CPU Performance Equation

- Execution Time = \text{seconds/program}

\[
\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}
\]

- Program
- Architecture (ISA)
- Compiler
- Compiler (Scheduling)
- Organization (uArch)
- Microarchitects
- Technology
- Physical Design
- Circuit Designers
Implementation Review

• First, let’s think about how different instructions get executed
Instruction Fetch

- Send the Program Counter (PC) to memory
- Fetch the current instruction from memory
  - IR <= Mem[PC]
- Update the PC to the next sequential
  - PC <= PC + 4 (4-bytes per instruction)
- Optimizations
  - Instruction Caches, Instruction Prefetch
- Performance Affected by
  - Code density, Instruction size variability (CISC/RISC)
Abstract Implementation
Instruction Decode/Reg Fetch

• Decide what type of instruction we have
  • ALU, Branch, Memory
  • Decode Opcode

• Get operands from Reg File
  • A <= Regs[IR_{25..21}]; B <= Regs[IR_{20..16}];
  • Imm <= SignExtend(IR_{15..0})

• Performance Affected by
  • Regularity in instruction format, instruction length
Calculate Effective Address:

- Calculate Memory address for data
- $ALU_{output} \leq A + \text{Imm}$
- \text{LW  R10, 10(R3)}

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
</table>


Calculate Effective Address:

- Calculate target for branch/jump operation
- **BEQZ, BNEZ, J**
  - \( ALU_{output} \leq NPC + \text{Imm}; \text{cond} \leq A \text{ op } 0 \)
  - “op” is a check against 0, equal, not-equal, etc.
  - J is an unconditional
- \( ALU_{output} \leq A \)
Execution: ALU Ops

• Perform the computation
• Register-Register
  • $\text{ALU}_{\text{output}} \leq A \text{ op } B$
• Register-Immediate
  • $\text{ALU}_{\text{output}} \leq A \text{ op } \text{Imm}$
• No ops need to do effective address calc \textit{and} perform an operation on data
• Why?
Memory Access

- Take effective address, perform Load or Store
- Load
  - LMD \( \leq \text{Mem}[\text{ALU}_{\text{output}}] \)
- Store
  - \( \text{Mem}[\text{ALU}_{\text{output}}] \leq B \)
Mem Phase on Branches

- Set PC to the calculated effective address
- BEQZ, BNEZ
  - If (cond) PC <= ALU_{output} else PC <= NPC
Write-Back

- Send results back to register file
- Register-register ALU instructions
  - $\text{Regs}[\text{IR}_{15..11}] <= \text{ALU}_{\text{output}}$
- Register-Immediate ALU instruction
  - $\text{Regs}[\text{IR}_{20..16}] <= \text{ALU}_{\text{output}}$
- Load Instruction
  - $\text{Regs}[\text{IR}_{20..16}] <= \text{LMD}$
- Why does this have to be a separate step?
What is Pipelining?

• Implementation where multiple instructions are simultaneously overlapped in execution
  • Instruction processing has N different stages
  • Overlap different instructions working on different stages

• Pipelining is not new
  • Ford’s Model-T assembly line
  • Laundry – Washer/Dryer
  • IBM Stretch [1962]
  • Since the ’70s nearly all computers have been pipelined
Pipelining Advantages

- **Unpipelined**

- **Pipelined**

  
  ![Diagram showing time and latency](chart)

  - Latency
  - $1/\text{throughput}$
Representation of Pipelines

Program execution order (in instructions)

- lw $10, 20($1)
- sub $11, $2, $3

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6

<table>
<thead>
<tr>
<th>lw R10, 20(R1)</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub R11, R2, R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Pipeline Hazards

- **Hazards**
  - Situations that prevent the next instruction from executing in its designated clock cycle

- **Structural Hazards**
  - When two different instructions want to use the same hardware resource in the same cycle (resource conflict)

- **Data Hazards**
  - When an instruction depends on the result of a previous instruction that exposes overlapping of instructions

- **Control Hazards**
  - Pipelining of PC-modifying instructions (branch, jump, etc)
How to resolve hazards?

- Simple Solution: Stall the pipeline
  - Stops some instructions from executing
  - Make them wait for older instructions to complete
  - Simple implementation to “freeze” (de-assert write-enable signals on pipeline latches)
  - Inserts a “bubble” into the pipe
  - Must propagate upstream as well! Why?
### Structural Hazards

- **Two cases when this can occur**
  - Resource used more than once in a cycle (Memory, ALU)
  - Resource is not fully pipelined (FP Unit)
- **Imagine that our pipeline shares I- and D-memory**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw R10, 10(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub R11, R2, R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>add R12, R4, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>add R13, R6, R7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Structural Hazards

- **Stall**
  - Low Cost, Simple (+)
  - Increases CPI (-)
  - Try to use for rare events in high-performance CPUs

- **Duplicate Resources**
  - Decreases CPI (+)
  - Increases cost (area), possibly cycle time (-)
  - Use for cheap resources, frequent cases
    - Separate I-, D-caches, Separate ALU/PC adders, Reg File Ports
Structural Hazards

• Pipeline Resources
  • High performance (+)
  • Control is simpler than duplication (+)
  • Tough to pipeline some things (RAMs) (-)
  • Use when frequency makes it worthwhile
  • Ex. Fully pipelined FP add/multiplies critical for scientific

• Good news
  • Structural hazards don’t occur as long as each instruction uses a resource
    – At most once
    – Always in the same pipeline stage
    – For one cycle
  • RISC ISAs are designed with this in mind, reduces structural hazards
Pipeline Stalls

• What could the performance impact of unified instruction/data memory be?

Loads ~15% of instructions, Stores ~10%

Prob (Ifetch + Dfetch) = .25

\[ \text{CPI}_{\text{Real}} = \text{CPI}_{\text{Ideal}} + \text{CPI}_{\text{Stall}} = 1.0 + 0.25 = 1.25 \]
Data Hazards

- Two operands from different instructions use the same storage location
- Must appear as if instructions are executed to completion one at a time
- Three types of Data Hazards
  - Read-After-Write (RAW)
    - True data-dependence (Most important)
  - Write-After-Read (WAR)
  - Write-After-Write (WAW)
### RAW Example

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add R3, R2, R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R4, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R6, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R7, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- First Add writes to R3 in cycle 5
- Second Add reads R3 in cycle 3
- Third Add reads R3 in cycle 4
  - We would compute the wrong answer because R3 holds the “old” value
Solutions to RAW Hazards

• As usual, we have a couple of choices
• Stall whenever we have a RAW
  • Huge performance penalty, dependencies are common!
• Use Bypass/Forwarding to minimize the problem
  • Data is ready by end of EXE (Add) or MEM (Load)
  • Basic idea:
    – Add comparator for each combination of destination and source registers that can have RAW hazards (How many?)
    – Add muxes to datapath to select proper value instead of regfile
  • Only stall when absolutely necessary
Solutions to RAW Hazards:

• Two part problem: Detect the RAW, forward/stall the pipe
  • Need to keep register ID’s along with pipestages
  • Use comparators to check for hazards

• Operand 2 bypass ADD R1, R2, R3
  If (R3 == RD(MEM)) use ALUOUT(MEM)
  else (if R3 == RD(WB)) use ALUOUT (WB)
  else Use R3 from Register File
### Forwarding, Bypassing

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add R3, R2, R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R4, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R6, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R7, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Code is now “stall-free”**
- **Are there any cases where we must stall?**
### Load Use Hazards

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw R3, 10(R1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R4, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R6, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Unfortunately, we can’t forward “backward in time”

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw R3, 10(R1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R4, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R6, R3, R5</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```assembly
WBMEMEXIDIF
Add R6, R3, R5
```
```assembly
7
WB
```
```assembly
WBMEMEXIDIF
Add R4, R3, R5
```
```assembly
6
WB
```
```assembly
WBMEMEXIDIF
lw R3, 10(R1)
```
```assembly
5
MEM
```
```assembly
Memory
```
```assembly
WBMEMEXIDIF
Add R4, R3, R5
```
```assembly
4
EX
```
```assembly
WBMEMEXIDIF
lw R3, 10(R1)
```
```assembly
3
EX
```
```assembly
WBMEMEXIDIF
Add R6, R3, R5
```
```assembly
2
MEM
```
```assembly
WBMEMEXIDIF
lw R3, 10(R1)
```
```assembly
1
WB
```
Load Use Hazards

- Can the compiler help out?
  - Scheduling to avoid load followed by immediate use
- “Delayed Loads”
  - Define the pipeline slot after a load to be a “delay slot”
  - NO interlock hardware. Machine assumes the correct compiler
- Compiler attempts to schedule code to fill delay slots
- Limits to this approach:
  - Only can reorder between branches (5-6 instructions)
  - Order of loads/stores difficult to swap (alias problems)
  - Makes part of implementation architecturally visible
### Instruction Scheduling Example

```plaintext
a = b + c;
d = e – f;
```

How many cycles for each?

<table>
<thead>
<tr>
<th>No Scheduling Version</th>
<th>Scheduled Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb, b</td>
<td>LW Rb, b</td>
</tr>
<tr>
<td>LW Rc, c</td>
<td>LW Rc, c</td>
</tr>
<tr>
<td>ADD Ra, Rb, Rc</td>
<td>LW Re, e</td>
</tr>
<tr>
<td>SW a, Ra</td>
<td>ADD Ra, Rb, Rc</td>
</tr>
<tr>
<td>LW Re, e</td>
<td>LW Rf, f</td>
</tr>
<tr>
<td>LW Rf, f</td>
<td>SW a, Ra</td>
</tr>
<tr>
<td>SUB Rd, Re, Rf</td>
<td>SUB Rd, Re, Rf</td>
</tr>
<tr>
<td>SW d, Rd</td>
<td>SW d, Rd</td>
</tr>
</tbody>
</table>
Other Data Hazards: WARs

- **Write-After-Read (WAR) Hazards**
  - Can’t happen in our simple 5-stage pipeline because writes always follow reads
  - Preview: Late read, early write (auto-increment)
    
    ```
    i   DIV (R1), --, --
    i+1  ADD --, R1+, --
    ```
  - Preview: Out-of-Order reads (OOO-execution)
Other Data Hazards: WAWs

- **Write-After-Write (WAW) Hazards**
  - Can’t happen in our simple 5-stage pipeline because only one writeback stage (ALU ops go through MEM stage)
  - Preview: Slow operation followed by fast operation
    
    i    DIVF F0, --, --
    i+1  BFPT --, --, --
    i+2  ADDF F0, --, --

  - Also cache misses (they can return at odd times)

- **What about RARs?**
Control Hazards

- In base pipeline, branch outcome not known until MEM
- Simple solution – stall until outcome is known
- Length of control hazard is branch delay
  - In this simple case, it is 3 cycles (assume 10% cond. branches)
  - $CPI_{Real} = CPI_{Ideal} + CPI_{Stall} = 1.0 + 3 \text{ cycles} \times .1 = 1.3$
Control Hazards: Solutions Fast Branch Resolution

• Performance penalty could be more than 30%
  • Deeper pipelines, some code is very branch heavy
• Fast Branch Resolution
  • Adder in ID for PC + immediate targets
  • Only works for simple conditions (compare to 0)
  • Comparing two register values could be too slow

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Instr.</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +1</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +2</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Hazards: Branch Characteristics

- Integer Benchmarks: 14-16% instructions are conditional branches
- FP: 3-12%
- On Average:
  - 67% of conditional branches are “taken”
  - 60% of forward branches are taken
  - 85% of backward branches are taken
  - Why?
Control Hazards: Solutions

1. Stall Pipeline
   - Simple, No backing up, No Problems with Exceptions

2. Assume not taken
   - Speculation requires back-out logic:
     - What about exceptions, auto-increment, etc
   - Bets the “wrong way”

3. Assume taken
   - Doesn’t help in simple pipeline! (don’t know target)

4. Delay Branches
   - Can help a bit… we’ll see pro’s and con’s soon
Control Hazards: Assume Not Taken

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untaken Branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Looks good if we’re right!

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taken Branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +1</td>
<td>IF</td>
<td>flush</td>
<td>flush</td>
<td>flush</td>
<td>flush</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Target +1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Hazards: Branch Delay Slots

• Find one instruction that will be executed no matter which way the branch goes
• Now we don’t care which way the branch goes!
  • Harder than it sounds to find instructions
• What to put in the slot (80% of the time)
  • Instruction from before the branch (indep. of branch)
  • Instruction from taken or not-taken path
    – Always safe to execute? May need clean-up code (or nullifying branches)
    – Helps if you go the right way
• Slots don’t help much with today’s machines
  • Interrupts are more difficult (why? We’ll see soon)
Consider the following deeply pipelined organization for a register-memory machine:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF1</td>
<td>Begin Instruction Fetch</td>
</tr>
<tr>
<td>IF2</td>
<td>End Instruction Fetch</td>
</tr>
<tr>
<td>ID</td>
<td>Instruction Decode, Register Read</td>
</tr>
<tr>
<td>ALU1</td>
<td>Address calculation (for branches and memory references); ALU operation for register-register type instructions; branch condition evaluation</td>
</tr>
<tr>
<td>MEM1</td>
<td>Begin memory access for memory instructions; write-back for register-register instructions.</td>
</tr>
<tr>
<td>MEM2</td>
<td>Complete memory access for memory instructions;</td>
</tr>
<tr>
<td>ALU2</td>
<td>Additional ALU cycle for register-memory operations.</td>
</tr>
<tr>
<td>WB</td>
<td>Writeback for register-memory operations; assume register file reads/writes work on split cycles as in the basic DLX pipeline.</td>
</tr>
</tbody>
</table>

Question: To avoid structural hazards, how many ALUs, Reg Read/Write ports are needed?
Pipelining Example

• How many stall cycles between instructions? (do not assume forwarding)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>ALU1</td>
<td>MEM1</td>
<td>MEM2</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>ADD Rx, R1, Ry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Would forwarding help?
Pipelining Example

- How many stall cycles between instructions? (do not assume forwarding)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>ALU1</td>
<td>MEM1</td>
<td>MEM2</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>ADD Rx, R1, 10(Ry)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Pipelining Example

- How many stall cycles between instructions? (do not assume forwarding)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>ALU1</td>
<td>MEM1</td>
<td>MEM2</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>ADD Rx, Ry, 10(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelining Example

- How many stall cycles between instructions? (do not assume forwarding)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, 10(Rx)</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>ALU1</td>
<td>MEM1</td>
<td>MEM2</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>ADD Ry, R1, Rz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Pipelining Example

- **How many stall cycles between instructions? (do not assume forwarding)**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, 10(Rx), Ry</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>ALU1</td>
<td>MEM1</td>
<td>MEM2</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>ADD Rz, 40(R1), Ra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Now for the hard stuff!

- Precise Interrupts
  - What are interrupts?
  - Why do they have to be precise?

- Must have well-defined state at interrupt
  - All older instructions are complete
  - All younger instructions have not started
  - All interrupts are taken in program order
Interrupt Taxonomy

- Synchronous vs. Asynchronous (HW error, I/O)
- User Request (exception?) vs. Coerced
- User maskable vs. Nonmaskable (Ignorable)
- Within vs. Between Instructions
- Resume vs. Terminate

The difficult exceptions are resumable interrupts within instructions

- Save the state, correct the cause, restore the state, continue execution
# Interrupt Taxonomy

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Sync vs. Async</th>
<th>User Request Vs. Coerced</th>
<th>User mask vs. nonmask</th>
<th>Within vs. BetweenInsn</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Device Req.</td>
<td>Async</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Invoke O/S</td>
<td>Sync</td>
<td>User</td>
<td>Nonmask</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Tracing Instructions</td>
<td>Sync</td>
<td>User</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Sync</td>
<td>User</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Arithmetic Overflow</td>
<td>Sync</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Page Fault (not in main m)</td>
<td>Sync</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Misaligned Memory</td>
<td>Sync</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Mem. Protection Violation</td>
<td>Sync</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Using Undefined Insns</td>
<td>Sync</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Hardware/Power Failure</td>
<td>Async</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>
Interrupts on Instruction Phases

- Exceptions can occur on many different phases
- However, exceptions are only handled in WB
- Why?

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>IF</th>
<th>ID</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Overflow</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Fault (not in main memory)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Misligned Memory</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Mem. Protection Violation</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to take an exception?

1. Force a trap instruction on the next IF
2. Squash younger instructions (Turn off all writes (register/memory) for faulting instruction and all instructions that follow it)
3. Save all processor state after trap begins
   - PC-chain, PSW, Condition Codes, trap condition
   - PC-chain is length of the branch delay plus 1
4. Perform the trap/exception code then restart where we left off
Summary of Exceptions

• Precise interrupts are a headache!
• All architected state must be precise
• Delayed branches
• Preview: Out-of-Order completion
  • What if something writes-back earlier than the exception?
• Some machines punt on the problem
  • Precise exceptions only for integer pipe
  • Special “precise mode” used for debugging (10x slower)
Multicycle Operations

• Basic RISC pipeline
  • All operations take 1 cycle
• Unfortunately, not the case in real processors
  • FP add, Integer/FP Multiply can be 2-6 cycles
  • 20-50 cycles for integer/FP divide, square root
  • Cache misses can be hundreds of cycles

• Difficulties
  • Hard to pipeline
  • Differ in number of clock cycles
  • Number of operands varies
Multicycle Operations

- For example, longer latency in FP unit
- EX may continue for as long as FP takes to finish
- Assume four separate ALUs
  - Integer unit
  - FP/Integer Multiplier
  - FP Adder
  - FP/Integer Divider
- Instruction stalls all instruction behind it if it cannot proceed to EX
Multicycle Terminology

- **Initiation Interval**
  - Number of cycles that must elapse between issuing 2 operations of a given type.

- **Latency**
  - Number of cycles between an instruction that *produces* a result and an instruction that *uses* the result.

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Latency</th>
<th>Initiation Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data Memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP Multiply</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP Divide</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>
Multicycle Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVD F2, F2, F3</td>
<td></td>
<td></td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
</tr>
<tr>
<td>ADDD F10, F2,</td>
<td></td>
<td></td>
<td></td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Integer unit

FP/integer multiply

FP adder

FP/integer divider

MEM

WB
New Issues
- Structural Hazards on non-pipelined units
- Register writes per cycle can > 1 (what is the max?)
- WAW hazards are possible – are WAR?
- Instruction complete out of order (what is the problem?)
- Longer latency ops (what is the problem?)
Structural Hazards

- FP Divide not pipelined
  - Too much hardware needed
- Register Write port contention
  - Can fix through replicating hardware (multiported register file)
  - Can fix through stalls in ID
    - Track WB usage in ID with WB reservation bits (shift register)
    - Simplest scheme (all hardware is in ID)
  - Can fix through stalls when entering MEM or WB
    - Less hardware, but multiple stall points
WAW Hazards

- Why weren’t they a problem before?

<table>
<thead>
<tr>
<th>MULD F0, F4, F6</th>
<th>IF</th>
<th>ID</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F0, F4, F6</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Are they a problem?
  - Why generate 2 writes without an intervening read?
    - Branch Delay slots, Instruction that trap conflict with trap handler
  - Could happen, so we must check
WAW Hazard Logic

- **Solutions:**
  - Stall younger instruction writeback
    - Intuitive solution, fairly simple implementation
  - Squash older instruction writeback
    - Why not? The younger will overwrite it anyway…
      - No stalling/performance loss
    - What about precise exceptions?
Multicycle: Summary of Hazards

- Three more checks must be performed in ID
  - Check for structural hazards
    - Make sure functional unit (FU) is not busy
    - Make sure Reg Write port is available
  - Check for RAW data hazard
    - Wait until sources are not a pending destination in any pipeline registers not available before instruction needs result
  - Check for WAW data hazards
    - Determine if any instruction in A1...A4, or M1...M7, or D has the same register destination as the instruction. If so stall!
- Concepts are the same, logic is more complicated
Multicycle: Out of Order Completion

• What could go wrong here?
  DIVD  F0, F2, F4
  ADDD  F10, F10, F8
  SUBD  F12, F12, F14

• Solutions
  1. Ignore the problem (1960s, early 70s)
  2. Buffer the results (with forwarding) until all earlier ops complete
     – History files, Future Files (Can be combined with out-of-order issue)
  3. Imprecise exception with enough info to allow trap-handlers to clean up
  4. Hybrid: Allow issue to continue only if all older instructions have cleared their exception points