RISC Processors

Instruction Set Architecture

"Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct program for that machine."

- The ISA defines:
  - Operations that the processor can execute
  - Data Transfer mechanisms + how to access data
  - Control Mechanisms (branch, jump, etc)
  - "Contract" between programmer/compiler and HW

- ISA is important:
  - Not only from the programmer's perspective.
  - From processor design and implementation perspectives as well.

Instruction Set Architecture

- Programmer visible part of a processor:
  - Registers
  - Addressing Modes
  - Instruction Format
  - Exceptional Conditions
  - Instruction Set

CISC features

- One instruction could do the work of several instructions.
  - For example, a single instruction could load two numbers to be added, add them, and then store the result back to memory directly.
- Many versions of the same instructions were supported;
  - Different versions did almost the same thing with minor changes.
  - For example, one version would read two numbers from memory, and store the result in a register. Another version would read one number from memory and the other from a register and store the result to memory.

Background of RISC

- John Cocke and his colleagues developed simpler ISAs and compilers for minicomputers. As an experiment, they retargeted their research compilers to use only the simple register-register operations and load-store data transfers of the IBM 360 ISA, avoiding the more complicated instructions. They found that programs ran up to three times faster using the simple subset.

- Emer and Clark found 20% of the VAX instructions needed 60% of the microcode and represented only 0.2% of the execution time.

RISC: *Reduced Instruction Set Computer*

A type of microprocessor architecture that utilizes a small, highly-optimized set of instructions

History: The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy.

Design features of RISC processors:

- *one cycle execution time*: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called pipelining;
- *pipelining*: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- *large number of registers*: RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

MIPS

The MIPS processor was developed as part of a VLSI research program at Stanford University in the early 80s.

Professor John Hennessy started the development of MIPS with a brainstorming class for graduate students. The readings and idea sessions helped launch the development of the processor which became one of the first RISC processors, with IBM and Berkeley developing processors at around the same time.

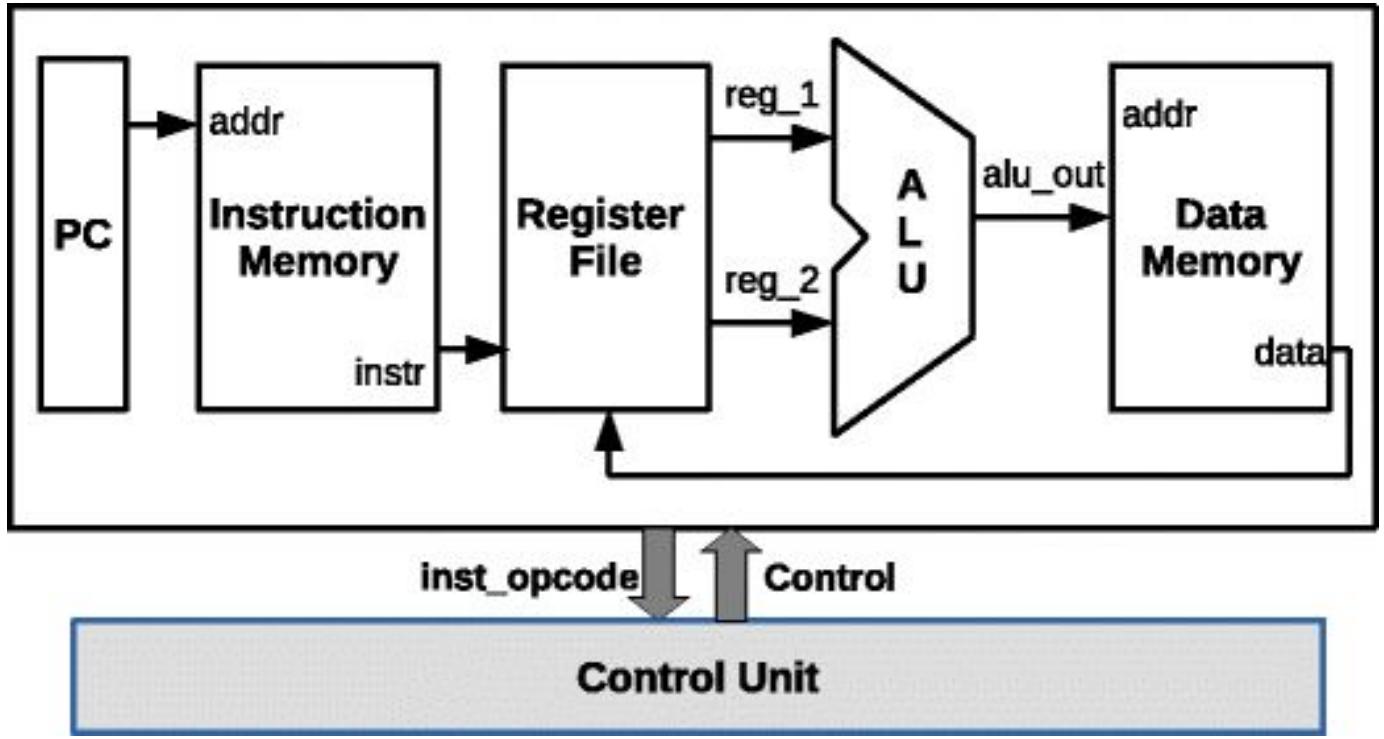
COMPUTER ARCHITECTURE

John Hennessy

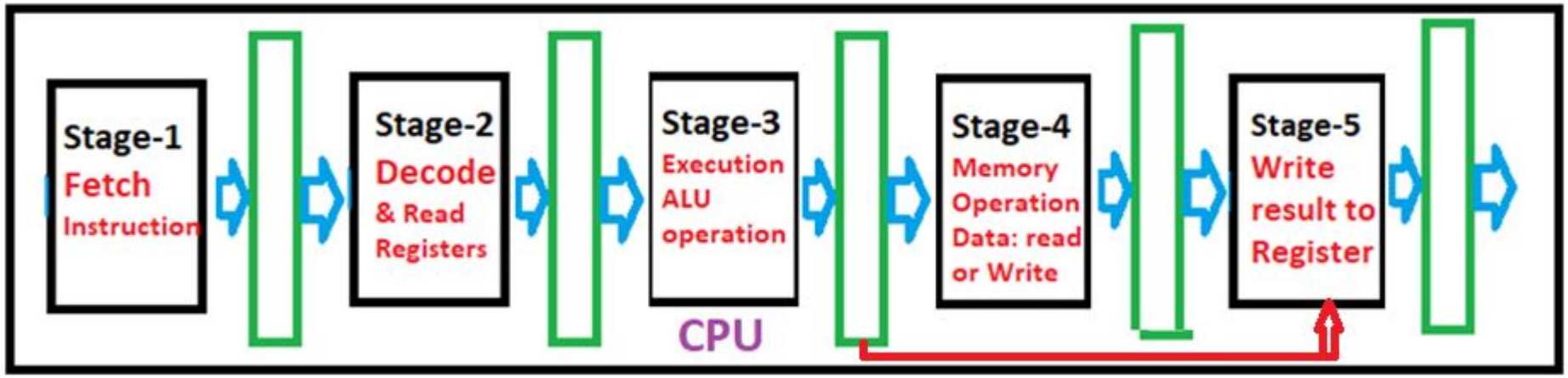
- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

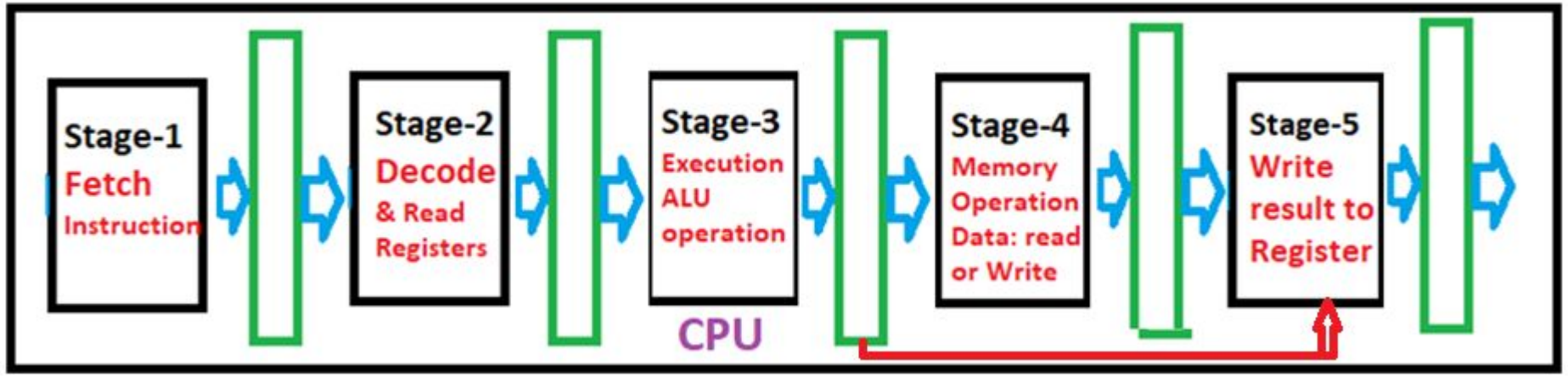
MIPS Architecture

- The Stanford research group had a strong background in compilers, which led them to develop a processor whose architecture would represent the lowering of the compiler to the hardware level, as opposed to the raising of hardware to the software level, which had been a long running design philosophy in the hardware industry.
- Thus, the MIPS processor implemented a smaller, simpler instruction set. Each of the instructions included in the chip design ran in a single clock cycle. The processor used a technique called pipelining to more efficiently process instructions.
- MIPS used 32 registers, each 32 bits wide (a bit pattern of this size is referred to as a *word*).



- Instruction cycle of MIPS processor was subdivided into **five stages**:
- **Instruction Fetch (IF)**
- **Instruction Decode (ID) and Register Read**
- **Execution (EXE)**
- **Memory read/write(MEM)**
- **Write Back** result **(WB)** to Registers





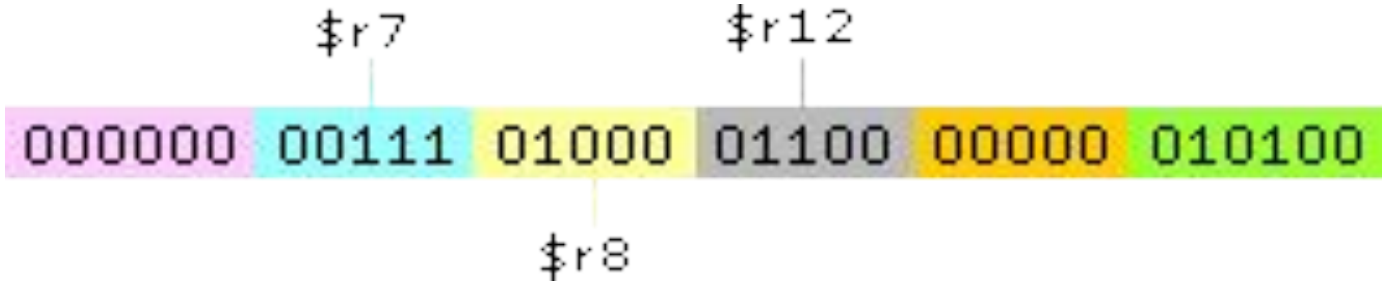
Instruction\ clock cycles	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WB	
Instruction-4				IF	ID	EX	MA	
Instruction-5					IF	ID	EX	
Instruction-6						IF	ID	
Instruction-7							IF	

- Single cycle implementation

Instruction Set

- The MIPS instruction set consists of about **111 total instructions**, each represented in 32 bits. An example of a MIPS instruction is below:

add $r12, $r7, $r8,



- Three-operand arithmetical and logical instructions and all instructions are 32-bit
- Arithmetical and logical instructions are Register based (operands in registers and result will be stored in register)
- 32 general-purpose registers of 32-bits each
- MIPS addition instruction. The instruction tells the processor to compute the sum of the values in registers 7 and 8 and store the result in register 12. The dollar signs are used to indicate an operation on a register. The colored binary representation on the right illustrates the 6 fields of a MIPS instruction. The processor identifies the type of instruction by the binary digits in the first and last fields. In this case, the processor recognizes that this instruction is an addition from the zero in its first field and the 20 in its last field.
- The operands are represented in the blue and yellow fields, and the desired result location is presented in the fourth (purple) field. The orange field represents the *shift amount*, something that is not used in an addition operation.

The instruction set consists of about 111 total instructions. A variety of basic instructions, including:

21 arithmetic instructions (+, -, *, /, %)

8 logic instructions (&, |, ~)

8 bit manipulation instructions

12 comparison instructions (>, <, =, >=, <=, ¬)

25 branch/jump instructions

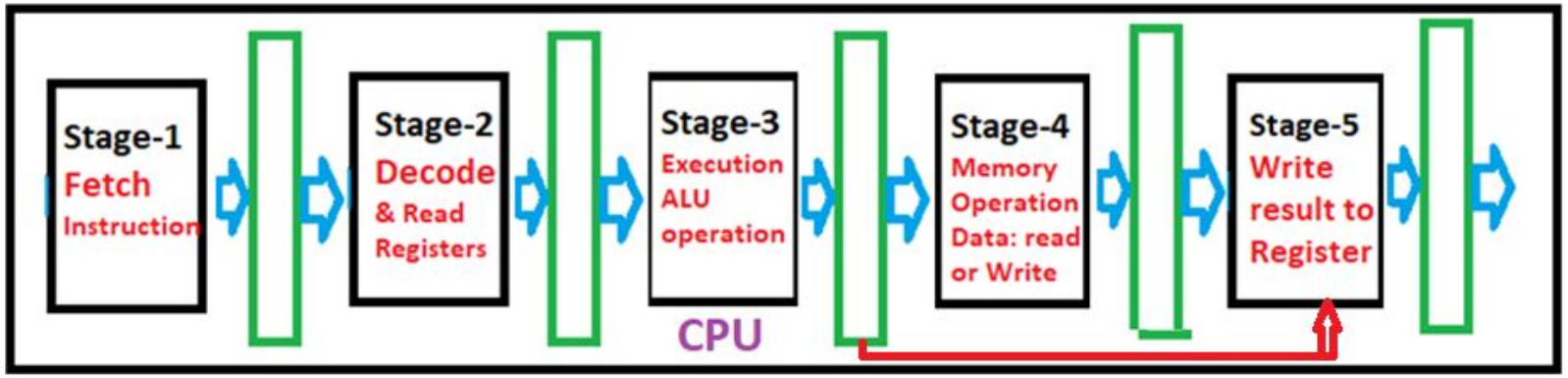
15 load instructions

10 store instructions

8 move instructions

4 miscellaneous instructions

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.

CPI = 1

Instruction\ clock cycles	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WB	
Instruction-4				IF	ID	EX	MA	
Instruction-5					IF	ID	EX	
Instruction-6						IF	ID	
Instruction-7							IF	

Non-pipelined processor (CPI = 5)

Instruction/Clock cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instruction-1	IF	ID	EX	MEM	WB										
Instruction-2						IF	ID	EX	MEM	WB					
Instruction-3											IF	ID	EX	MEM	WB

Features of RISC Processors

- ❑ Small number of instructions
- ❑ Small number of addressing modes
- Large number of registers (>32)
- Instructions execute in one or two clock cycles
- Uniformed length instructions and fixed instruction format.
- Register-Register Architecture:
  - Separate memory instructions (load/store)
- Separate instruction/data cache
- Hardwired control
- Pipelining

CISC features

- One instruction could do the work of several instructions.
  - For example, a single instruction could load two numbers to be added, add them, and then store the result back to memory directly.
- Many versions of the same instructions were supported;
  - Different versions did almost the same thing with minor changes.
  - For example, one version would read two numbers from memory, and store the result in a register. Another version would read one number from memory and the other from a register and store the result to memory.

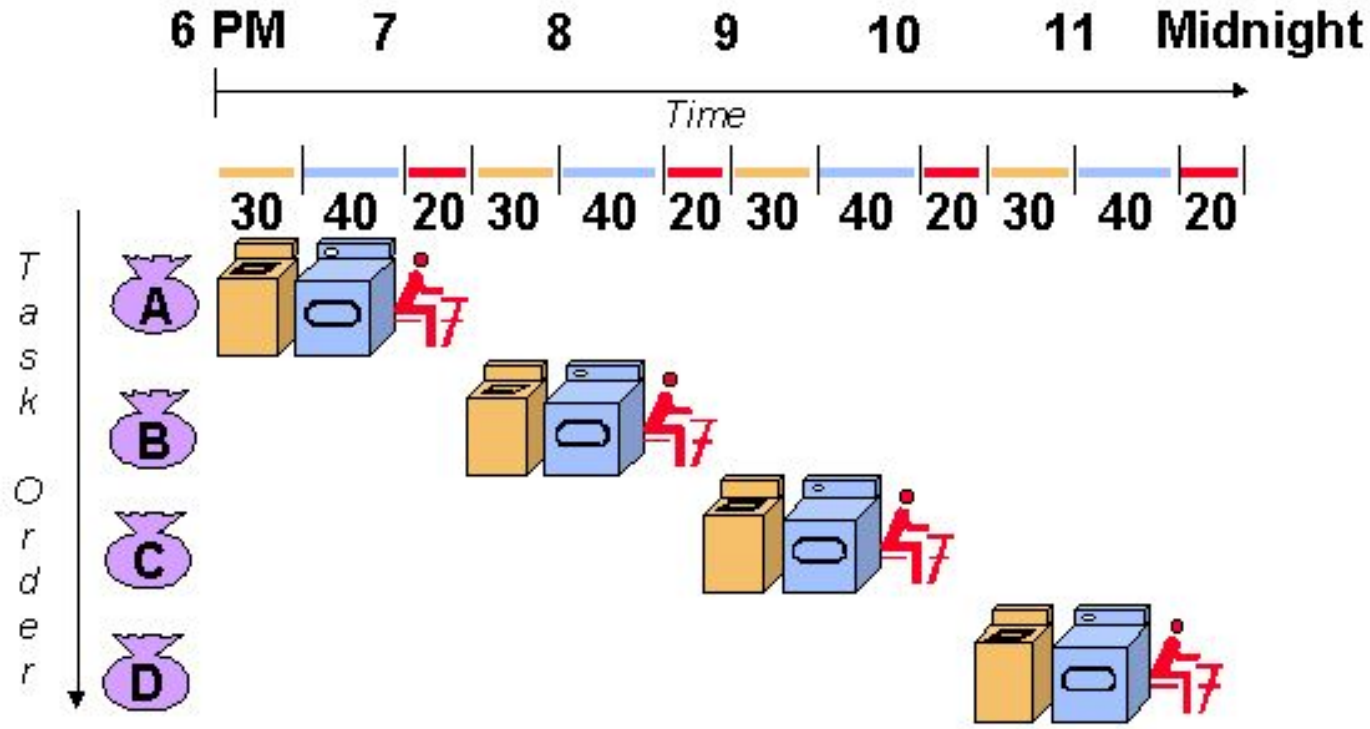
RISC vs CISC

- The formula for processor performance:

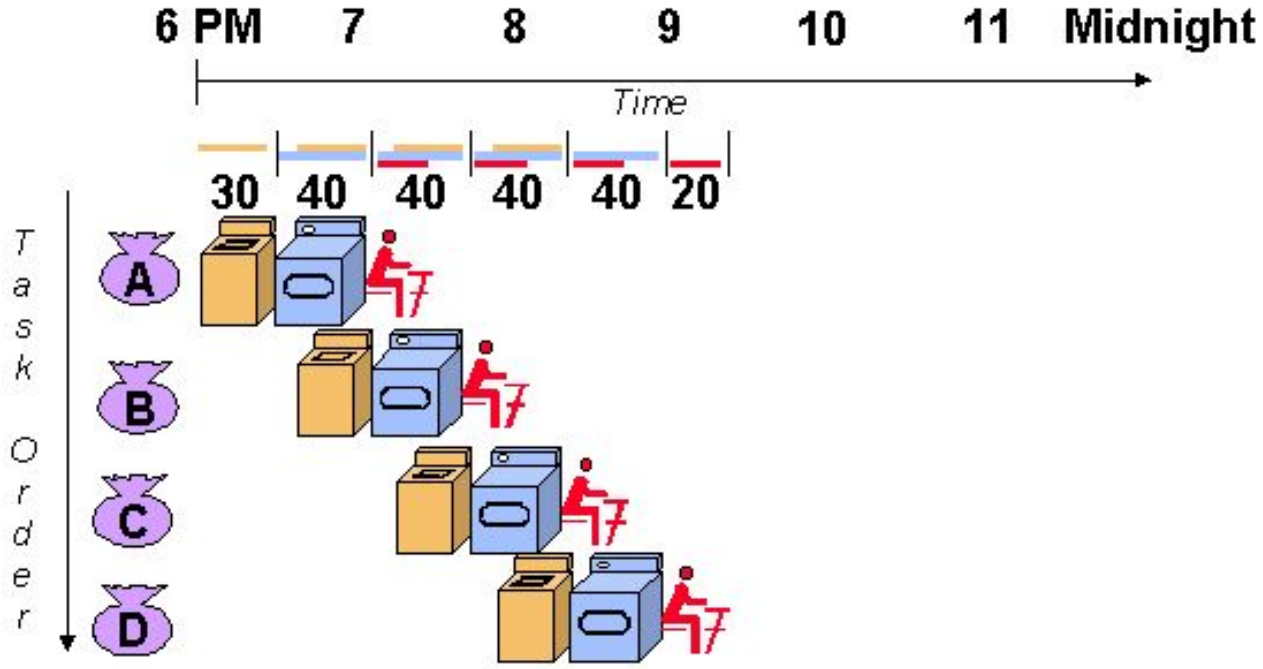
Time/Program = Instructions /Program × (*Clock cycles*) */Instruction* × *Time*/ (*Clock cycle*)

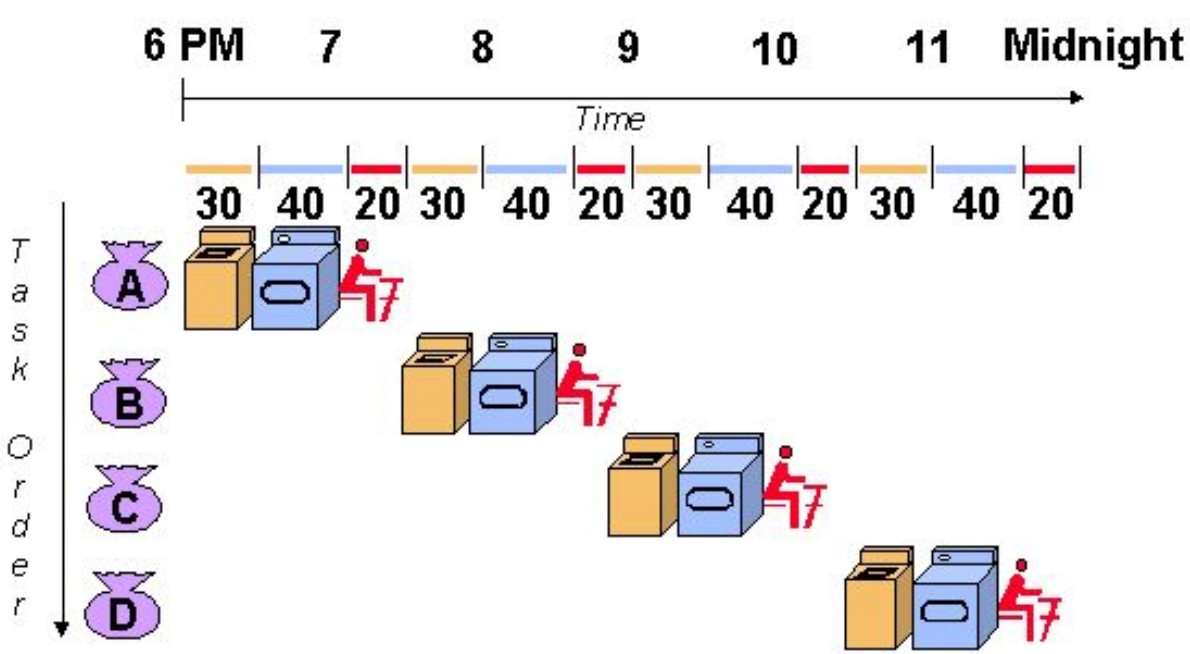
- DEC engineers later showed that the more complicated CISC ISA executed about 75% of the number instructions per program as RISC (the first term), but in a similar technology CISC executed about five to six more clock cycles per instruction (the second term), making RISC microprocessors approximately 4× faster.

- 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:

- fetch instructions from memory
- read registers and decode the instruction
- execute the instruction or calculate an address
- access an operand in data memory
- 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 pre-RISC processors (now called CISC, or Complex Instruction Set Computer), 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 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

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 instructs it 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.

data dependency

- 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 performance stalls.

Branch instructions

- 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.

```
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.

- The processor could stall, but a more sophisticated method of dealing with branch instructions is branch prediction. The processor makes a guess about which path to take - if the guess is wrong, anything written into the registers must be cleared, and the pipeline must be started again with the correct instruction. Some methods of branch prediction depend on stereotypical behavior. Branches pointing backward are taken about 90% of the time since backward-pointing branches are often found at the bottom of loops. On the other hand, branches pointing forward, are only taken approximately 50% of the time. Thus, it would be logical for processors to always follow the branch when it points backward, but not when it points forward. Other methods of branch prediction are less static: processors that use dynamic prediction keep a history for each branch and uses it to predict future branches. These processors are correct in their predictions 90% of the time.
- Still other processors forgo the entire branch prediction ordeal. The RISC System/6000 fetches and starts decoding instructions from both sides of the branch. When it determines which branch should be followed, it then sends the correct instructions down the pipeline to be executed.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

The Performance Equation

- The following equation is commonly used for expressing a computer's performance ability:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \text{ x } \frac{\text{cycles}}{\text{instruction}} \text{ x } \frac{\text{instructions}}{\text{program}}$$

- The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

The Overall RISC Advantage

- Today, the Intel x86 is arguable the only chip which retains CISC architecture. This is primarily due to advancements in other areas of computer technology. The price of RAM has decreased dramatically. In 1977, 1MB of DRAM cost about $5,000. By 1994, the same amount of memory cost only $6 (when adjusted for inflation). Compiler technology has also become more sophisticated, so that the RISC use of RAM and emphasis on software has become ideal.

RISC vs CISC

- The formula for processor performance:

Time/Program = Instructions /Program × (*Clock cycles*) */Instruction* × *Time*/ (*Clock cycle*)

- DEC engineers later showed that the more complicated CISC ISA executed about 75% of the number instructions per program as RISC (the first term), but in a similar technology CISC executed about five to six more clock cycles per instruction (the second term), making RISC microprocessors approximately 4× faster.