RISC Machines*

*From Chapter 2 of High Performance Computing

Complex Instruction Set Computers

The instruction set of a *Complex instruction set computer (CISC)* contains powerful primitives. Close in functionality to high level languages.

Why CISC?

At the time, it was the best choice.

- Compilers didn't generate fast enough code, or use memory well enough.

  Had to save space and time. Meant programming in assembly code.

  High level primitives were easy to understand and work with.

- "Powerful" instructions did multiple tasks with one command. Programmer didn't have to specify all the steps.

  Took up less memory. Loaded faster as only one instruction to fetch from slow main memory.

Problem was, these "powerful" instructions meant for complex, and slower chips.

Optimizing compilers didn't tend to use them. The commands did too much at once. Hard for the compiler to look for redundancies or opportunities to parallelize code.

RISC Machines

RISC designers wanted a high performance processor that had a high clock speed and fit on a single chip.

A CPU on a single chip means less cost, more reliable, and faster clock speed.

Needed to create a new minimal instruction set to make a processor that could fit on a single chip. *Reduced instruction set computers* (RISC) were born.

Why would simpler instructions allow the designer to crank up the clock speed?

RISC Machines Cont.

RISC machines simpler than CISC a myth. Latest RISC most complex ever built.

Complexity moved from instruction set into compiler. A good optimizing compiler can make/break a RISC chip.

Characterizing RISC Machines

Common features found in RISC machines:

- Instruction pipelining
- Pipelining floating point execution
- Uniform instruction length
- Delayed Branching
- Load/store architecture
- Simple addressing modes

Instruction Pipelines

In a computer, every action is synchronized to a clock. Determines speed of system. Technical reasons make it difficult to increase. Cost often prohibitive.

Alternative, is to partially overlap execution of instructions (say: two additions).

Want more than one in progress at same time. Could have two adders, but that means doubling the H/W and thus cost.

Cheaper approach: after launching one instruction, immediately launch second without waiting for first to complete.

Nearly same performance as doubling H/W

Instruction Pipelines Cont.

This approach is called pipelining. Uses fact that many operations can be decomposed into several identifiable steps that use different resources.

Fig 2-1. Say operations can be broken down into 5 independent steps. An operation enters pipeline at left, and after 5 clock ticks exits.

As soon as 1st instruction clears stage 1, second can be started. Can have up to 5 instructions "in flight" at same time.

Powerful! After 5 clock ticks, we get 1 instruction per cycle instead of 1 every 5 ticks!

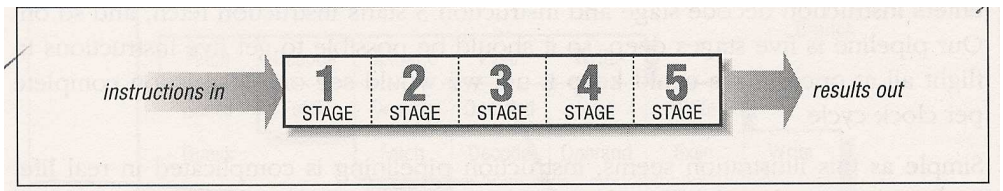

Figure 2-1: A pipeline *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Instruction Processing

Instruction processing can be pipelined. Can be broken down into 5 steps in Fig 2-2.

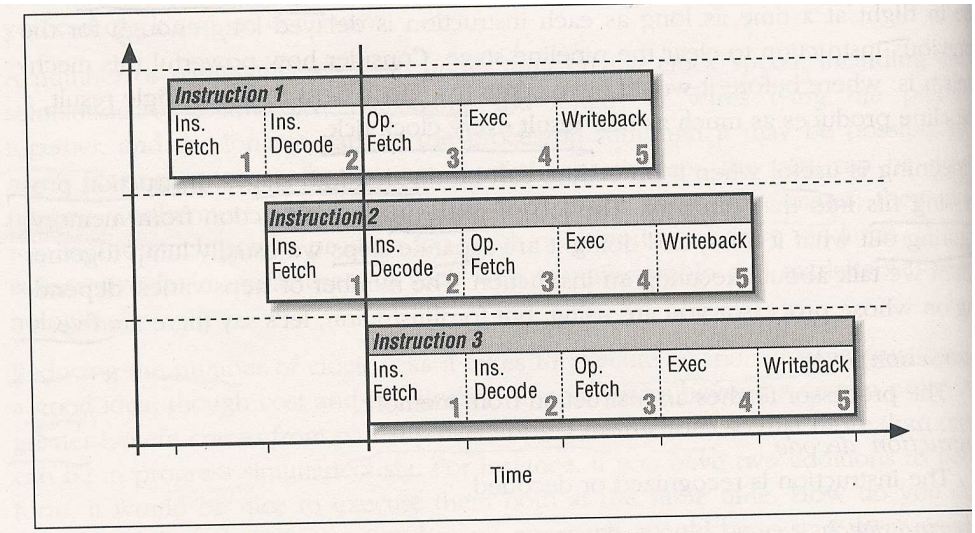

Figure 2-2: Three instructions in flight through one pipeline *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Instruction Processing Cont.

Problems: Each step must occur on different instructions at same time. If stage 2 takes longer than stage 1, then pipeline frozen until stage 2 complete.

A stall at any stage freezes entire pipeline. They must move in lockstep.

What about a jump caused by an if statement?

Processor doesn't know if a branch will take place until executes instruction.

Program Branch Problem

If branch taken, then information in pipeline incorrect and must be flushed. Will be 5 clock cycles before next instruction can be processed, instead of 1! See Fig 2-3.

The longer the pipeline, the greater the penalty! The more stalls, the slower overall performance.

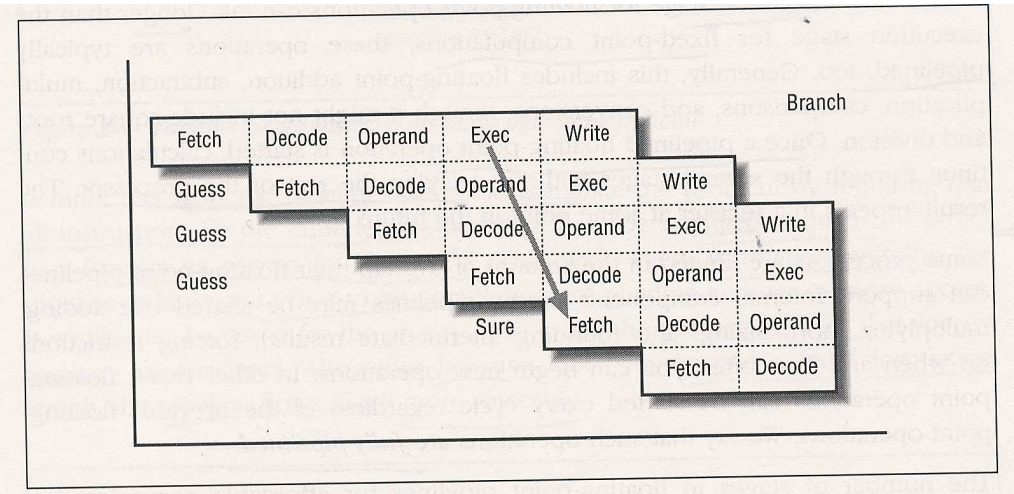

Figure 2-3: Detecting a branch *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Pipelining Floating-Point Operations

As floating-point operations typically take longer than integer, they are usually pipelined.

Usually includes addition subtraction, multiplication, and comparisons.

Once computation started, rest of processor can perform other tasks. Results appear later in a register.

Can have multiple floating-point operations in pipeline at once.

Uniform Instruction Length

For pipeline, want each stage to take about same time. For instruction fetch, how do we know how many bytes to fetch?

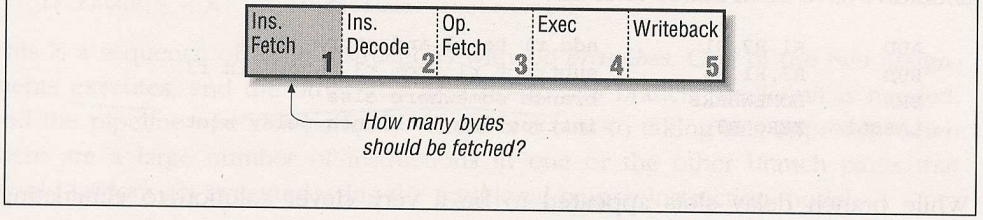

Figure 2-4: Variable length instructions make pipelining difficult *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Uniform Instruction Length Cont.

For CISC machines, instructions can be of variable length. If long instruction, pipeline stalls.

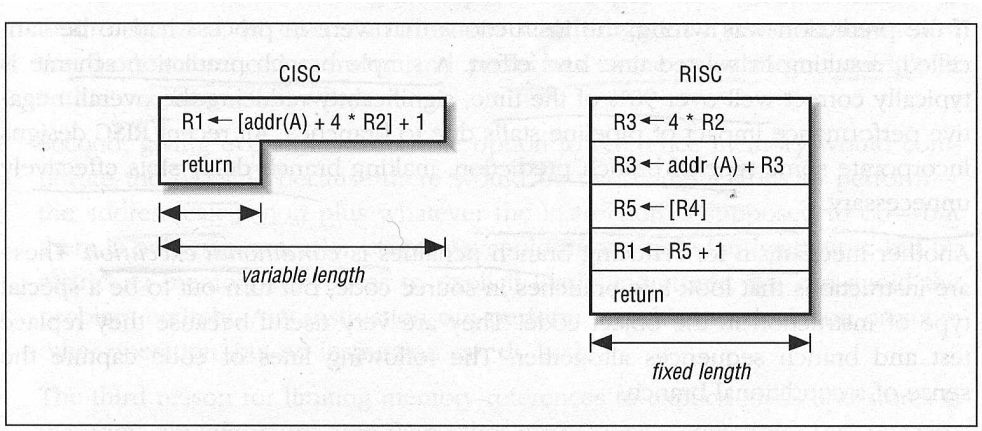

Figure 2-5: Variable-length CISC versus fixed-length RISC instructions *

Eliminate this by making all instructions same length. Thus know complete and doesn't need additional memory fetches.

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Delayed Branches

Used to reduce cost of a misguessed branch.

Required instruction after branch that can be executed no matter what.

More robust approach is to "predict" direction of branch.

During decode stage, CPU notices instruction is a branch. Consults a table that keeps recent behaviour of branch and makes a guess.

Based on guess, processor starts fetching instructions on "predicted" branch.

Load/Store Architecture

Memory access is limited to explicit load and store operations. In a CISC processor, arithmetic operations often include embedded memory references.

- Instructions must be same length. A memory reference and calculation won't fit in one instruction word.

- Complicates pipeline. If calculation could also perform memory access, would need two execution stages.

- Embedded memory references take more time, and would stall pipeline.

Simple Addressing Modes

Want to avoid complex address modes as they require several memory references, and thus take more time. Stalls pipeline.

Can still do complex data structures. Compiler generates explicitly extra address calculations.

Often easier for the compiler to optimize.

Second-Generation RISC

Superscalar Processors: Can now fit more on a chip. Add duplicate elements to increase performance. ie. multiple instruction and floating point pipeline.

Must be independent of each other.

Problem: must find enough to do in parallel (that doesn't violate a precedence) to keep all elements busy.

Also called *multiple instruction issue processors.*

Second-Generation RISC Cont.

Superpipelined Processors: Pipeline depth increased above 5 stages. By decomposing instructions into 10 stages (for example), then *should* be able to double clock speed.

Increases penalty of a stall!

Need good compiler.

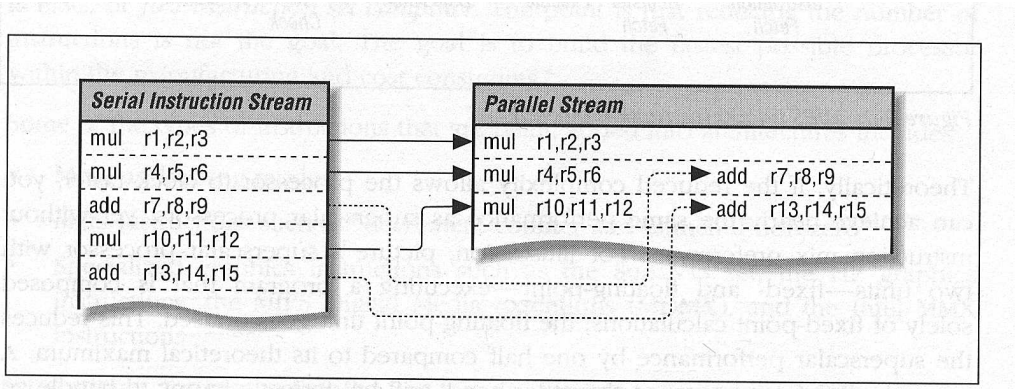

Figure 2-6: Decomposing a serial stream *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Post RISC Architecture

Two-way superscalar processors were achieving about 1.6-1.8 instructions per cycle.

Next step would naturally be four-way or eight-way, right?

Need to keep four (eight) units busy. Hard to find four (eight) sets of instructions in a row to execute in parallel.

Solution? *Out of order execution* and *speculative computation.*

If the processor could not find enough instructions in the sequential stream, it looks ahead for instructions that don't rely on previous ones. It then computes them in advance to take up unused resources.

But, what if that instruction would never get executed?

Speculative Computation

Must separate the idea of *computing* a value from *executing* an instruction.

If need to make use of unused resources, processor computes the value of instructions that it thinks it may need, particularly if they are a slow operation. The results are then stored in an internal, hidden, register.

If a branch occurs, and the instruction isn't needed, the results are discarded.

If required, instruction appears to execute in one clock cycle.

This is called speculative computation.

Instructions that are being executed out of order need to be stored somewhere. This is done in the *instruction reorder buffer* (IRB).

The Post-RISC Pipeline

First two stages of pipeline still fetch and decode. Decode includes branch prediction.

Next, instructions are placed in the IRB to be computed ASAP.

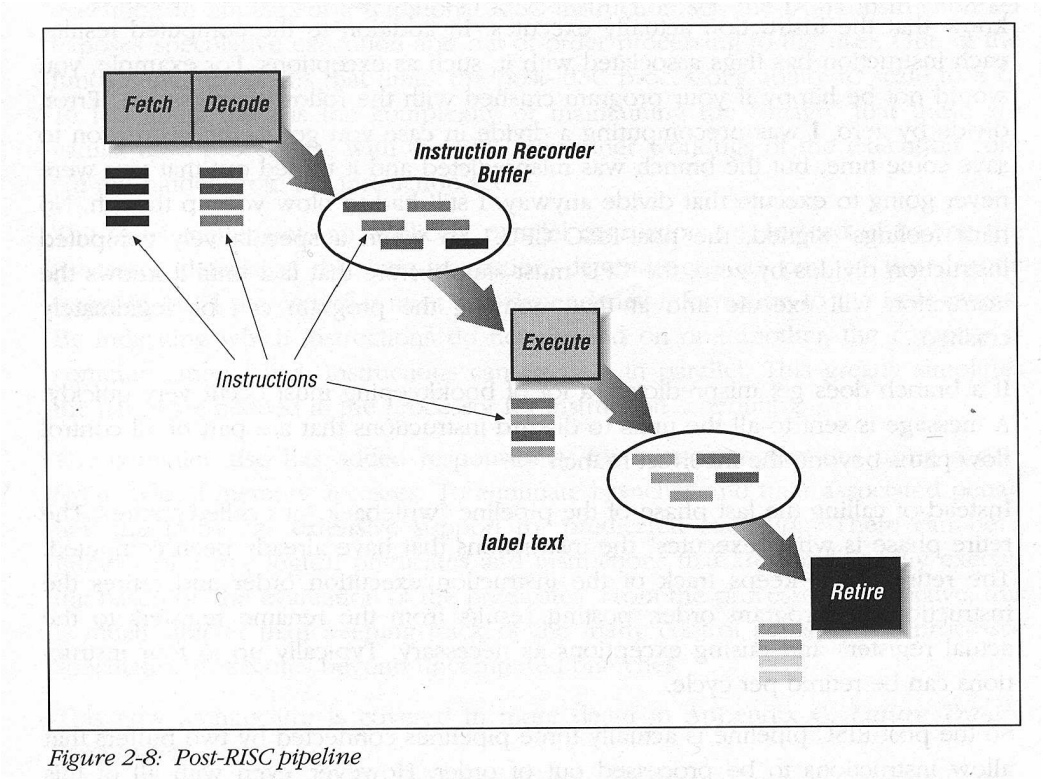

Figure 2-8: Post-RISC pipeline *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

The Post-RISC Pipeline Cont.

The IRB holds instructions waiting to execute.

When decode unit predicts a branch, the instructions on branch are marked so they can be easily found later if prediction incorrect.

In IRB, instructions go to computation units as soon as they have their operands.

As results not seen externally, any instruction ready to go can be computed.

Results stored in registers hidden from programmer called *rename registers*.

Execution unit similar to one from a normal superscalar RISC processor. Pipelined. Generally, up to four instructions can be performed in parallel, if available.

Storing Computations

After results of instructions stored, must wait till the instructions that proceeded it are evaluated to know if instruction actually should be executed.

Not only results stored, but the flags associated with it. Don't want processor to act on an exception before even know if instruction executed!

If branch mispredicted, all instructions from this path must be discarded.

Retiring Instructions

Last phase of pipeline called "retire."

Instructions that have already been computed are now "executed."

Keeps track of instruction order, and "retires" the instructions in the original program order. Normally up to 4 per clock cycle.

Data is copied from rename registers to actual registers, and exceptions are raised as needed.

Post-RISC pipeline actually three pipelines tied together by two buffers.

Externally, appears as a regular RISC processor with expected instruction execution.