3.3.5 Reduced instruction set computing (RISC) processors

ISAs

The **instruction set** or the **instruction set architecture (ISA)** is the set of basic instructions that a processor understands. The instruction set is a portion of what makes up architecture.

Historically, the first two philosophies to instruction sets were: reduced (RISC) and complex (CISC). The merits and argued performance gains by each philosophy are and have been thoroughly debated.

CISC

Complex Instruction Set Computer (CISC) is rooted in the history of computing. Earlier developments were based around the idea that making the CPU more complex and supporting a larger number of potential instructions would lead to increased performance. This idea is at the root of CISC processors, such as the Intel x86 range, which have very large instruction sets reaching up to and above 300 separate instructions. They also have increased complexity in other areas, with many more specialized **addressing modes** and **registers** also being implemented.

The complexity of the processor hardware and architecture that was built for CISC meant that chips were difficult to understand and program for, it also meant that they were very expensive to produce.

RISC

Reduced Instruction Set Computer (RISC) was realized in the mid-1980s by IBM.

It was the beginning of the RISC philosophy. The idea here was that the best way to improve performance would be to simplify the processor workings as much as possible. RISC processors, such as the IBM PowerPC processor, have greatly simplified and reduced the **instruction set,** numbering in the region of one hundred instructions or less. Addressing modes are simplified back to **four or less,** and the length of the codes is fixed to allow standardization across the instruction set.

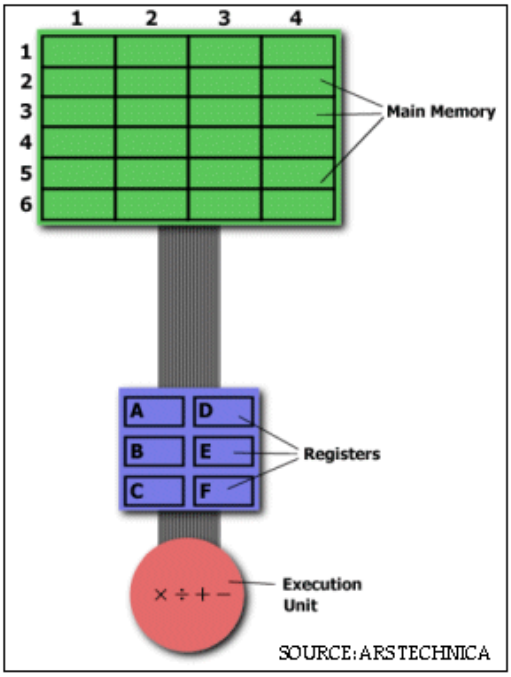
3.3.5 Reduced instruction set computing (RISC) processors

Changing the architecture to this extent means that fewer transistors are used to produce the processors. This means that RISC chips are much cheaper to produce than their CISC counterparts. Also the reduced instruction set means that the processor can execute the instructions more quickly, potentially allowing for greater speeds.

However, only allowing such simple instructions means a greater burden is placed upon the software itself. Less instructions in the instruction set means a greater emphasis on the efficient writing of software with the instructions that are available.

Multiplying Two Numbers in Memory

On the right is a diagram representing the storage scheme for a generic computer. The main memory is divided into a 6 x 4 square. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the 6 registers. Let's say we want to find the product of 2 numbers – one stored in location 2:3 and another stored in location 5:2 – and then store the product back in the location 2:3.



The CISC Approach

The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (MULT). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit. And then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with a single instruction:

MULT 2:3, 5:2

MULT is what is known as a "complex instruction" It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we say "a equals whatever is stored in 2:3" and "b equals whatever is stored in 5:2" then this command is identical to the C statement "a = a * b"

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively

short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands:

"LOAD" which moves data from the memory bank to a register.

"PROD" which finds the product of two operands located within the registers.

"STORE" which moves data from a register to the memory banks.

In order to perform the exact series of steps described in the CISC approach, a programmer would need to code 4 lines of assembly:

LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A

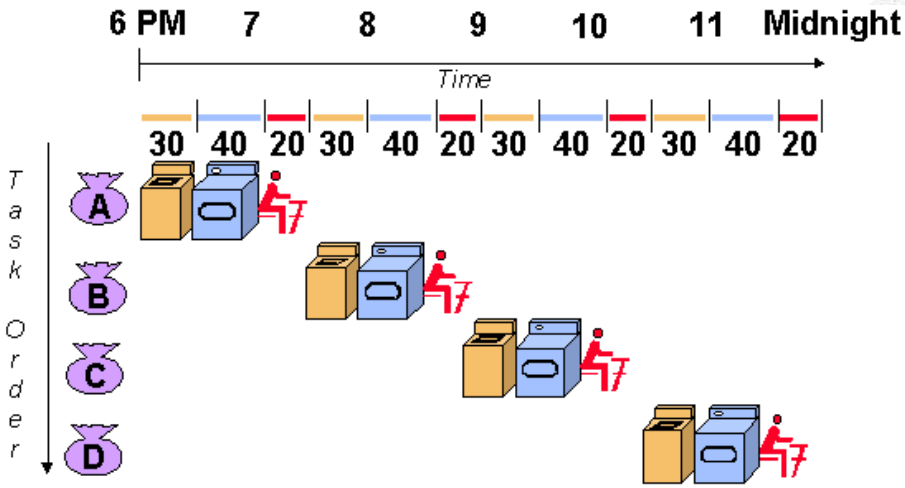
At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code so more RAM is needed to store assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

CISC Processors	RISC Processors
1. Complex Instruction Set Computer (CISC) processors has a bigger instruction set with many addressing modes.	1.Reduced Instruction Set Computers (RISC) processors have a smaller instruction set with few addressing modes.
2. It has to use a separate micro-programming unit with a control memory to implement complex instructions.	2.It has a hard-wired programmed-unit without a control memory, and separate hardware to implement each and every instruction.
3. An easy compiler design	3. A complex compiler design.
4. The calculations are slower and precise	4The calculations are faster and precise
5. Decoding of instruction is complex	5. Decoding of instruction is simple
6.The Execution time is very high.	6. It takes very less execution time.

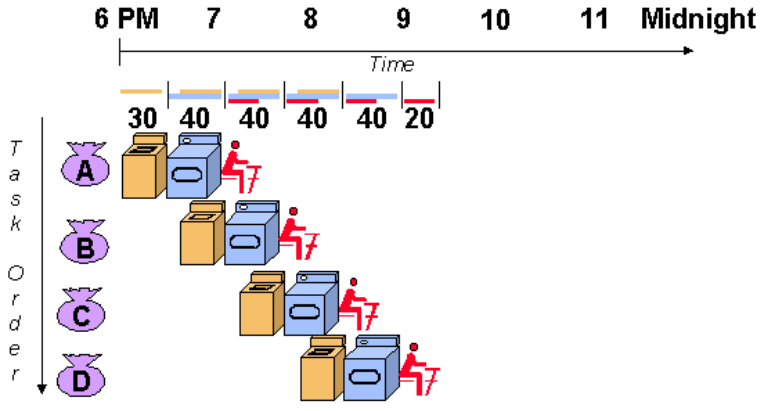
How Pipelining Works

Pipelining, a standard feature in RISC processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time, more instructions can be executed in a shorter period of time.

A useful method of demonstrating this is the laundry analogy. Let's say that there are four loads of dirty laundry that need to be washed, dried, and folded. We could put the first load in the washer for 30 minutes, dry it for 40 minutes, and then take 20 minutes to fold the clothes. Then pick up the second load and wash, dry, and fold, and repeat for the third and fourth loads. Supposing we started at 6 PM and worked as efficiently as possible, we would still be doing laundry until midnight.



However, a smarter approach to the problem would be to put the second load of dirty laundry into the washer after the first was already clean and whirling happily in the dryer. Then, while the first load was being folded, the second load would dry, and a third load could be added to the pipeline of laundry. Using this method, the laundry would be finished by 9:30.



RISC Pipelines

A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of steps, they are basically variations of these five, used in the MIPS R3000 processor:

1. Fetch instructions from memory
2. Read registers and decode the instruction
3. Execute the instruction or calculate an address
4. Access an operand in data memory
5. Write the result into a register

If you glance back at the diagram of the laundry pipeline, you'll notice that although the washer finishes in half an hour, the dryer takes an extra ten minutes, and thus the wet clothes must wait ten minutes for the dryer to free up. Thus, the length of the pipeline is dependent on the length of the longest step. Because RISC instructions are simpler than those used in CISC processors, they are more conducive to pipelining. While CISC instructions varied in length, RISC instructions are all the same length and can be fetched in a single operation. Ideally, each of the stages in a RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction each clock cycle and averages one cycle per instruction (CPI).

Pipeline Problems

In practice, however, RISC processors operate at more than one cycle per instruction. The processor might occasionally stall as a result of data dependencies and branch instructions.

A data dependency occurs when an instruction depends on the results of a previous instruction. A particular instruction might need data in a register which has not yet been stored since that is the job of a preceding instruction which has not yet reached that step in the pipeline.

For example:

```
add $r3, $r2, $r1
add $r5, $r4, $r3
……more instructions that are independent of the first two
```

In this example, the first instruction tells the processor to add the contents of registers **r1** and **r2** and store the result in register **r3**. The second instruction tells processor to add **r3** and **r4** and store the sum in **r5**. We place this set of instructions in a pipeline. When the second instruction is in the second stage, the processor will be attempting to read **r3** and **r4** from the registers. Remember, though, that the first instruction is just one step ahead of the second, so the contents of **r1** and **r2** are being added, but the result has not yet been written into register **r3**. The second instruction therefore

cannot read from the register **r3** because it hasn't been written yet and must wait until the data it needs is stored. Consequently, the pipeline is **stalled** and a number of empty instructions **(known as bubbles)** go into the pipeline. Data dependency affects long pipelines more than shorter ones since it takes a longer period of time for an instruction to reach the final register-writing stage of a long pipeline.

MIPS' solution to this problem is **code reordering**. If, as in the example above, the following instructions have nothing to do with the first two, the code could be rearranged so that those instructions are executed in between the two dependent instructions and the pipeline could flow efficiently. The task of code reordering is generally left to the compiler, which recognizes data dependencies and attempts to minimize processor stalls.

Branch instructions are those that tell the processor to make a decision about what the next instruction to be executed should be based on the results of another instruction. Branch instructions can be troublesome in a pipeline if a branch is conditional on the results of an instruction which has not yet finished its path through the pipeline.

For example:

```
Loop : add $r3, $r2, $r1
       sub $r6, $r5, $r4
       beq $r3, $r6, Loop
```

The example above instructs the processor to add **r1** and **r2** and put the result in **r3**, then subtract **r4** from **r5**, storing the difference in **r6**. In the third instruction, **beq** stands for **“branch if equal”**. If the contents of **r3** and **r6** are equal, the processor should execute the instruction labeled "Loop." Otherwise, it should continue to the next instruction. In this example, the processor cannot make a decision about which branch to take because neither the value of **r3** or **r6** have been written into the registers yet.

Pipelining Developments

In order to make processors even faster, various methods of optimizing pipelines have been devised.

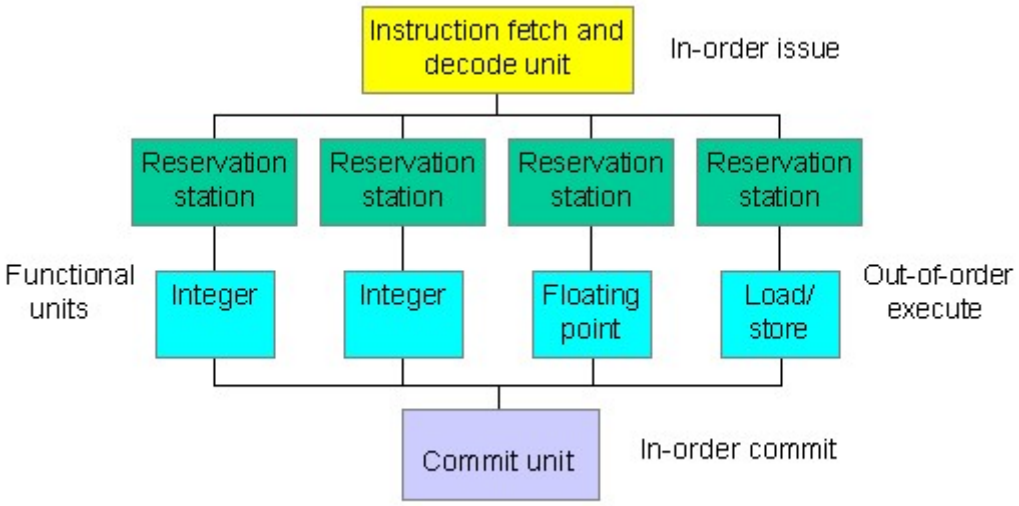
Superpipelining refers to dividing the pipeline into more steps. The more pipe stages there are, the faster the pipeline is because each stage is then shorter. Ideally, a pipeline with five stages should be five times faster than a non-pipelined processor (or rather, a pipeline with one stage). The instructions are executed at the speed at which each stage is completed, and each stage takes one fifth of the amount of time that the non-pipelined instruction takes. Thus, a processor with an 8-step pipeline (the MIPS R4000) will be even faster than its 5-step counterpart. The MIPS R4000 chops its pipeline into more pieces by dividing some steps into two. Instruction fetching, for example, is now done in two stages rather than one. The stages are as shown:

1. Instruction Fetch (First Half)
2. Instruction Fetch (Second Half)
3. Register Fetch
4. Instruction Execute
5. Data Cache Access (First Half)
6. Data Cache Access (Second Half)
7. Tag Check
8. Write Back

Superscalar pipelining involves multiple pipelines in parallel. Internal components of the processor are replicated so it can launch multiple instructions in some or all of its pipeline stages. The RISC System/6000 has a forked pipeline with different paths for floating-point and integer instructions. If there is a mixture of both types in a program, the processor can keep both forks running simultaneously. Both types of instructions share two initial stages (Instruction Fetch and Instruction Dispatch) before they fork. Often, however, superscalar pipelining refers to multiple copies of all pipeline stages (In terms of laundry, this would mean four washers, four dryers, and four people who fold clothes). Many of today's machines attempt to find two to six instructions that it can execute in every pipeline stage. If some of the instructions are dependent, however, only the first instruction or instructions are issued.

Dynamic pipelines have the capability to schedule around stalls. A dynamic pipeline is divided into three units: the instruction fetch and decode unit, five to ten execute or functional units, and a commit unit. Each execute unit has reservation stations, which act as buffers and hold the operands and operations.

3.3.5 Reduced instruction set computing (RISC) processors



While the functional units have the freedom to execute out of order, the instruction fetch/decode and commit units must operate in-order to maintain simple pipeline behavior. When the instruction is executed and the result is calculated, the commit unit decides when it is safe to store the result. If a stall occurs, the processor can schedule other instructions to be executed until the stall is resolved. **This, coupled with the efficiency of multiple units executing instructions simultaneously, makes a dynamic pipeline an attractive alternative.**

How do RISC and CISC processors handle interrupts?

Introduction

an **interrupt** is a signal sent to the processor from a hardware device, indicating that the device requires attention. One is sent, for example, when a key has been pressed or when one of the software needs updating. This sending of a signal is known as an interrupt request.

RISC OS deals with the interrupt by temporarily halting its current task, and entering an **interrupt routine.** This routine deals with the interrupting device quick enough that you will never realize that your program has been interrupted.

Interrupts provide a very efficient means of control since the processor doesn't have to be responsible for regularly checking to see if any hardware devices need attention. Instead, it can concentrate on executing your code or whatever else its current main task may be, and only deal with hardware devices when necessary.