CS 61C Lecture Notes Week 14

Topic: Pipeline

Reading: Patterson & Hennessy, Section 6.1

The central processor in any computer has several essentially independent parts. One part decodes an instruction, deciding which operation it's trying to perform. Another part does arithmetic. Another part reads and writes the processor's registers, and another part reads and writes the computer's main memory.

In a traditional computer, which does one instruction at a time, most of that hardware is idle most of the time. While the processor is decoding the instruction, it's not doing arithmetic or reading or writing registers or memory. And the same is true about each step; while the machine is reading the values in the chosen registers, the instruction decoder is idle.

Pipelining is the idea that while carrying out one instruction, we can be decoding the one after it, and reading the one after *that* from memory, all at the same time.

P&H have a good analogy to a laundromat, in which you can be drying one load while washing the next at the same time.

RISC is designed for pipelining

All modern processors use pipelining, but it's very hard to pipeline a traditional complex architecture. A processor like the Intel Pentium, which is a pipelined CISC design, is extremely complicated, and it's easy to have hardware design bugs.

The RISC idea came about originally because memory access speeds became comparable to instruction execution speeds (mainly because of the use of caches), so it was no longer desirable to do as much calculation as possible in each instruction. But even when the balance shifts so that processors are faster than memory access, many of the RISC design principles remain in use because they make pipelining easier.

The most obvious example of a RISC principle that helps pipelining is that all instructions should take about the same amount of time. If it took five times as long to dry a load of laundry as to wash it, using the washer in parallel with the dryer wouldn't help that much; the washed load would still have to wait a long time for the dryer to be free.

In particular, this is why RISC instructions never involve more than one memory reference. Computers used to have instructions with names like Block Transfer that would copy an arbitrary number of consecutive words from one part of memory to another. Such an instruction can't be pipelined. Even without such extreme examples, it used to be common to have arithmetic instructions that would take arguments from memory and/or store the computed value in memory. These instructions could do as many as three memory references. RISC machines use the load/store idea, so that all arithmetic operands must be in memory, and the only memory references are made by specialized load/store instructions that do no other computation.

Pipelining also explains why it's important that the register fields are in the same place in R-format and I-format instructions. This allows the processor to start reading the values from the operand registers at the same time that it's decoding the instruction. (If the instruction turns out to be a jump, for example, these register values won't be used, but that's okay.)

This notion of pipelining is based on the assumption that hardware within the processor is relatively expensive, so that (for example) we want to have only one ALU, but use pipelining to ensure that the ALU is always busy. The very latest computers, using very high density integrated circuit technology, allow for substantial duplication of hardware functions within a processor chip; there might be four or eight ALUs instead of just one. Those machines use a much more complicated approach called *superscalar* processing; the processor might read a dozen instructions at a time and figure out how they can be reordered so that as many as possible can be done in parallel. But for now we'll consider a simple pipeline.

The MIPS pipeline

The MIPS R2000 architecture has a five-stage pipeline:

1. Fetch instruction from memory.
2. Decode the instruction, while simultaneously reading register operands.
3. Perform arithmetic (including the base/offset addition for a load or store instruction).
4. Load or store from/to memory, if applicable.
5. Write the result of arithmetic, or the value fetched from memory, into a register, if applicable.

The arithmetic instructions don't use memory, and therefore don't do anything in step 4. It would be nice if we could have a four-stage pipeline in which step 4 would be combined with either 3 or 5, in the same way that reading the registers is combined with decoding the instruction in step 2. Unfortunately, this wouldn't work. If we combined steps 3 and 4, so that an arithmetic instruction would do the calculation in step 3, but a load/store would access memory in step 3, we get in trouble because a load/store instruction has to compute the memory address (essentially doing an *addi* computation) before it can do the actual transfer from/to memory. If we combined steps 4 and 5, so that the final step would be putting a result either in memory or in a register, then load instructions wouldn't work because they must load from memory in step 4 and write the loaded value into a register in step 5.

(But we do get some benefit from the fact that register access is faster than memory access. Step 5 (write to a register) takes only the first half of each clock cycle; step 2 (read from a register) is done in the second half of each cycle, so the second step of the instruction three cycles later can read the newly-written value during the same clock cycle. An example of this is in P&H Figure 6.37, page 481.)

Potential problems with pipelining

So far we've talked about pipelining as if the instructions in a program were independent of each other, so any stream of instructions could be overlapped smoothly. But in real programs, later instructions depend on results computed by earlier instructions. It's quite common that the result of one instruction is used by the very next instruction in sequence; this won't always work when the instructions are pipelined, because the second instruction tries to use information that the first instruction hasn't computed yet.

P&H list three kinds of *hazards* that can interfere with smooth pipelining.

A *structural* hazard means that two instructions, at two different stages of the pipeline, are trying to use the same piece of hardware at the same time.

In the MIPS architecture, one instruction can be in step 1 (fetching the instruction itself from memory) while another is in step 4 (fetching data from memory). This would be an example of a structural hazard; the MIPS design solves this problem by providing separate pathways into the processor for instructions and data. (This is why there are two separate caches.)

A *data* hazard means that one instruction depends on the result that another instruction hasn't yet computed. For example, if we say

```
add $10, $8, $9
add $11, $10, $4
```

the second `add` instruction needs the value that the first instruction will put into register 10. But when the second instruction is in stage 2 (fetching values from registers), the first instruction is only in stage 3 (doing the arithmetic); the desired value won't really be in `$10` until two cycles later, when the first instruction reaches stage 5.

The solution to this problem is a little complicated and expensive. The machine must notice that the second instruction uses a register set by the first instruction; the machine will then take the value produced by the adder in step 3 of the first instruction and copy that value directly into one of the inputs to the adder in time for step 3 of the next instruction. That is, the second instruction doesn't actually read from `$10` at all.

The name for this solution is *forwarding.*

A *control* hazard means that a decision has to be made based on a value that hasn't been computed yet. Every branch or jump instruction creates a control hazard, because the machine can't know which instruction to fetch next until the branch has decided. We are already fetching the next instruction before we even know that this is a branch or jump! If it's a *conditional* branch, there is the added problem that the branch itself may depend on data computed by the previous instruction.

There are two parts to the MIPS solution. First of all, the most straightforward way to implement a conditional branch would be to make the decision in stage 3, after all the operands are fetched into the arithmetic unit. But instead they include special branch comparison hardware that can read the registers and make the branch decision all in stage 2, without using the regular arithmetic unit in stage 3. This means that a conditional branch is no worse than an unconditional branch or jump.

The second part of the solution is a design decision that seems very strange to people experienced in pre-RISC assembly language programming. In the MIPS architecture, a branch or jump does not really determine where the next instruction comes from. Instead, the instruction following the branch in memory is always done, just as if there were no branch, and the branch controls which instruction to fetch after *that* one. This is called a *delayed branch.* The reason we haven't had to worry about it all this time is that the assembler automatically reorders instructions so that something fills the "delay slot." If we're lucky, the branch doesn't depend on the instruction before it, and the assembler can just interchange those two instructions. If we're unlucky, the assembler may have to insert a "no-operation" instruction after the branch, something like

```
add $0, $0, $0
```

that has no effect. This is just one more way in which an assembly language program isn't a line-by-line equivalent of machine language: The assembler's simulated machine has traditional branching that takes effect immediately, whereas the actual machine has delayed branching. This is the most extreme example of the way RISC design pushes some traditional hardware tasks into software. What makes it tolerable is that a layer of software isolates even the assembly language programmer from this complexity.

(Delayed-effect instructions are becoming less popular as pipelines get deeper and processors have more complicated circuitry that can reorder instructions within the processor. The earliest MIPS design used a delay slot to solve the data hazard of load instructions, which get their data one clock cycle later than register-to-register instructions, but this constraint is no longer used in newer MIPS implementations.)

(By the way, the delayed branch mechanism solves a problem we handwaved away earlier: the race condition between the RFE instruction and the JR that returns to the user program. These instructions actually appear in the opposite order, with the RFE in the delay slot of the JR. This means that the *decision* to jump back to the user happens *before* the RFE, but the actual fetching of the user instruction happens after it. If another exception happens immediately after the RFE, that's okay; the PC that will be stored in the EPC register is the user's PC.)

Despite everything, sometimes a sequence of instructions comes up for which the MIPS can't keep the pipeline moving at full speed. One instruction has to wait for another to advance. This is called a pipeline *stall.* (For example, the multiplier in some MIPS models can take more than one clock cycle. This is why multiplication is divided into two steps, the instruction that computes the 64-bit result and then the instructions that copy half of that result into a general register. With luck, you can do something else between those two instructions, so that there is no stall. But sometimes the `mflo` or `mfhi` must stall until the `mult` is finished.) Every architecture has some stalls, but the goal is to minimize them.

For programmers, the important big idea to take away from all this is that the architecture promises that any sequence of instructions will work correctly, but it doesn't promise optimal efficiency. A naive ordering of instructions may lead to many pipeline stalls. Modern compilers therefore are written taking into account the precise behavior of the pipeline on each target machine.