Chapter 4

The Processor

Data Hazards

Pipeline Complexity

- Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power. The Pentium 4 pipeline stages do not include the commit stages. If they were included, the Pentium 4 pipelines would be even deeper.

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue Width</th>
<th>Out-of-Order/Speculation</th>
<th>Cores/Chip</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>Ns</td>
<td></td>
<td>1</td>
<td>5 W</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>Ns</td>
<td></td>
<td>1</td>
<td>10 W</td>
</tr>
<tr>
<td>Intel Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>Yes</td>
<td></td>
<td>1</td>
<td>29 W</td>
</tr>
<tr>
<td>Intel Pentium 4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>Yes</td>
<td></td>
<td>1</td>
<td>75 W</td>
</tr>
<tr>
<td>Intel Pentium 4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>Yes</td>
<td></td>
<td>1</td>
<td>103 W</td>
</tr>
<tr>
<td>Intel Core</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>Yes</td>
<td></td>
<td>2</td>
<td>75 W</td>
</tr>
<tr>
<td>Intel Core i5 Nehalem</td>
<td>2010</td>
<td>3300 MHz</td>
<td>14</td>
<td>Yes</td>
<td></td>
<td>1</td>
<td>87 W</td>
</tr>
<tr>
<td>Intel Core i5 Ivy Bridge</td>
<td>2012</td>
<td>3400 MHz</td>
<td>14</td>
<td>Yes</td>
<td></td>
<td>8</td>
<td>77 W</td>
</tr>
</tbody>
</table>
A8 Pipeline

- The first three stages fetch instructions into a 12-entry instruction fetch buffer. The Address Generation Unit (AGU) uses a Branch Target Buffer (BTB), Global History Buffer (GHB), and a Return Stack (RS) to predict branches to try to keep the fetch queue full. Instruction decode is five stages and instruction execution is six stages.

Core i7 Pipeline

- The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mis-predictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.
Review - Pipeline Registers

- Need registers between stages to hold information produced in previous cycle.

Review - Five Instruction Sequence

Once the pipeline is full, one instruction is completed every cycle, so CPI = 1
Review: Can Pipelining Get Us Into Trouble?

- Yes: **Pipeline Hazards**
  - **Structural hazards**: different instructions in different stages (or the same stage) conflicting for the same resource.
  - **Data hazards**: an instruction cannot continue because it needs a value that has not yet been generated by an earlier instruction.
  - **Control hazards**: fetching the next instruction cannot continue because it does not know the outcome of a branch – special case of a data hazard – separate category because they are treated in different ways.
- Can usually resolve hazards by **waiting**.

Pipelined Control - Simplified
Pipelined Control

- Control signals are derived from the instruction opcode as in single-cycle implementation.
Structural Hazards

- A structural hazard is a conflict for use of a resource.
- A combination instruction/data memory would create a structural hazard in a pipelined architecture
  - Load/store requires data access.
  - Instruction fetch would have to stall for that cycle.
- Fix with separate instruction and data memories (I$ and D$).

A Single Memory Would Be a Structural Hazard

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
<th>lw</th>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
<th>Inst 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. Order</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>Reading data from memory</td>
<td>Reading instruction from memory</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Register File Access

Eliminate register file access hazard by doing reads in the second half of the cycle and writes in the first half.

Data Hazards

- An instruction *produces* a value in a given pipeline stage.
- A subsequent instruction *consumes* that value in a pipeline stage.
- The consumer may have to be delayed so that the time of consumption is later than the time of production.
Data Hazards

- Data hazards occur when an instruction depends on data computed by a previous instruction:
  - `add $s0, $t0, $t1`
  - `sub $t2, $s0, $t3`

One Way to “Fix” a Data Hazard

Can fix data hazards by waiting – stall – but impacts CPI.
A Better Way - Forwarding

- Use result when it is computed
  - Don't wait for it to be stored in a register.
  - Requires extra connections in the datapath.

**Forwarding Example**

- **add** $1,$
- **sub** $4,$1,$5
- **and** $6,$1,$7
- **or** $8,$1,$9
- **xor** $4,$1,$5
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - The value may not be available when needed.
  - Can’t forward backward in time.

Example of Load-use Data Hazard

- Dependencies backward in time cause **hazards**.

```
lw $s0, 20($t1)
sub $s2, $s0, $t3
```
**Code Scheduling to Avoid Stalls**

- Reorder code to avoid the use of the load result in the next instruction.
- C code for \( A = B + E; \ C = B + F; \)

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

13 cycles

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

11 cycles

**Data Hazards in ALU Instructions**

- Consider this sequence:
  ```
  sub $2, $1, $3
  and $12, $2, $5
  or $13, $6, $2
  add $14, $2, $2
  sw $15, 100($2)
  ```
- Can we resolve hazards with forwarding?
- How do we detect when to forward?
## Dependencies & Forwarding

<table>
<thead>
<tr>
<th>Value of register $$2$:</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Program execution order (in instructions):

- sub $\$2$, $\$1$, $\$3$
- and $\$12$, $\$2$, $\$5$
- or $\$13$, $\$6$, $\$2$
- add $\$14$, $\$2$, $\$2$
- sw $\$15, 100(\$2)$

### Data Forwarding

- Take the result from the earliest point that it exists in **any** of the pipeline state registers and forward it to the functional units that need it in that particular cycle.
- For the ALU functional unit: the inputs can come from **any** pipeline register rather than just from ID/EX by
  - Adding multiplexers to the inputs of the ALU.
  - Connecting the Rd write-data in EX/MEM or MEM/WB to either (or both) of the EX’s stage Rs and Rt mux inputs.
  - Adding the proper control hardware to control new muxes.
- Other functional units may need similar forwarding logic (e.g., the DM).
- With forwarding, you can achieve a CPI of 1 even in the presence of data dependencies.
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
  - ALU operand register numbers in EX stage are given by
    - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  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

Detecting the Need to Forward

- But only if forwarding instruction will write to a register
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0
### Forwarding Paths

#### 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

#### 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
Double Data Hazard

- Consider the sequence:
  - add $1, $1, $2
  - add $1, $1, $3
  - add $1, $1, $4

  Both hazard situations occur
  - Want to use the most recent.
  - Revise MEM hazard condition
    - Only fwd if EX hazard condition isn’t true.

Revised Forwarding Conditions

- MEM hazard
  - 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
Datapath with Forwarding

Load-Use Data Hazard

Need to stall for one cycle
Load-Use Hazard Detection

- Check when instruction is decoded in ID stage.
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble.

How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation).
- Prevent update of PC and IF/ID register
  - Simply forces the same instruction to be decoded again.
  - Following instruction is fetched again.
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage if needed.
Stall/Bubble in the Pipeline

Datapath with Hazard Detection
Summary

- All modern day processors use pipelining for performance (a CPI of 1 and a fast clock cycle).
- Pipeline clock rate limited by **slowest** pipeline stage – so designing a balanced pipeline is important.
- Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly.
  - Data hazards
    - Stall (impacts CPI).
    - Forward (requires hardware support).
  - Control hazards – put the branch decision hardware in as early a stage of the pipeline as possible
    - Stall (impacts CPI).
    - Delay decision (requires compiler support).
    - **Static** and **dynamic prediction** (requires hardware support).