Computer Science 324
Computer Architecture
Mount Holyoke College
Fall 2007

Topic Notes: Pipelines

We have all seen and experienced examples of pipelining in our daily lives. The book uses a laundry analogy, but any kind of "assembly line" type of operation might be a good example.

The laundry analogy is a good one. Consider how much more quickly laundry can be finished if we take advantage of the fact that the washer and dryer (and, in the book's example, the folder and the storer) can all operate in parallel, and each stage can start doing its work as soon as the previous stage completes its work.

Similar ideas can be used to create a pipeline of instructions being executed by a processor. We have seen that MIPS instructions can be executed in phases and used this to develop the multi-cycle data path and control. These phases can also be used to create such a pipeline.

But before considering the details, we will briefly consider the complexity of ISAs, as this plays a role in how hard it is to pipeline instructions and how effective such a pipeline can be.

RISC vs. CISC

We have used the terms RISC and CISC in passing, but now is a good time to say a bit more about them.

Many ISAs are too complex to be implemented directly in hardware. Instead, microarchitectures are implemented in hardware, and the actual ISA is simulated on this microarchitecture, guided by a microprogram.

Decoding and executing the instruction can require several microcode steps, and the more complex the instruction (more operands, etc) the longer it will take.

Characteristics of *CISC* or *Complex Instruction Set Computer* architectures:

- CISC architectures have large instruction sets: many operations supported as single machine instructions
- Individual CISC instructions may be very complex
  - Motorola 68000 `movem.l` instruction, that moves a specified subset of the 16 user-visible registers onto the call stack in a single instruction

```
swap: link  %a6,#0
      movem.l #0x010f,-(%sp)
      ...
      movem.l (%sp)+,#0xf080
```

```
        unlk    %a6
        rts
```

  - Order for pushing: `a7..a0,d7..d0`, so the above results in `a0` and `d3..d0` being pushed onto the stack – 5 memory writes and a register modification!
  - Order for popping: `d0..d7,a0..a7` – same registers but this time 5 memory reads above

- VAX-11 `SOBGTR` instruction – "Subtract One and Branch on GreaTeR than" – subtract 1 from the specified register and branch to the target address if the result is greater than 0. So an entire loop could be written:

```
        MOVAW   DATA,R6         ; load array ptr into R6
        MOVL    NUM,R9          ; initialize R9 with the number of elts
DOUBLE: ADDW2   (R6),(R6)+      ; double entry, increment ptr
        SOBGTR  R9,DOUBLE       ; loop control
```

- VAX-11 `CMPC5` instruction that compares strings

```
CMPC5   R5,STRING1,#^A/ /,R7,STRING2
```

  This compares the character string whose starting address is specified by `STRING1` and whose length is in register `R5` to the string `STRING2`, length `R7`, using the space character to pad the shorter string for comparison purposes, using registers `R0`, `R1`, `R2` and `R3` to store information about the result of the comparison.

- CISC instructions support a large variety of addressing modes. The 68000 has about a dozen operand modes, e.g.,

  - register direct
  - immediate
  - register indirect (pointers)
    `(%a3)`
  - register indirect with offset (structures)
    `4(%a6)`
  - register indirect with displacement and indexing (arrays)
    `4(%a4,%d1.w)`
  - address register indirect with predecrement
    `-(%a7)`
  - address register indirect with postincrement
    `(%a7)+`

  These can be used with nearly any instruction!

- CISC instructions often allow memory-based data for one or more operands in most instructions, so a single instruction may make several (slow) memory accesses

- CISC instructions can vary in length, where the length often varies according to the addressing mode (extension words on 68000)
  - larger constant/offset values allowed – just in the next memory location
- CISC implementations are often microcoded
- CISC implementations are heavily pipelined
- CISC systems usually offer a relatively small numbers of user-visible registers (a dozen or two)
- CISC results in a relatively compact code footprint (each instruction accomplishes a lot)
- If you have to program directly in assembly language, CISC isn't too bad. Compilers may make use of only a subset of available instructions (for optimizations or portability of code generation)

Researchers (many at Stanford) in the 1980's advocated for a *RISC* or *reduced instruction set computer*. MIPS is an example.

Characteristics of RISC architectures:

- Most RISC instructions can be executed in one "cycle"
  - SPARC supports a single step of multiply, not a full multiply in one instruction
- RISC instruction sets may be large, but are easily decoded
  - small number of opcodes: MIPS has 14 arithmetic, 4 load/store, 4 ($\times 14$) conditionals, 4 misc, 3 multiple cycle *macro* instructions
- RISC instructions support a small number of addressing modes
- RISC memory access is only through explicit operations: `LOAD` and `STORE`
- RISC instructions are fixed-length (no extension words) with simple instruction formats: 3-operand operations, memory reference, conditional branch, jumps
- RISC architectures are simple enough for hardware-based decoding, but there is likely some microcoding done
- RISC instructions are pipelined, but pipes are shorter
  - typical pipeline: instruction fetch, instruction decode, operand decode, execution, data write
- RISC architectures offer larger numbers of registers, frequently supplanting the use of stack (maybe thousands)

  - SPARC rotating register file: globals, ins, locals, and outs, and a register window exposes a rotating subset of the file

- Programs in a RISC assembly are longer, so have a relatively large code footprint
- RISC is harder to program by hand – but easier for a compiler

Intel Pentium processors are CISC but with much of the complexity implemented by an interpreter that sits above a RISC core.

Compare the assembly code generated from a few simple C programs for a variety of architectures:

See Example:
`/home/jteresco/shared/cs324/examples/assembly`

Pipelines

We'll consider a pipeline for MIPS, which is typical of many RISC pipelines.

We have seen the instructions in our MIPS subset include five steps:

1. **IF**: instruction fetch
2. **ID**: register read and instruction decode
3. **EX**: execute or calculate address
4. **MEM**: memory read or write
5. **WB**: write back register

For our example, we will assume that the stages cost 100 time units each, except those that access memory, which cost 200.

Figure 6.3 in the text shows the single-cycle and a simple pipelined execution.

Note that the speed of our pipeline in this case is limited to the speed of slowest component.

The MIPS instruction set was designed with pipelines in mind, so it has features that make pipelining easier:

1. all instructions are the same length, allowing the next **IF** in the pipeline to proceed immediately
2. there are very few instruction formats, allowing both register read and instruction decode to be in the **ID** pipeline stage
3. the limitation on memory access to just the `lw` and `sw` instructions allows **EX** to combine execution and address calculation

4. memory alignment means we always retrieve the entire instruction in a single memory access

A typical RISC system might have a 5-stage pipeline like this.

A CISC system may have a 12-18 stage pipeline.

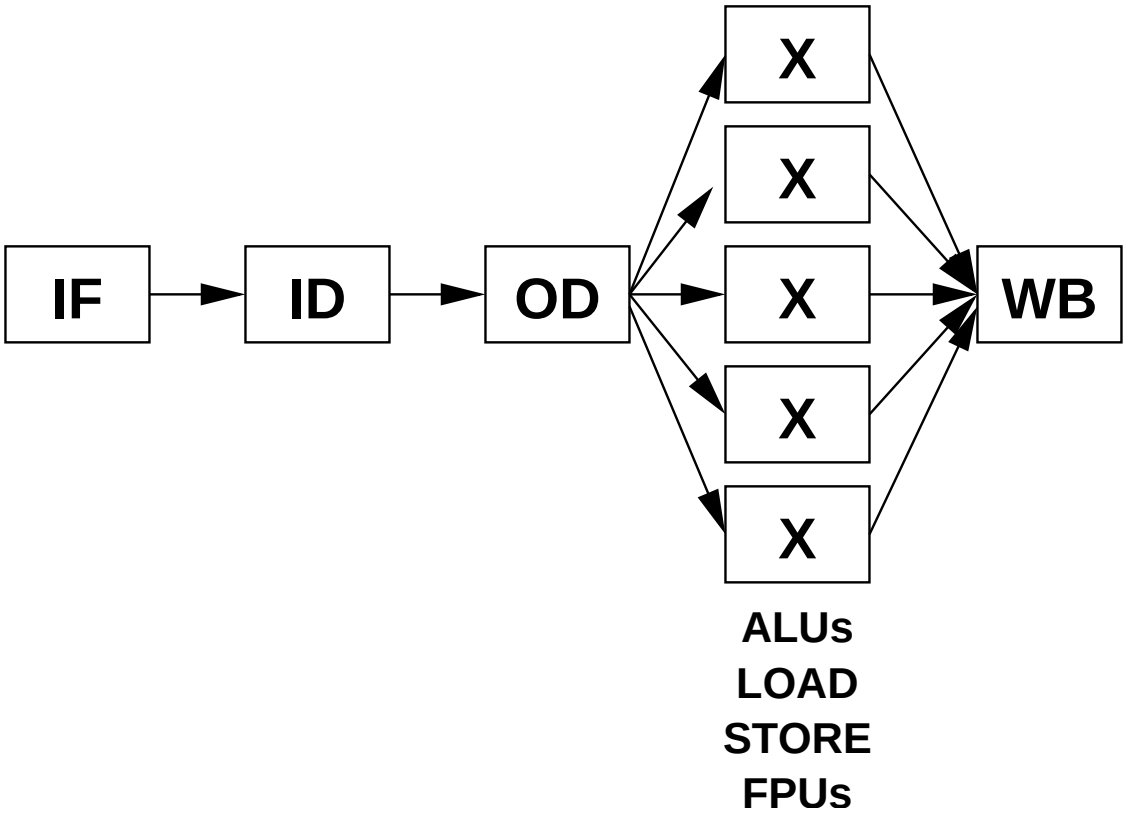
Goal: 100-200 stage pipelines to get very significant speedups.

Many architectures now have multiple pipelines as well.

The original Pentium had two pipelines, and a smart compiler could keep both pipelines busy, effectively doubling the number of instructions completed in the same number of cycles.

If 2 is good, why not 4 or more? Too much duplication of hardware.

Another option: just have multiple functional units, not all stages of the pipeline.



This is especially useful if the execute stage takes longer anyway.

This is a *superscalar* processor.

Hazards

In the ideal situation, we can always be executing an instruction at each stage of the pipeline to achieve maximum throughput. Several things, called *hazards*, can happen that prevent this. They fall into three catagories.

1. *Structural hazards* occur when two instructions executing in a pipeline need the same physical hardware (memory or ALU) at the same time – the implementation and instruction set design can avoid these

2. *Data hazards* occur when an instruction needs to access data that has not yet been produced by another instruction further ahead in the pipeline but which has not yet completed

   For example:

   ```
   add   $s0, $t0, $t1
   sub   $t2, $s0, $t3
   ```

   Here, the value in register `$s0` needs to be read for the `sub` instruction before it has been written by the `add` instruction.

   A simple solution is to have the compiler introduce dummy instructions, or *bubbles*, to *stall* the pipeline.

   This is costly, and there's really no reason to wait to store the result of the `add` into `$s0` and then retrieve it from `$s0` for the `sub` instruction. It has been computed in time, so we can *forward* or *bypass* the value from one internal state register to another, as shown in Figure 6.5 of P&H.

   Figure 6.6 shows that even this is not always sufficient to avoid a bubble – using a value being read from memory in one instruction as an operand in the next is impossible without stalling.

3. *Control hazards* occur when a branch instruction is executed, but subsequent instructions are already in the pipeline behind it – instructions that are not supposed to be executed at all!

   If we notice a branch, we need to go back into our pipeline and cancel any instructions that we've started into the pipeline that are no longer going to happen because there was a branch.

   This will introduce "bubbles" into the pipeline. We can't start doing more useful work in that slot in the pipeline, because we'd already have had to fetch the instruction and we don't know what instruction that will be.

   The bubbles might be inserted by a compiler, in which case we don't have to worry during execution since the items in the pipeline after the branch are not going to do any real work.

   The big danger is that the cancelled instruction could destroy some context before we realize it's not supposed to happen. In this simple pipeline, we're probably safe, since nothing is written back to a register or memory until the last step, but longer pipelines may require more care.

Delayed Branching and Branch Prediction

We can try to minimize the effect of control hazards with two techniques: *delayed branching* and *branch prediction*.

Consider the execution of this code in a pipelined system:

```
beq  $1,$2,32
add  $3,$4,$5
```

The jump/branch is in the pipeline, but by the time we know it's a branch, the add is already in the pipeline.

A compiler can do this on purpose – we know the add is going to happen even though we're taking a branch before we get there.

Most modern architectures have a *delayed branch* of 1 or 2 instruction cycles to allow this optimization.

If we don't have something to do in those *delay slots*, the compiler may have to fill them with `nop`s.

This helps eliminate some bubbles in the pipeline, but when we do have `nop`s, that's still a bubble.

Compilers (or programmers) can also unroll loops to help eliminate branches and keep pipelines full.

This is generally a good thing anyway because branches aren't doing useful work – just wasted time.

But what about conditional branches?

```
bne     loop
add
```

If we take the branch, then the add instruction should never have happened and we have to kill the instruction.

Branch prediction is very useful – try to determine which instruction is most likely to be executed after a branch in an attempt to keep the pipeline going.

Consider

```
if (C)
   S1
else
   S2
```

Which is more likely? Programmers probably make the "then" part the more likely case.

So a compiler might want to set things up to start pipelining S1 after the condition is checked.

How about a while loop or a for loop?

```
while (C)
  S1
```

Here, C will be false only once for the life of the while loop, so the best assumption is to predict a successful branch (another time around the loop).

The UltraSparc III actually has special branch instructions that a compiler can use when it knows a certain branch is likely to be taken the vast majority of the time.

Some rules of thumb:

1. If a branch leads deeper into code, assume the branch will fail.
2. Otherwise, assume the branch will be taken.

This gives about an 80% success rate for human-written code.

Today's branch prediction techniques in optimizing compilers are more intelligent and clever and can get more like 98%.

No matter how good our branch prediction is, it will sometimes fail and we need to be able to make sure instructions can be cancelled.

One possibility: allow instructions to do everything but store their result until we're absolutely sure.

Another headache: multiple conditional branches in the pipeline.

Pipelined Datapath and Control

We will now consider how to construct a data path and control to manage the 5-stage pipeline for our MIPS subset.

In Figure 6.9, we see the single-cycle data path we looked at last month redrawn to show the pipeline phases.

For the most part, information flows left-to-right in this diagram. The exceptions (in blue) represent hazards:

- **WB** puts a result back into the register file – this is a data hazard
- **MEM** may replace the PC with a branch/jump target – a control hazard

Figure 6.10 shows instructions being executed by a pipeline.

- stages are labeled by the components being used in each
- note that the register file is written in the first half of a cycle, and read in the second half; this reasonable assumption helps us avoid some potential hazards later on

Just like when we moved from the single-cycle implementation to the multi-cycle implementation, we will need to add registers to our data path to support pipelining. These registers are shown in Figure 6.11.

- each set of registers holds the values passed between each pair of adjacent stages
- each is large enough to hold the necessary values

The text presents a series of figures showing the active parts of the pipeline during the execution:

- Figure 6.12 shows the first two stages, which are identical for all instructions
- Figures 6.13 and 6.14 show the completion of a `lw` instruction
- Figures 6.15 and 6.16 show the completion of a `sw` instruction – note that nothing happens in stage 5 as this instruction takes only 4 steps
- Figure 6.17 adds extra values to the pipeline registers in recognition of the fact that the register number needs to be retained for the **WB** stage

Augmenting control to support a pipelined control may seem daunting, but it really is not:

- we can use the same control lines as we did for the single-cycle implementation, but each stage should be using the control as set for the instruction it is executing
- the control values can be stored in the pipeline registers to make this happen
- Figure 6.27 shows the pipelined data path with the control added

---

Dealing with Hazards

We noticed earlier that our pipelines cannot always operate at full capacity.

- some instructions do not need to use all stages of the pipeline
- instructions may depend on values computed in prior instructions that are still in the pipeline (data hazards)
- instructions may begin executing before a previous jump or branch is taken (control hazards)

The text discusses ways that the data path can be augmented to detect and deal with hazards. We will not look in as much detail as the text, but we will look at a few figures that demonstrate this:

- Figure 6.28 shows the dependencies in an instruction sequence – `$2` is computed by the first instruction and is used by the next 4
- Figure 6.29 shows that the needed value does exist in pipeline registers in time
- Figure 6.32 shows the data path augmented with additional lines and a forwarding unit that can resolve data hazards

- Figure 6.34 shows a data hazard that cannot be resolved through forwarding – in this case, a stall or bubble must be inserted into the pipeline as shown in Figure 6.35
- The data path augmented with a hazard detection unit is shown in Figure 6.36
- Figure 6.37 shows a branch instruction that results in a control hazard – instructions already in the pipeline that are not to be executed are flushed

We already discussed some branch prediction techniques. The text discusses others.