Pipeline Hazards

Let's put some instructions through the pipeline, and see what happens...

```
addi  $1, $2, 17    _ _ _ _ _
add   $3, $4, $5      _ _ _ _ _
lw    $6, 100($7)       _ _ _ _ _
sw    $8, 200($9)         _ _ _ _ _
```

That all looks fine... What sorts of things can go wrong? Let's just put a whole bunch of `add` instructions in a row, and think about it.

```
add   $1,  $2,  $3    _ _ _ _ _
add   $4,  $5,  $6      _ _ _ _ _
add   $7,  $8,  $9        _ _ _ _ _
add   $10, $11, $12         _ _ _ _ _
add   $13, $14, $1            _ _ _ _ _        (data arrives early; OK)
add   $15, $16, $7              _ _ _ _ _      (data arrives on time; OK)
add   $17, $18, $13               _ _ _ _ _    (uh, oh)
add   $19, $20, $17                 _ _ _ _ _  (uh, oh again)
```

These are examples of data hazards.

Defining hazards

The next issue in pipelining is *hazards*. A pipeline hazard refers to a situation in which a correct program ceases to work correctly due to implementing the processor with a pipeline. There are three fundamental types of hazard: data hazards, branch hazards, and structural hazards. Data hazards can be further divided into Write After Read, Write After Write, and Read After Write hazards.

Structural Hazards

These occur when a single piece of hardware is used in more than one stage of the pipeline, so it's possible for two instructions to need it at the same time.

So, for instance, suppose we'd only used a single memory unit instead of separate instruction memory and data memories. A simple (non-pipelined) implementation would work equally well with either approach, but in a pipelined implementation we'd run into trouble any time we wanted to fetch an instruction at the same time a `lw` or `sw` was reading or writing its data.

In effect, the pipeline design we're starting from has anticipated and resolved this hazard by adding extra hardware.

Interestingly, the earlier editions of our text used a simple implementation with only a single memory, and separated it into an instruction memory and a data memory when they introduced pipelining. This edition starts right off with the two memories.

Also, the first Sparc implementations (remember, Sparc is almost exactly the RISC machine defined by one of the authors) *did* have exactly this hazard, with the result that load instructions took an extra cycle and store instructions took two extra cycles.

Data Hazards

This is when reads and writes of data occur in a different order in the pipeline than in the program code. There are three different types of data hazard (named according to the order of operations that must be maintained):

RAW

A Read After Write hazard occurs when, in the code as written, one instruction reads a location after an earlier instruction writes new data to it, but in the pipeline the write occurs after the read (so the instruction doing the read gets stale data).

WAR

A Write After Read hazard is the reverse of a RAW: in the code a write occurs after a read, but the pipeline causes write to happen first.

WAW

A Write After Write hazard is a situation in which two writes occur out of order. We normally only consider it a WAW hazard when there is no read in between; if there is, then we have a RAW and/or WAR hazard to resolve, and by the time we've gotten that straightened out the WAW has likely taken care of itself.

(the text defines data hazards, but doesn't mention the further subdivision into RAW, WAR, and WAW. Their graduate level text mentions those)

Control Hazards

This is when a decision needs to be made, but the information needed to make the decision is not available yet. A Control Hazard is actually the same thing as a RAW data hazard (see above), but is considered separately because different techniques can be employed to resolve it - in effect, we'll make it less important by trying to make good guesses as to what the decision is going to be.

Two notes: First, there is no such thing as a RAR hazard, since it doesn't matter if reads occur out of order. Second, in the MIPS pipeline, the only hazards possible are branch hazards and RAW data hazards.

Resolving Hazards

There are four possible techniques for resolving a hazard. In order of preference, they are:

1. Forward. If the data is available somewhere, but is just not where we want it, we can create extra data paths to ``forward'' the data to where it is needed. This is the best solution, since it doesn't slow the machine down and doesn't change the semantics of the instruction set. All of the hazards in the example above can be handled by forwarding.
2. Add hardware. This is most appropriate to structural hazards; if a piece of hardware has to be used twice in an instruction, see if there is a way to duplicate the hardware. This is exactly what the example MIPS pipeline does with the two memory units (if there were

only one, as was the case with RISC and early SPARC, instruction fetches would have to stall waiting for memory reads and writes), and the use of an ALU and two dedicated adders.

3. Stall. We can simply make the later instruction wait until the hazard resolves itself. This is undesireable because it slows down the machine, but may be necessary. Handling a hazard on waiting for data from memory by stalling would be an example here. Notice that the hazard is guaranteed to resolve itself eventually, since it wouldn't have existed if the machine hadn't been pipelined. By the time the entire downstream pipe is empty the effect is the same as if the machine hadn't been pipelined, so the hazard has to be gone by then.
4. Document (AKA punt). Define instruction sequences that are forbidden, or change the semantics of instructions, to account for the hazard. Examples are delayed loads and delayed branches. This is the worst solution, both because it results in obscure conditions on permissible instruction sequences, and (more importantly) because it ties the instruction set to a particular pipeline implementation. A later implementation is likely to have to use forwarding or stalls anyway, while emulating the hazards that existed in the earlier implementation. Both Sparc and MIPS have been bitten by this; one of the nice things about the late, lamented Alpha was the effort they put into creating an exceptionally "clean" sequential semantics for the instruction set, to avoid backward compatibility issues tying them to old implementations.

Hazards and Pipeline Diagrams

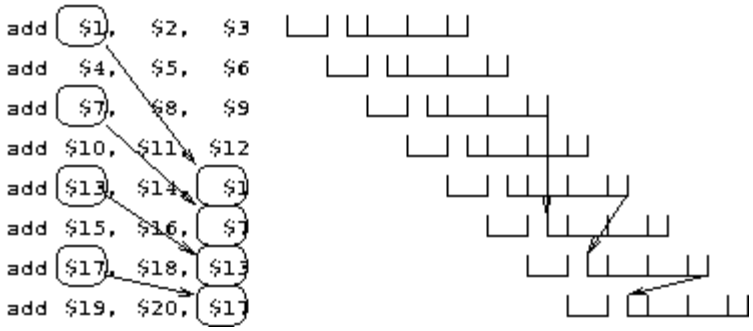
The definitions I put up above are all very nice, but they're a bit abstract. We can actually get a pretty good picture of what's going on regarding hazards by inspecting the pipeline diagram. First, remember the interpretation of the diagram:

1. Every row of the diagram is an instruction
2. Every column of the diagram is a time unit
3. Every "tick" in the diagram is a piece of hardware

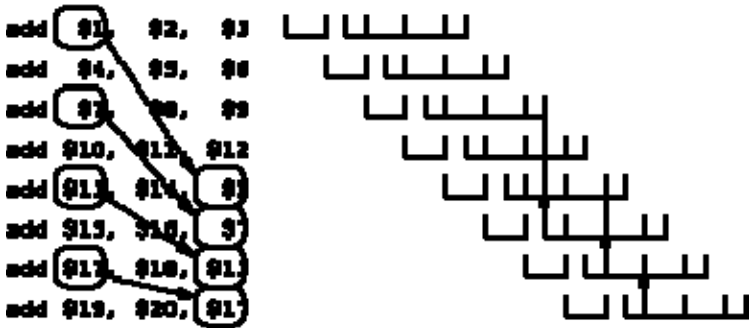
This gives us some rules for the diagram:

- Data can only move horizontally (down the pipeline) or vertically (forwarding). It can't go on a diagonal: if you're trying to move it forward in time (to the right), you have to move it horizontally down the pipeline until the right time to forward it, then it has to move vertically to the unit that needs it. It simply can't go back in time at all (if it could, I could watch tomorrow's horse races and retire in very short order!).
- One piece of hardware can only be used by one instruction at a time.

We can watch these rules in action in the example from last time. Here's what the diagram would look like with no forwarding (we're showing a little bit more detail than usual; in particular, we're showing the fact that the register file is only being used to write for half a cycle and to read for half a cycle. Most of the time we'll just show it as if there were a structural hazard, like we did up above):



The fact that there is data moving backwards in time is a problem! We resolve it by forwarding:



A Hazard that Needs a Stall

Now, let's look at what happens with another example:

This time, we can't resolve the hazard with forwarding: the data isn't available until the end of the memory stage of the first instruction, so the execute state of the second instruction has to wait a cycle for it:

After the second instruction has stalled one cycle, we can forward the data to it. This has fixed the first problem, but introduced a new one: the second and third instructions are both using the execute, memory, and writeback stages at the same time - a structural hazard. So we need to stall the third instruction to resolve this: