Appendix A: Pipelining

- **Pipelining** is an implementation technique whereby multiple instructions are overlapped in execution
  - Takes advantage of parallelism that exists among the actions needed to execute an instruction
    - *fetch* instruction from memory
    - *decode* to figure out what to do
    - *read* source operands
    - *execute*
    - *write* results

Datapath vs Control

- **Datapath**: Storage, FU, interconnect sufficient to perform the desired functions
  - *Inputs are Control Points*
  - *Outputs are signals*
- **Controller**: State machine to orchestrate operation on the data path
  - *Based on desired function and signals*
Five Stages of RISC Implementation

<table>
<thead>
<tr>
<th>Step</th>
<th>ALU</th>
<th>Load/store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>IR</td>
<td>NPC = PC + d</td>
<td></td>
</tr>
<tr>
<td>Instruction Decode/</td>
<td>A = Reg[rs]</td>
<td>R = Reg[rt]</td>
<td></td>
</tr>
<tr>
<td>Register Fetch</td>
<td>Imm = Sign-extended immediate field of IR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution/Effective address</td>
<td>ALU output = A + imm</td>
<td>ALU output = A + imm</td>
<td></td>
</tr>
<tr>
<td>Memory Access/Branch</td>
<td>LMD = Mem[ALU output]</td>
<td>If (cond) PC = ALU output</td>
<td></td>
</tr>
<tr>
<td>Write Back</td>
<td>Reg[rs] = ALU Output or Reg[rt] = ALU output</td>
<td>Reg[rt] = LMD</td>
<td></td>
</tr>
</tbody>
</table>

A Simple implementation of RISC IS

IF
ID
EX
MEM
WB

IF
ID
EX
MEM
WB

IR <= Mem[PC];
PC <= PC + 4
Reg[IRn] <= Reg[IRn] op [Mem[IRn]]
R-type (most ALU) Instructions

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>Instruction decode/ register read</td>
<td>Execute</td>
<td>Memory access</td>
<td>Write back</td>
</tr>
<tr>
<td>Next PC</td>
<td>Program Counter</td>
<td>Inst. Reg.</td>
<td>Registers</td>
<td>ALU</td>
</tr>
</tbody>
</table>

ALU with Imm(Addi, ...)Instructions

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>Instruction decode/ register read</td>
<td>Execute</td>
<td>Memory access</td>
<td>Write back</td>
</tr>
<tr>
<td>Next PC</td>
<td>Program Counter</td>
<td>Inst. Reg.</td>
<td>Registers</td>
<td>ALU</td>
</tr>
</tbody>
</table>

Load (LD, LW, …) Instructions

Store (SD, SW, …) Instructions
Branch (beq, bne, beqz) Instructions

IF Instruction fetch
ID Instruction decode/register read
EX Execution
MEM Memory access
WB Write back

Program Counter
Next PC
Inst. Reg.

Basic RISC Pipelining

- Basic idea:
  - Each instruction spends 1 clock cycle in each of the 5 execution stages.
  - During 1 clock cycle, the pipeline can process (in different stages) 5 different instructions.

<table>
<thead>
<tr>
<th>Instruction number</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i + 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alternative Visualization - component

Visualization with Pipeline Registers
# Adding Pipeline Registers

![Pipeline Registers Diagram](image)

# Description of Pipeline Stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF/ID</td>
<td>PC ← Mem(IPC);</td>
</tr>
<tr>
<td>ID</td>
<td>ID/EX.A ← Regs[IF/ID.IR6, ...]; ID/EX.B ← Regs[IF/ID.IR1, ...];</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ID/EX.IA ← IF/ID.IB; ID/EX.IB ← IF/ID.IB;</td>
</tr>
</tbody>
</table>

## Branch Instruction

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF/ID</td>
<td>PC ← Mem(IPC);</td>
</tr>
<tr>
<td>ID</td>
<td>ID/EX.A ← Regs[IF/ID.IR6, ...]; ID/EX.B ← Regs[IF/ID.IR1, ...];</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ID/EX.IA ← IF/ID.IB; ID/EX.IB ← IF/ID.IB;</td>
</tr>
</tbody>
</table>

## Load or Store Instruction

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>EX/EX.MEM.1B ← EX/EX.IB;</td>
</tr>
<tr>
<td>EX</td>
<td>EX/EX.MEM.1A ← ID/EX.IA;</td>
</tr>
<tr>
<td>EX</td>
<td>EX/EX.MEM.1A ← ID/EX.IA;</td>
</tr>
<tr>
<td>EX</td>
<td>EX/EX.MEM.1A ← ID/EX.IA;</td>
</tr>
<tr>
<td>EX</td>
<td>EX/EX.MEM.1A ← ID/EX.IA;</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.1B ← EX/EX.MEM.1B;</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.1B ← EX/EX.MEM.1B;</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.1B ← EX/EX.MEM.1B;</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.1B ← EX/EX.MEM.1B;</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.1B ← EX/EX.MEM.1B;</td>
</tr>
<tr>
<td>WB</td>
<td>WB ← Regs[MEM/WB.IR6, ...];</td>
</tr>
<tr>
<td>WB</td>
<td>WB ← Regs[MEM/WB.IR6, ...];</td>
</tr>
<tr>
<td>WB</td>
<td>WB ← Regs[MEM/WB.IR6, ...];</td>
</tr>
<tr>
<td>WB</td>
<td>WB ← Regs[MEM/WB.IR6, ...];</td>
</tr>
</tbody>
</table>
Pipeline Hazards

- Hazards are circumstances which may lead to stalls (delays, “bubbles”) in the pipeline if not addressed.

- Three major types:
  - **Structural hazards:**
    - Not enough HW resources to keep all instrs. moving.
    - Multiple instructions trying to use same hardware
  - **Data hazards:**
    - Data results of earlier instructions not available yet.
    - Instruction depends on result of prior instruction still in the pipeline
  - **Control hazards:**
    - Control decisions resulting from earlier instr. (branches) not yet made; don't know which new instr. to execute.

---

**Structural Hazard Example**

The processor has a combined instruction+data memory with only 1 read port.
Hazards Produce “Bubbles”

How do you “bubble” the pipe?

Textual View

<table>
<thead>
<tr>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>Load instruction</td>
</tr>
<tr>
<td>Instruction i+1</td>
</tr>
<tr>
<td>Instruction i+2</td>
</tr>
<tr>
<td>Instruction i+3</td>
</tr>
<tr>
<td>Instruction i+4</td>
</tr>
<tr>
<td>Instruction i+5</td>
</tr>
<tr>
<td>Instruction i+6</td>
</tr>
</tbody>
</table>

A pipeline stalled for a structural hazard – a load with only one memory port.
Speed Up Equation for Pipelining

\[
\text{Speedup} = \frac{\text{Avg. inst. time (unpipelined)}}{\text{Avg. inst. time (pipelined)}}
\]

\[
\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}}
\]

For simple RISC pipeline, Ideal CPI = 1:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}}
\]

Performance of Pipelines (continued)

- If \( \text{cycle time}_{\text{unpipelined}} = \text{cycle time}_{\text{pipelined}} \)
- Speedup = \( \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall cycles per instruction}} \)
- Example:
  - Unpipelining: 1ns clock, 4 cycles for ALU and branches operations, 5 cycles for memory operations. Relative frequencies of these operations are 40%, 20% and 40%. Pipelining the processor adds 0.2 ns of overhead to the clock. What is speedup?
Example: Dual-port vs. Single-port

- **Machine A**: Dual ported memory (“Harvard Architecture”)
- **Machine B**: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

Which machine is faster, and by how much?

- SpeedUpA = Pipeline Depth / (1 + 0) x (clock_{unpipe} / clock_{pipe})
  = Pipeline Depth
- SpeedUpB = Pipeline Depth / (1 + 0.4 x 1) x (clock_{unpipe} / (clock_{unpipe} / 1.05))
  = (Pipeline Depth / 1.4) x 1.05
  = 0.75 x Pipeline Depth

SpeedUpA / SpeedUpB = Pipeline Depth / (0.75 x Pipeline Depth) = 1.33 → Machine A is 1.33 times faster

Data Hazard Example: Hazard on R1

```
ADD R1,R2,R3
SUB R4,R1,R5
AND R6,R1,R7
OR R8,R1,R9
XOR R10,R1,R11
```
Solution: Forwarding for Data Hazards

ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11

Another Forwarding Example (LW-SW)

ADD R1, R2, R3

SW R4, 12(R1)

LR R4, 0(R1)

SW R4, 12(R1)
Three Types of Data Hazards

- Let \( i \) be an earlier instruction, \( j \) a later one.
  - **RAW** (read after write)
    - \( j \) tries to read a value before \( i \) writes it
  - **WAW** (write after write)
    - \( i \) and \( j \) write to the same place, but in the wrong order.
    - Only occurs if more than 1 pipeline stage can write
  - **WAR** (write after read)
    - \( j \) writes a new value to a location before \( i \) has read the old one.
    - Only occurs if writes can happen before reads in pipeline (in-order).
An Unavoidable Stall

Stalling in midst of instruction
Data Hazard Prevention

- A clever compiler can often reschedule instructions to avoid a stall.
  
  - A simple example:
    
    Original code:
    
    1. lw r2, 0(r4)
    2. add r1, r2, r3 ← Note: Stall happens here.
    3. lw r5, 4(r4)
    
    Transformed code:
    
    1. lw r2, 0(r4)
    2. lw r5, 4(r4)
    3. add r1, r2, r3 ← No stall needed.

Simple RISC Pipeline Stall Statistics
Control (Branch) Hazards

- Suppose the new PC value is not computed until the MEM stage.

10: beq r1, r3, 36
14: and r2, r3, r5
18: or r6, r1, r7
22: add r8, r1, r9
36: xor r10, r1, r11

--- What do you do with the 3 instructions in between? How do you do it?

Control (Branch) Hazards

- Then we must stall 3 clocks after every branch!
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Four Branch Hazard Alternatives

- #1: Stall until branch direction is clear
- #2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
- #3: Predict Branch Taken
  - 53% MIPS branches taken on average
  - But haven’t calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome
Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER a following instruction
  - branch instruction
  - sequential successor
  - sequential successor
  - sequential successor
  - sequential successor
  - Branch delay of length n
  - branch target if taken

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Predict-Not-Taken

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i + 1</td>
<td>IF</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Delayed Branches

Machine code sequence:
- Branch instruction
- Delay slot instruction(s)
- Post-branch instructions

Branch is taken (if taken) at this point

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction (i + 1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction (i + 1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Scheduling the Branch-Delay Slot

(a) From before
- ALU
- ID
- EX
- MEM
- WB

(b) From target
- ALU
- ID
- EX
- MEM
- WB

(c) From tail through
- ALU
- ID
- EX
- MEM
- WB
**Delayed Branch**

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

**Performance of Pipelines with Stalls**

- Speedup
  - Avg. inst. time (unpipelined) / Avg. inst. time (pipelined)
  - ...
  - CPI unpipelined / CPI pipelined
  - ...
  - Pipeline Depth / (1 + Pipeline stall cycles per instruction)
  - Pipeline Depth / (1 + Branch frequency x Branch penalty)
Evaluating Branch Alternatives

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

Pipeline speedup = \( \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}} \)

Detecting Control Signals

<table>
<thead>
<tr>
<th>Situation</th>
<th>Example code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>LD  R1, 45(R2) DADD R5, R6, R7 DSUB R8, R6, R7 OR R9, R6, R7</td>
<td>No hazards</td>
</tr>
<tr>
<td>Dependence requiring stall</td>
<td>LD  R1, 45(R2) DADD R5, R1, R7 DSUB R8, R6, R7 OR R9, R6, R7</td>
<td>Detect use of R1 during ID of DADD and stall</td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>LD  R1, 45(R2) DADD R5, R6, R7 DSUB R8, R1, R7 OR R9, R6, R7</td>
<td>Detect use of R1 during ID of DSUB and set mux control signal that accepts result from bypass path</td>
</tr>
<tr>
<td>Dependence with accesses in order</td>
<td>LD  R1, 45(R2) DADD R5, R6, R7 DSUB R8, R6, R7 OR R9, R1, R7</td>
<td>No action required</td>
</tr>
</tbody>
</table>
## A.4 What makes pipelining hard to implement

- **Exceptions**
  - Power failure, I/O device request, arithmetic overflow, page fault, ...

- **ISA complications**
  - Complex addressing modes and instructions

There are 5 instructions executing in 5 stage pipeline when an interrupt occurs:

- How to stop and restart the pipeline?
- Who caused the interrupt?

<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem that causes the interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic interrupt</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation</td>
</tr>
</tbody>
</table>

- **Solution #1**
  - Interrupt status vector per instruction
  - Defer check until last stage, kill state update if exception

- **Solution #2**
  - Interrupt ASAP
  - Restart everything that is incomplete

- **Simultaneous exceptions in more than one pipeline stage, e.g.,**
  - Load with data page fault in MEM stage
  - Add with instruction page fault in IF stage
  - Add fault will happen BEFORE load fault
A.5 Pipeline for Multi-cycle instructions

- Floating point instructions take many cycles to complete.
- Often implemented by multiple executions of the EX stage.
- Not all instructions will take the same amount of cycles to finish!
- Latency:
  - number of intervening cycles between an instruction that
    produces a result and instruction that uses the result
  - Usually: depth of the EX stage - 1
- Initiation interval:
  - Number of cycles that must elapse between issuing two
    operations of a given type
- Multi-cycle instructions/pipelines increase the probability for
  occurring WAW and RAW hazards.

Multi-cycle Operations for FP

- [Diagram showing pipeline stages: IF, ID, EX, MEM, WB, with EX stages labeled as FP integer multiply, FP adder, FP divide.]
Pipelined Multiple-Issue FPU

Multicycle Instructions

<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>25</td>
</tr>
</tbody>
</table>
Typical FP Code Seq. with Stalls

- MUL.D stalls
  - 1 cycle in IF waiting for new value of F4 from MEM stage of L.D
- ADD.D stalls
  - 1 cycle in IF waiting for MUL.D to leave ID,
  - then 6 cycles in ID waiting for new F0 to be returned by MUL.D stage M7.
- S.D stalls
  - 6 cycles in IF waiting for ADD.D to leave ID,
  - then 2 cycles in EX waiting for new F2 to be returned by ADD.D stage A4, then 1 more cycle in EX waiting for ADD.D to clear MEM stage.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17</td>
</tr>
<tr>
<td>MUL.D</td>
<td>IF ID EX ME WB</td>
</tr>
<tr>
<td>ADD.D</td>
<td>IF ID A1 A2 A3 A4 ME WB</td>
</tr>
<tr>
<td>S.D</td>
<td>IF ID EX ME WB</td>
</tr>
</tbody>
</table>

FPU Pipelining Issues

- Notice instructions may complete out-of-order:
  - MUL.D IF ID M1 M2 M3 M4 M5 M6 M7 ME WB
  - ADD.D IF ID A1 A2 A3 A4 ME WB
  - L.D IF ID EX ME WB
  - S.D IF ID EX ME WB

- Stall for RAW is longer and more frequent (Fig. A33)
- WAW is possible; WAR is not (why?)
- Structural hazards in MEM & WB stages.
- Out-of-order completion impacts exception handling.
- Multiple WBs are likely (Fig. A.34)
A.34

3 Instructions want to perform a write back to the FP register simultaneously

- No structural hazard exists for MEM.

<table>
<thead>
<tr>
<th>MUL.D 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>ME</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D. F2,(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solutions

- A. Conflicting register or memory writes:
  - Increase no. of write ports: costly, rarely used.
  - Detect structural hazard (and stall) in ID
  - Detect collisions before MEM or WB stage: can select which inst to stall, longest latency one has higher probability of RAW hazard.
Solutions

B. WAW Hazards: (see example A.34)
- If L.D is a cycle earlier and has F2 as a dest reg its results would be overwritten by ADD.D!
- If an inst between ADD.D and L.D uses F2 then it’s a RAW and no WAW. Most WAWs occur when useless inst are executed! But, such sequences do occur in reasonable code.

Options:
- delay L.D until ADD.D is in MEM
- detect hazard, make ADD.D a no-op and issue L.D right away.

C. Handling hazards at Issue (ID) stage:
- Check structural hazards: functional unit, WB port
- Check RAW hazards: Issue with forwarding
- Check WAW hazards: Not issue to make sure write in order
FP Stall Statistics

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Typical</th>
<th>FP SPEC benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide</td>
<td>9.0</td>
<td>25.9</td>
</tr>
<tr>
<td>Multiply</td>
<td>5.7</td>
<td>12.1</td>
</tr>
<tr>
<td>Add/subtract</td>
<td>6.0</td>
<td>24.5</td>
</tr>
<tr>
<td>Compare</td>
<td>6.9</td>
<td>8.8</td>
</tr>
<tr>
<td>Other</td>
<td>3.2</td>
<td>4.5</td>
</tr>
</tbody>
</table>

ISA Design Impacts Pipelining

- Variable instruction lengths & run times:
  - Introduces delays due to pipeline inequities.
  - Complicates hazard-detection & precise exceptions

- Sophisticated addressing modes:
  - Post-autoincrement complicates hazard detection, restarting, introduces WAR & WAW hazards.
  - Multiple-indirect modes complicate pipeline control & timing.

- Self-modifying code:
  - What if you overwrite an instruction in the pipe?

- Implicit condition codes: WAR hazards, restarts