of Interrupts

Interrupts have received different names in the machines coming from different vendors because they wanted to emphasize different characteristics of various events. We can see every interrupt as being characterized by one of the following:

1. **User request v. system**
   The user may have the possibility to request an interrupt as is the case with breakpoints or calling operating system services. Most of the interrupts come however from the system;

2. **User maskable v. nonmaskable**
   There are interrupts the user may disable or enable in the program, such as the breakpoints or arithmetic overflows. Other interrupts must always determine a response from the CPU, as I/O requests or hardware failure events; these are said to be nonmaskable.

3. **Within v. between instructions**
   Some events require a very quick response, they must be treated at once, as is the case with a power failure interrupt, an overflow or a memory page fault. Other interrupts may be treated at the end

of instructions (i.e. between instructions) as happens with I/O requests or breakpoints.

4.Synchronous v. asynchronous

An interrupt is said to be synchronous if it occurs at the same place, every time the program is executed with the same data and memory allocation. This is the case with arithmetic overflows, breakpoints, page faults or undefined instructions. Asynchronous interrupts are those that occur unexpectedly, without any time relation to the program being executed, as happens with hardware problems, power failures, or I/O requests.

5. Resume v. terminate

If the program stops after the interrupt then we have a terminating event; this is the case for example with power failures, hardware troubles or undefined instructions.

In all cases the status of the program must be saved, another program (an interrupt handler) must be invoked to solve the problem that caused the interrupt, and finally the program's state restored and the program restarted.

6.3 Checking for interrupts

We have to modify the state-diagram of the Control Unit to check for interrupts; we also have to update the hardware to provide support for interrupts handling. At this point we should make a major decision: do we want to accept within instruction interrupts or only between instructions? This is a major decision because the two options are very different in complexity. To decide this we must have very clear in mind what kind of system de we want. If the system has to have virtual memory, then there is no way around, we must accept interrupts within instructions: any memory access may yield a page fault interrupt. We'll come back to virtual memory in chapter 10, but to ease the understanding, here is a small introduction:

Our machine has 32bit addresses; this means 2^{32} objects (whatever they are, depending upon organization, bytes, half-words or words) can be referred. The system's main memory is much smaller, in the range of MBytes. The virtual memory creates the user the illusion of having that much memory as the address says, 4GBytes. For this, the external memory is used, basically the hard disk. The object referred by some address may reside at a given moment in the main memory, or it may be somewhere on the disk. In the latter case an access to this object requires a much longer time (milliseconds) than it would if the object was in the main memory (hundreds of nanoseconds at most), due to disk latency. In such a situation the memory subsystem (more precisely the Memory management Unit) issues a page fault interrupt, and the CPU has to start a sequence of operations that will eventually the object required in the main memory. When it is here the instruction that led to the page fault, addresses the memory again, with the same address, and will, this time, succeed.

It is easy to see that the page fault interrupt must be treated at the very moment it appears; there is no way around this. On the other hand, if we are designing a less sophisticated CPU, which is not meant to support a virtual memory system, nor is it meant to be pipelined, then the simpler approach of interrupt handling can be taken, i.e. accepting interrupts only between instructions. As a matter of fact this has been the case from the early microprocessors to many of the recent ones (the Intel 80x86 family for example). With the emergence of RISC architectures in the early '80s with their emphasis on efficient implementation (pipelining), things have changed dramatically: interrupts must be treated when they appear, not after the instruction has terminated.

Because one of our goals is to have an easy to implement architecture, and we have chosen the Instruction Set in such a way that a pipelined implementation is possible, we will discuss this latter approach in dealing with interrupts. This is also the major trend in architectures today.The basic actions the Control Unit must take when an interrupt occurs are:

- save the current PC (the address of the interrupted instruction) in the Interrupt Address Register (IAR on Figure 2.2), also called Exception Program Counter

- transfer to the operating system at some specified address (call an interrupt handler)

At the return from the interrupt handler, the content of the IAR is used to restart the instruction.

The operating system must know, besides the instruction which caused the interrupt, the reason for that interrupt: was it an overflow, or maybe a page fault? There are two ways to keep track of the reason for an interrupt:

- use a **status register** in which every kind of interrupt writes its code, before control gets to the Operating System

- **vectored interrupts:** in this approach every kind of interrupt forces a call to a different address, the address where its interrupt handler (or at least its beginning) is located.

The use of a status register for interrupts requires only an entry point for all possible interrupts (a single call address); the interrupt handler then decodes the content of the status register to execute the proper sequence of code. Assuming a vectored implementation (a status register implementation is slightly more difficult in that it requires at least one extra instruction that accesses the content of the status register) what we need, as hardware, is a way to load the PC with the address for the interrupt: this means a small “table” with addresses must be kept in the Control Unit, and a multiplexer has to be provided to select between the bus Dest (see Figure 2.2) and this table. These addresses are fixed at the moment the circuit is designed, and there must be as many as different kinds of interrupts.

With all this in mind we can now modify the state-diagram for the Control Unit to include interrupts. Figure 6.1 presents the initial part of the diagram, the instruction fetch and decode. Pending interrupts must be treated at this moment, i.e. the between instructions interrupts, page fault interrupt from the instruction read cycle, and wrong instruction at the end of a decoding cycle. A wrong instruction may appear if, somehow, the content of the code area in the memory has been overwritten or if an instruction is fetched from another part of the memory that the code area.

Figure 6.2 presents the part of the state-diagram corresponding to the add and load instructions. Arithmetic instructions must test for overflow, while load/store must test for page fault interrupt.

We must define two new instructions, one to allow user defining a software interrupt, and another one to return from the exception handler. The two instructions are:

- trap: transfer control to a vectored address (fixed by design), it saves in IAR the PC of the next instruction, then loads into PC the vector address;

- rex (return from exception): used to return from an interrupt (exception) handler to the main program. It restores the processor's status word (which was saved by trap) and then load PC with the content of IAR;

Note here that the mechanism we use to return from subroutines:

```
jr r31
```

cannot be used in this case; the trap instruction does more than an usual jal instruction and it must have a corresponding return instruction (rex).

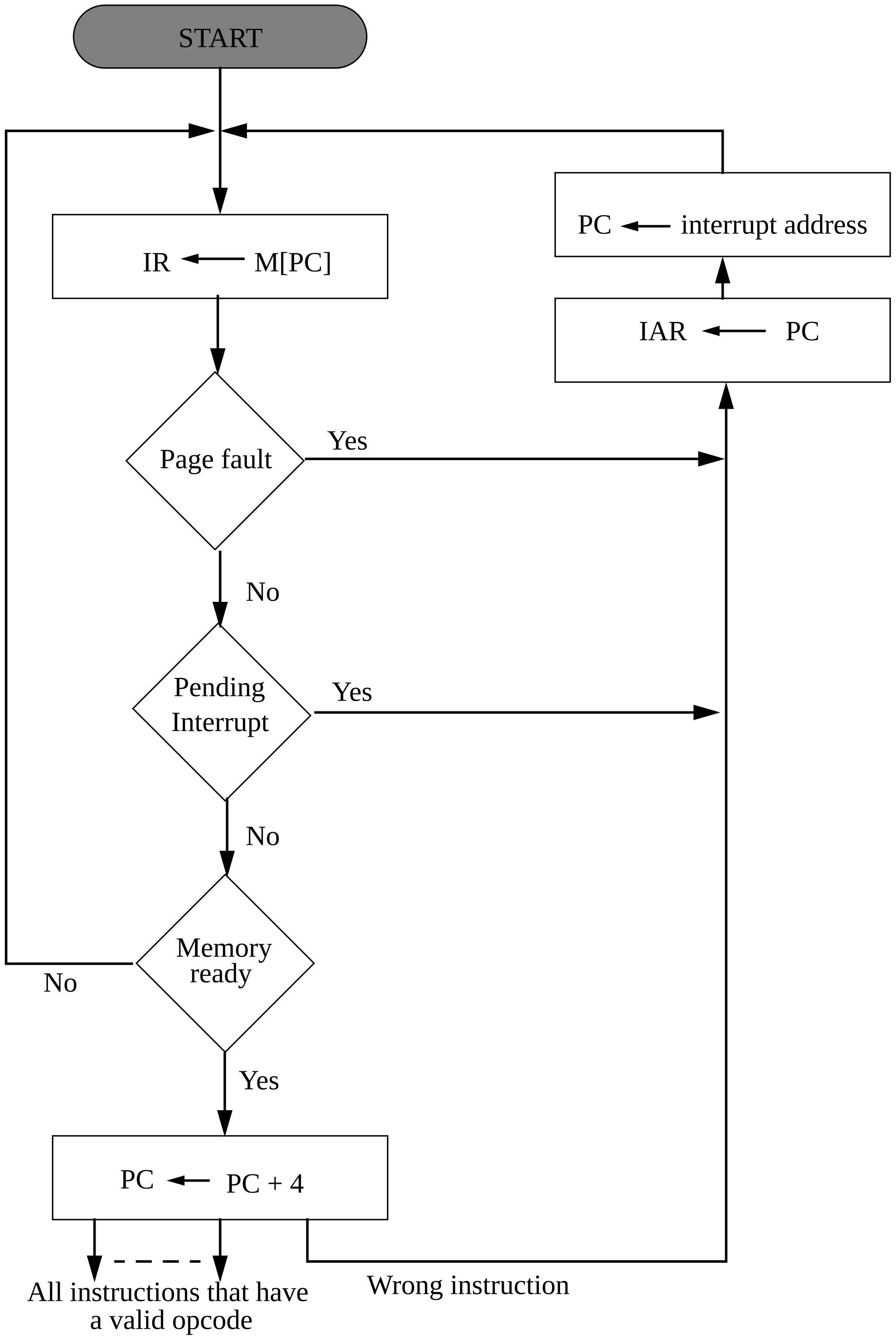

Figure 6.1 Including interrupts in the state-diagram of the control unit. Only the initial part, corresponding to instruction fetch is shown.

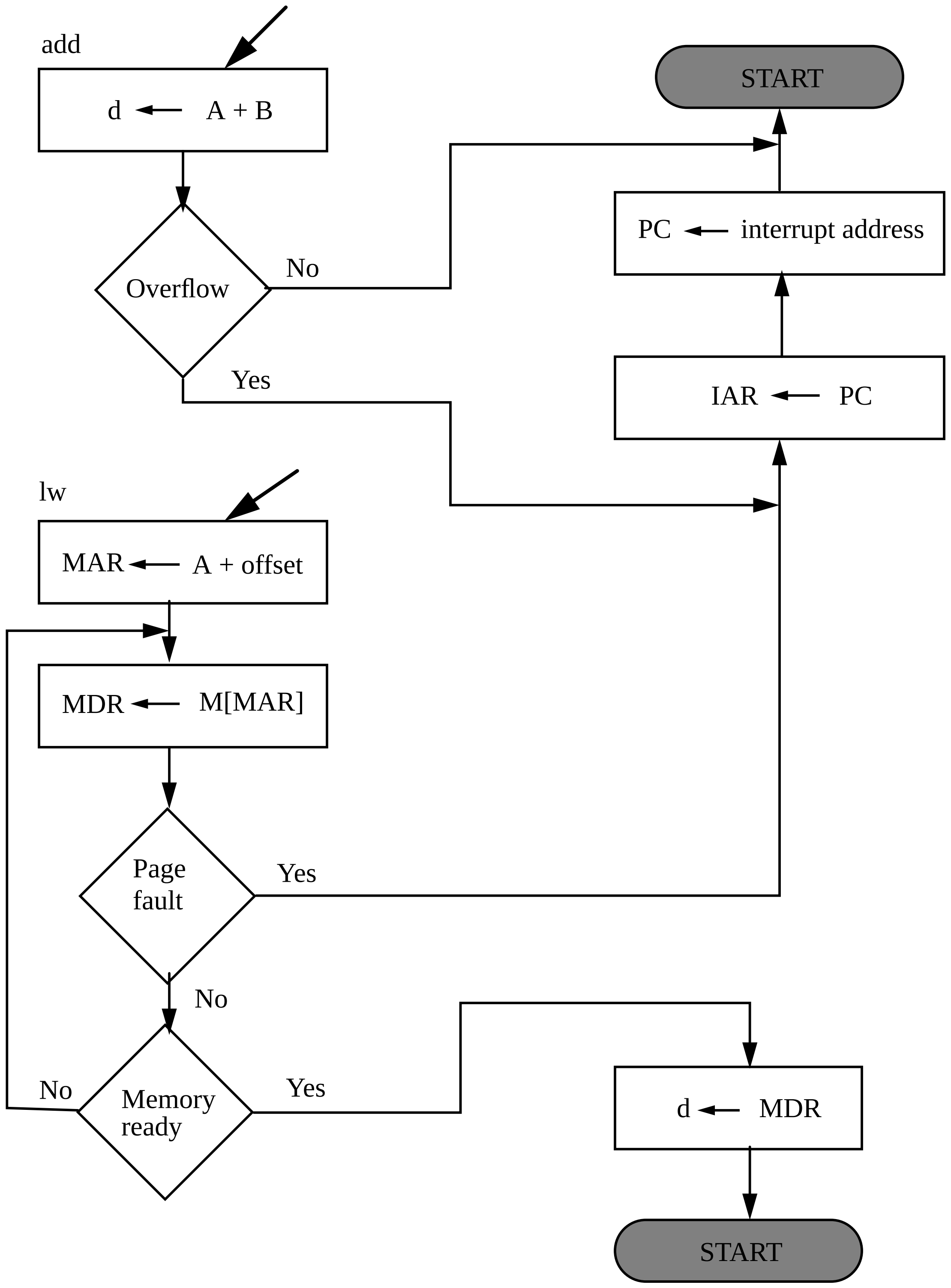

Figure 6.2 Testing for interrupts in the body of the state-diagram. Only add and lw are represented.

Registers are considered to be a part of the program status, and we must take care that the exception handler does not change their content; however the execution of this piece of code requires working with registers, both for addressing (the only addressing modes we have are immediate and based), and for computation. Let's save some registers to make room for the handler! This can be done indeed using the run time stack and the register designed to hold the stack pointer. Sometimes the handler is very simple, and saving registers at the beginning of the handler and restoring then at the end is expensive. For these reasons we could reserve one or two registers in our register set for the exclusive use of the operating system.

6.4 Some problems in checking for interrupts

Even though checking for interrupts does not add very many states to the original state diagram, it is worth noting that its complexity increases due to the many tests that appear. Instead of unconditional state transitions we now have almost everywhere conditional transitions; the combinatorial circuit gets bigger as a result. Clearly Figure 6.1 and Figure 6.2 are very schematic, and here is the reason:

- In the case of an interrupt the address of the current instruction must be saved in the IAR, i.e. the PC corresponding to the current instruction. For interrupts that occur in the instruction fetch stage this is true because PC has not been incremented yet.

- After the instruction decode state the PC holds the address of the following instruction ($PC \leftarrow PC+4$). For all interrupts that occur after this state we must save PC-4 into IAR. Hence there must be different states in the state diagram to reflect this situation.

Another serious problem that appears is that arithmetic operations change the state of registers no matter if an interrupt occurs or not. Taking a closer look at Figure 6.2 we see that the result is written into the destination register in the same state the operation is performed, and only then the overflow is tested. This seems to be unimportant; however some architectures require the instructions to have no effect if interrupts occur (this happens with MIPS for instance). That means that the state of the machine must be left precisely the same as it was at the beginning of the instruction.

Example 6.1 ARITHMETIC OPERATIONS:

Consider the following sequence of code:

```
..........................
add r1, r1, r2
..........................
```

and explain what happens if there is an overflow at the addition.

Answer:

The result of the addition (erroneous due to the overflow) is written into r1 at the end of the add cycle. The address of this instruction is saved into the IAR and, after the exception is handled, the system returns, trying to execute this instruction again. But at this moment the content of r1 is no longer what it was during the first addition attempt!

As the above example points out, the problem is not that simple as it seemed to. We probably need to modify the state-diagram in such a way that the result of an arithmetic operation is not directly written in the destination until after the overflow test is performed. This requires some extra hardware, a register to hold the result until after the overflow test is performed, and oddly enough, more states in the state-diagram thus increasing the CPI.

An alternative solution to this problem is to have a set of alternate registers that hold copies of values the regular registers had at the beginning of instruction. These registers are called *shadow registers*, and the interrupt handler uses them to restore the state of the CPU as it was before the interrupted instruction.

Our design makes an attempt to handle three kind of interrupts; this changes in a rather substantial way the initial design: more hardware is needed, and a the state-diagram has to be redrawn. It should be clear by now why the design of the Control Unit is the most challenging part of the project: it must handle the variety of interactions between interrupts and instructions while remaining small enough and fast.

Handling the interrupts is not done only in hardware nor only in software. The design must provide sufficient hardware support for an efficient software manipulation.

6.5 What is Really Hard About Interrupts

The example we had in section 6.4 showed what kind of problems can arise from interrupts that occur in the middle of execution. Many instructions require that the machine has to be restored in the state it was before starting to execute the faulting instruction.

A system that meets the above requirement is said to be *restartable*.

As we suggested, our design can be modified to be made restartable, though there is a heavy price paid for this, the loss in performance. This explains why supecomputers are not restartable.

The most difficult problems arise with instructions that modify the machine's state before it is known that an interrupt might occur. Obviously the more complicated an instruction is, and the more time it takes to execute, the greater are chances to modify the state before an interrupt occurs. As an example consider the MOVC3 instruction on the VAX. Its format is:

```
MOVC3 LEN, S, D
```

and it copies LEN bytes starting at address S, to LEN bytes starting at address D. The length LEN is specified as a 16 bit integer; that means it can move up to 2^{16} bytes before ending. Moving a byte every 500 nanoseconds it will take 32.7 milliseconds for this instruction to complete. Clearly we can not ask from this instruction to restart after running for milliseconds. Not only would be this wasteful but, after restarting, a new interrupt could occur forcing a new restart, and so on with very little chances for this instruction to ever terminate. On the other hand, interrupts may not be disabled for milliseconds without the risk of losing important events. Special techniques have been developed to manage interrupts for such long running instructions.

This example suggests why most modern architectures have shifted towards simple instructions executing in a couple of clock cycles. Easing the handling of interrupts is yet another reason for this move.

6.6 A Case Study: Interrupts in MIPS

MIPS is the name of one of the first RISC machines, introduced in 1981 at Stanford. The MIPS architecture provides a simple mechanism for exceptions. The CPU operates in one of two modes, **kernel** or **user**. In user mode only the 32 general purpose integer registers, and the 32 floating point registers are available for read/write operations. In kernel mode an extra set of special purpose registers is available, and the content of those registers can be modified. In this mode addresses that are otherwise inaccessible (those having the MSB = 1), can be accessed and it is here where the exception handler resides, as well as data accessible only to the operation system.

Two instructions, mtc0 and mfc0, allow moving data between the general registers and the special set of registers. mtc0 which moves a general register into a special purpose one works only in kernel mode. The registers which are of interest at this moment are:

- Status Register;
- Cause Register;
- Exception Program Counter (EPC).

The Status Register has the following structure:

31-16	15	14	13	12	11	10	9	8	7-6	5	4	3	2	1	0
....	IM7	IM6	IM5	IM4	IM3	IM2	IM1	IM0	..	KUo	IEo	KUp	IEp	KUc	IEc

The signification of bits in the register is as follows:

- IM7 to IM2: interrupt masks for hardware interrupts
- IM1 to IM0: interrupt masks for software interrupts
- KUo: the **old** Kernel/User bit
- IEo: the old Interrupt Enable bit
- KUp: the **previous** Kernel/User bit
- IEp: the previous Interrupt Enable bit
- KUc: the **current** Kernel/User bit
- IEc: the current Interrupt Enable bit.

When an interrupt occurs, the operating mode, kernel or user, is saved in the KUc after KUp has been moved into KUo, and KUc into KUp. In other words when an exception occurs the *previous* state is saved as the *old* state and the *current* state is saved as the *previous* state; the *old* state is lost. When the rfe instruction is executed the *previous* state becomes the *current* state and the *old* state becomes the *previous*. It is easy to see that the Status Register implements a simple three level stack for the KU and IE bits. The current executing mode is saved in the Status Register such that the correct executing mode is restored after the execution of an rfe instruction (return from exception); an exception may occur while working in kernel mode.

When the exception occurs the system disables all interrupts, thus preventing the exception handler from being interrupted again, before it has any chance to save the sensitive information, as the EPC, the Cause

Register, the Status Register and other registers it will be using. After all this information has been saved, interrupts may be enabled by setting IRc to one. The rfe instruction also enables interrupts if they were enabled at the moment when the interrupt occurred.

The Interrupt Masks allow enabling/disabling individual interrupts. For an interrupt to be enabled both IEc and IMi (i = 0..7) must be enabled (set to one).

The Cause Register has the following bit allocation:

31-16	15	14	13	12	11	10	9	8	7	6-2	1	0
0	IP7	IP6	IP5	IP4	IP3	IP2	IP1	IP0	0	ExcCode	0	0

The meaning of bits in the Cause Register is as follows:

- IP7 to IP2: hardware Interrupt Pending flag;
- IP1 to IP0: software Interrupt Pending flag;
- ExcCode: the Exception Code; an unsigned integer which indicates the class of interrupt that occurred.

The flag IPi (i == 0..7) is set to one when an interrupt occurs on the interrupt line i. However, no action is being taken unless both IEc and IMi are enabled (set to one).
Some of these exception codes can be seen in the table below:

ExcCode	Mnemonics	Meaning
0	Int	Interrupt
4	AdEL	Load from an illegal address
5	AdES	Store to an illegal address
6	IBE	Bus error on instruction fetch
7	DBE	Bus error on data reference
8	Sys	Syscall instruction
9	Bp	Break instruction
10	RI	Reserved instruction
11	CpU	Coprocessor unavailable
12	Ov	Overflow
15	FPE	Floating-point exception

The MIPS architecture fixes a single address, 0x80000080, as the address where the control is transferred when an exception occurs.The instruction syscall permits user exceptions.

Exercises

6.1 Let's suppose that the ideal CPI of our machine is 4 (calculated with some instruction mix), and we want to take into account the effect of interrupts. There is an interrupt every millisecond (from the real-time clock) and the clock cycle of the machine is 20 ns. Servicing the interrupt requires 50 cycles. What is the new value of CPI in this case, and how does the initial performance compare with the new one? Redo your computations if interrupts are coming at a 10 microseconds rate.

6.2 Suppose we have to modify the state-diagram for our Control Unit to make interrupts restartable (i.e. after an interrupt is handled the instruction is restarted). What is the new CPI for every instruction? Is there any special hardware requirement for this?