SIENAcollege
Computer Science

Computer Science 220
Assembly Language & Comp. Architecture
Siena College
Fall 2011

Topic Notes: Pipelines

We have all seen and experienced examples of pipelining in our daily lives. When a task can be broken into subtasks that can be done in sequence and subtasks of different tasks can be done in parallel, pipelining is possible.

The book uses a laundry analogy (Figure 4.25), 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. We don't process a single load of laundry any more quickly (in fact, we'll see that pipelining may *increase* the time for a single load), but the overlap in successive loads leads to a faster overall completion time.

Similar ideas can be used to create a pipeline of instructions being executed by a processor. The execution of machine instructions can be usually be broken down into phases. These phases can be used to create such a pipeline.

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

From our datapath and control experience, we can see that the instructions in our MIPS subset can be viewed as having 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 running example, we will assume that the register access stages cost 100 time units each, while memory access and ALU operations cost 200.

Figure 4.26 in the text shows how long the different instructions in our MIPS subset will take given these latencies. This means that if we were to use a regular single-cycle implementation, the cycle time would need to be 800 (the time for the `lw`) instruction, since all instructions must be given the same amount of time to execute.

Figure 4.27 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. A single instruction now takes as much as 900ps to complete – it's slower! But...the parallelism allows us to complete an instruction every 200 units. So effectively, we are speeding up execution by a factor of 4 ($\frac{800}{200}$) *if* we can keep the pipeline going.

Note also that the register accesses are strategically set up so that a register read takes place in the second half of a 200ps slot and the register write takes place in the first half. This will be beneficial later on.

Figure 4.28 shows a symbolic representation of the execution of an `add` instruction showing which components are used in each step.

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 (the control circuitry) 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 (for R-formats or conditional branch comparisons) and address calculation (for memory access) – no one instruction needs to do both
4. memory alignment means we always retrieve the entire instruction in a single memory access

A 5-stage pipeline like this is typical of a RISC system.

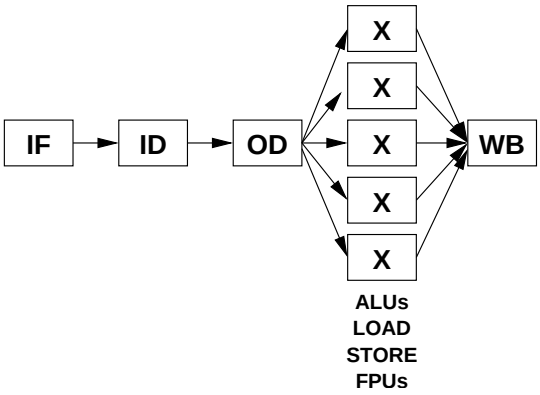
A system with a more complex instruction set may have a 12-18 stage pipeline, as a single instruction performs more work.

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? There is a limit as we'd have too much duplication of hardware and not enough instructions to make use of it.

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 an ideal situation, we can always be executing an instruction at each stage of the pipeline to achieve maximum throughput. However, we need to make sure that the end result of a pipelined execution is identical to a purely sequential execution. Several situations lead to *hazards*, where certain instructions cannot execute one immediately following another in a pipeline while still producing the correct result. In these cases, we may need to delay the execution of some instructions to guarantee correct results, at the expense of lowering throughput.

Hazards fall into three categories.

1. *Structural hazards* occur when two instructions executing in a pipeline need the same physical hardware (memory or ALU) at the same time. The pipeline 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 its execution.

   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, just not

yet written back to the register file! Instead, we can *forward* or *bypass* the value from one internal state register to another, as shown in Figure 4.29.

Figure 4.30 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:

```
j somewhere
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.

Consider a simple for loop:

```
  int sum = 0;
  for (int i = 1; i <= 10; i++) sum += i;
```

which might be translated into MIPS as:

```
        add $s1, $0, $0      # sum = 0
        addi $t1, $0, 1      # i = 1
loop:   slti $t2, $t1, 11    # compare i to 11
        beq over             # skip loop if i is not < 11
        add $s1, $s1, $t1    # sum += i
        addi $t1, $t1, 1     # i++
        j loop               # back around
over:
```

If we have a branch delay slot, we can swap the `addi` that performs `i++` and the `j` that goes back to the top of the loop. We always want to do the `addi`, even though it would then come *after* the `j`.

Compilers (or programmers) can also unroll loops to help eliminate branches to help keep pipelines full. The code above might be rewritten as:

```
        add $s1, $0, $0      # sum = 0
        addi $s1, $s1, 1     # sum += 1
        addi $s1, $s1, 2     # sum += 2
        addi $s1, $s1, 3     # sum += 3
        addi $s1, $s1, 4     # sum += 4
        addi $s1, $s1, 5     # sum += 5
        addi $s1, $s1, 6     # sum += 6
        addi $s1, $s1, 7     # sum += 7
        addi $s1, $s1, 8     # sum += 8
        addi $s1, $s1, 9     # sum += 9
        addi $s1, $s1, 10    # sum += 10
```

This is generally a good thing anyway because branches aren't doing useful work – they're just just wasted time. In this case, we reduce from 54 instructions in the original code to only 11. But we could only do this by noticing that the loop was going to run exactly 10 times and replicating the appropriate code.

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: we try to determine which instruction is most likely to be executed after a branch in an attempt to keep the pipeline going. We cannot be perfect – after all the whole point of a conditional branch is to make a decision. But the more we can guess correctly, the more likely we are to keep the pipeline full.

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 organize code 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 they are supposed to occur.

In architectures with longer pipelines, we may also need to deal with multiple conditional branches in the pipeline at once!

Pipelined Datapath and Control

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

In Figure 4.33, we see the single-cycle data path we looked at before, 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 4.34 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
- in this case, no hazards arise

We will need to add registers to our data path to support pipelining. These registers are shown in Figure 4.35.

- 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:

- The top half of Figure 4.36 shows the **IF** stage:
  - the instruction from memory is retrieved and stored in the **IF/ID** pipeline registers
  - PC+4 is computed and stored in the **IF/ID** pipeline registers
  - The PC is updated with either PC+4 or the result of a branch instruction
- The bottom half of Figure 4.36 shows the **ID** stage:
  - the instruction stored in the **IF/ID** pipeline registers is used to retrieve 2 values from the register file, which are both sent to the **ID/EX** pipeline registers

  - the immediate field of the instruction is sign-extended to 32 bits and stored in the **ID/EX** pipeline registers
  - the PC+4 value is passed along from the **IF/ID** pipeline registers to the **ID/EX** pipeline registers for use later
  - we don't need all of these values, but we don't necessarily know which, yet, so we pass them all along

- Figure 4.37 shows the **EX** stage for a `lw` instruction
  - the sign-extended immediate value is added to the base register, both of which come from the **ID/EX** pipeline registers
  - this sum (the effective address for the memory access) is stored in the **EX/MEM** pipeline registers

- Figure 4.38 (top) shows the **MEM** stage for a `lw` instruction
  - the effective address stored in the **EX/MEM** pipeline registers is used to retrieve a value from the data memory
  - this value is stored in the **MEM/WB** pipeline registers

- Figure 4.38 (bottom) shows the **WB** stage for a `lw` instruction
  - the value retrieved from memory, saved in the **MEM/WB** pipeline registers, is sent back to the register file for storing

- Figure 4.41 adds extra values to the pipeline registers in recognition of the fact that the register number needs to be retained for the **WB** stage (if we don't, we'd be using the destination register from a different instruction!)

- Figure 4.42 highlights the parts of the datapath that are used for `lw`

- Figures 4.39 and 4.40 show the completion of a `sw` instruction
  - here, we need to remember the value from the register file that is to be stored in memory. It must be passed along during the **EX** phase from the **ID/EX** registers to the **EX/MEM** registers
  - during **MEM**, we store the value at the location specified by the effective address, both coming from the **EX/MEM** pipeline registers
  - the **WB** stage does nothing for `sw`

- Figure 4.43 shows a sequence of instructions in a pipeline, and Figure 4.45 shows the instructions in execution at the fifth step of this execution sequence

Augmenting control to support a pipelined control may seem daunting, but it really is not as bad as we'd expect.

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.

Control values can be stored in the pipeline registers to make this happen.

Figure 4.46 shows the pipelined data path with the control added.

We need only store control signals at each stage that are to be used in that or in subsequent stages (Figure 4.50).

Figure 4.51 shows the complete pipelined datapath.

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)

Data Hazards

We first look at enhancements to our pipelined datapath and control that will deal with data hazards.

Figure 4.52 shows the dependencies in an unfortunate instruction sequence

- `$2` is computed by the first instruction and is used by the next 4.
- The second and third instructions need the value of `$2` before the first instruction has had a chance to store it.
- The last two instructions are fine, remembering that values are written to the register file in the first half of the clock cycle, read in the second half.

Upon closer inspection, though, we see that the values needed by the second and third instructions are available in pipeline registers, as shown in Figure 4.53.

- the value for `$2` in the `and` instruction is in the **EX/MEM** pipeline registers
- the value for `$2` in the `or` instruction is in the **MEM/WB** pipeline registers

We need to detect when these conditions occur and account for them. There are two main cases to deal with:

1. the destination register (`Rd`) of the instruction in the **MEM** phase is one of the source registers (`Rs` or `Rt`) of the instruction in the **EX** phase
2. the destination register (`Rd`) of the instruction in the **WB** phase is one of the source registers (`Rs` or `Rt`) of the instruction in the **EX** phase

For the first case, we need to divert a value from the **EX/MEM** pipeline registers as an input to the ALU. In the second, we take a value from the **MEM/WB** pipeline registers.

The text breaks this info four cases we can check for:

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

In our example from Figure 4.52, the "sub-and" hazard is of type 1a and the "sub-or" hazard is of type 2b.

So we want to detect these conditions and perform appropriate forwarding only when it matters – that is, when the first instruction is actually going to write the ALU result into a register (i.e. its `RegWrite` control line is 1).

We can also skip forwarding if the destination register is `$0` since it will never be changed to have any value other than 0.

Figure 4.54 shows the addition of a forwarding unit that can present inputs to the ALU from pipeline registers instead of from the register file. The forwarding unit handles **EX** or **MEM** hazards by setting two new control lines, `ForwardA` and `ForwardB`.

The `ForwardA` and `ForwardB` lines can be set according to these rules:

- for an **EX** hazard:

```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 10
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 10
```

- for a **MEM** hazard:

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01
```

However, the rules for **MEM** do not take into account that the forwarding from **MEM** should not take place if there is an **EX** hazard for the same destination register (a *double data hazard*), since that value would overwrite (in a non-pipelined world) the value in the register to be used by the **EX** instruction. The rules for **MEM** are then augmented to:

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
                and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
                and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01
```

Figures 4.56 and 4.57 show the data path augmented with additional lines and a forwarding unit that can resolve data hazards.

We next consider an even more unfortunate data hazard, a “load-use” hazard as shown in Figure 4.58.

This one cannot be resolved through forwarding – the value has not yet been retrieved from the data memory by the time it is needed.

In his case, we need to *stall* the pipeline to wait for the value to become available, as shown in Figure 4.59.

This is accomplished by adding a *hazard detection unit*.

Using our notation from before, we know a stall is necessary when:

```
ID/EX.MemRead and
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
   (ID/EX.RegisterRt = IF/ID.RegisterRt))
```

Note that we are detecting this condition as early as possible – when the offending sequence instructions are in the **IF** and **ID** stages. This makes it easier to stall.

The stall is accomplished by adding a *bubble* to the pipeline – an instruction that does nothing, a *nop*, or "no op".

This requires that:

- the PC is not updated
- the **IF/ID** pipeline registers are not updated
- the control entries in the **ID/EX** pipeline register are loaded with all 0's, which results in the bubble/nop.

The data path augmented with a hazard detection unit is shown in Figure 4.60.

Stalls reduce the performance – we're spending time executing a nop instead of meaningful instructions. However, they are essential to ensure correct behavior.

An optimizing compiler should be aware of the details of the pipeline and can rearrange instructions (in many cases) to avoid the need for a stall at execution time.

Control Hazards

Recall that a control hazard occurs when a branch/jump instruction is being executed and subsequent (partially-executed) instructions in the pipeline need to be cancelled since they never should have been executed.

The specifics of our pipeline mean that a conditional branch's outcome is not known until the **MEM** phase.

Figure 4.61 shows a branch instruction that results in a control hazard – instructions already in the pipeline that are not to be executed are flushed (their control lines are set to 0).

Figure 4.62 shows how we can reduce the cost of a control hazard by adding hardware to determine the result of a conditional branch sooner (during **ID**):

- The branch target adder is moved to **ID**.
- An equality checker is added that compares the values coming out of the register file that can quickly determine the result of a `beq` (or `bne`).

Another technique that works well with short pipelines (such as our 5-stage pipeline) is the use of a *branch delay slot*.

- Here, we *always* execute the instruction immediately following a branch or jump.

- This way, a programmer or (hopefully) a compiler can reorder instructions so that some useful work can be done in the slot after the branch.
- This is not always possible, so a `nop` instruction may need to be inserted in some cases.
- However, it is a quite effective way to reduce the cost and frequency of control hazards. Real implementations of MIPS (among other ISAs) make use of branch delay slots.

Figure 4.64 shows some examples of how code can be reordered to take advantage of branch delay slots.

We already discussed some branch prediction techniques. The text discusses some of them in a bit more detail.