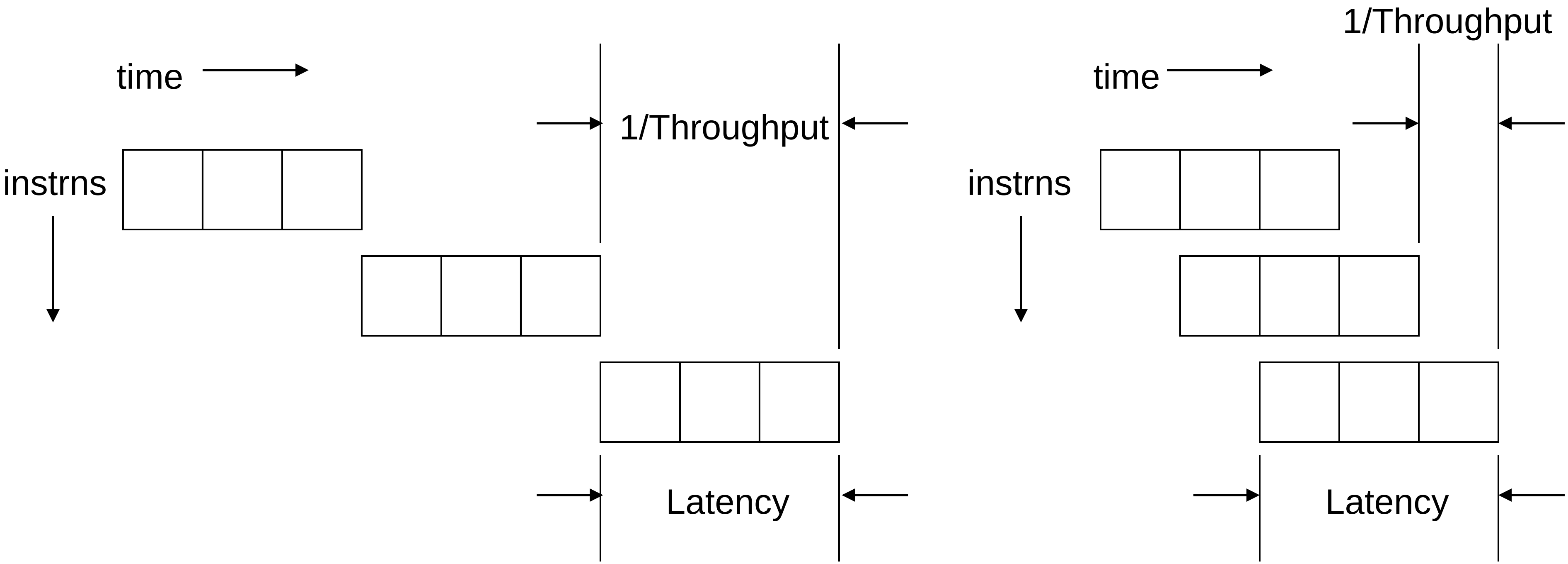
# *Appendix C: Pipelining: Basic and Intermediate Concepts*

- Key ideas and simple pipeline (Section C.1)
- Hazards (Sections C.2 and C.3)
    - Structural hazards
    - Data hazards
    - Control hazards
- Exceptions (Section C.4)
- Multicycle operations (Section C.5)

# *Pipelining - Key Idea*




***Ideally***,

$$Time_{pipeline} = \frac{Time_{sequential}}{Pipeline\ Depth}$$

$$Speedup = \frac{Time_{sequential}}{Time_{pipeline}} = Pipeline\ Depth$$

# *Practical Limit 1 – Unbalanced Stages*

Consider an instruction that requires $n$ stages

$s_1, s_2, \ldots, s_n$, taking time $t_1, t_2, \ldots, t_n$.

Let $T = \Sigma t_i$

Without pipelining

Throughput =

Latency =

With an n-stage pipeline

Throughput =

Latency =

Speedup

# Practical Limit 2 - Overheads

Let $\Delta > 0$ be extra delay per stage

e.g., latches

$\Delta$ limits the useful depth of a pipeline.

With an n stage pipeline

$$Throughput = \frac{1}{\Delta + max\ t_i} < \frac{n}{T}$$

$$Latency = n \times (\Delta + max\ t_i) \geq n\Delta + T$$

$$Speedup = \frac{\Sigma t_i}{\Delta + max\ t_i} < n$$

# *Example*

Let $t_{1,2,3}$ = 8, 12, 10 $ns$ *and* $\Delta$ = 2 $ns$

Throughput =

Latency =

Speedup =

# Practical Limit 3 - Hazards

$$\text{Pipeline Speedup} = \frac{\text{Time}_{\text{sequential}}}{\text{Time}_{\text{pipeline}}} = \frac{\text{CPI}_{\text{sequential}}}{\text{CPI}_{\text{pipeline}}} \times \frac{\text{Cycle Time}_{\text{sequential}}}{\text{Cycle Time}_{\text{pipeline}}}$$

If we ignore cycle time differences:

$$\text{CPI}_{\text{ideal-pipeline}} = \frac{\text{CPI}_{\text{sequential}}}{\text{Pipeline Depth}}$$

$$\text{Pipeline Speedup} = \frac{\text{CPI}_{\text{ideal-pipeline}} \times \text{Pipeline Depth}}{\text{CPI}_{\text{ideal-pipeline}} + \text{Pipeline stall cycles}}$$

# *Pipelining a Basic RISC ISA*

Assumptions:

- Only loads and stores affect memory
  - Base register + immediate offset = effective address
- ALU operations
  - Only access registers
  - Two sources – two registers, or register and immediate
- Branches and jumps
  - Address = PC + offset
  - Comparison between a register and zero

***The last assumption is different from the 6$^{th}$ edition of the text and results in a slightly different pipeline. We will discuss reasons and implications in class.***

# A Simple Five Stage RISC Pipeline

Pipeline Stages

- IF – Instruction Fetch
- ID – Instruction decode, register read, branch computation
- EX – Execution and Effective Address
- MEM – Memory Access
- WB – Writeback

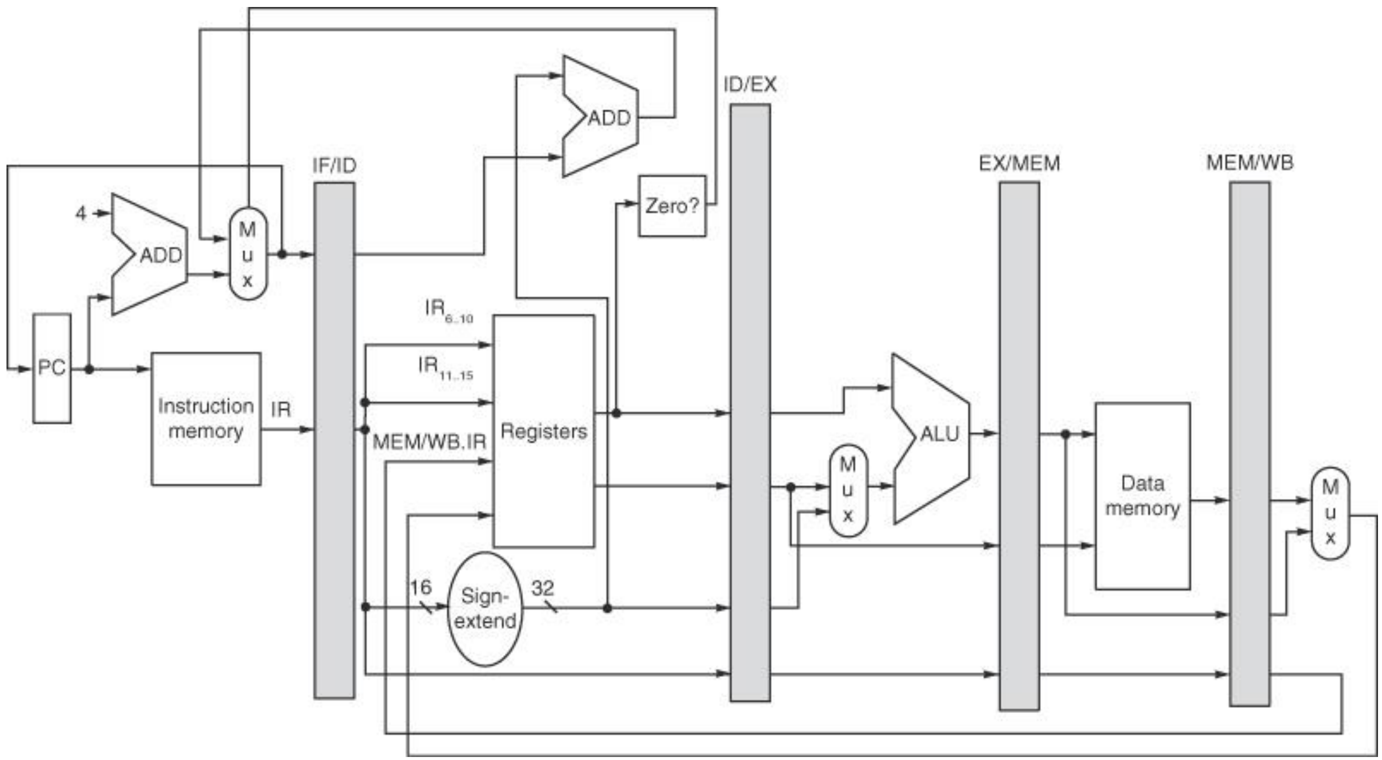
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| i | IF | ID | EX | MEM | WB | | | | |
| i+1 | | IF | ID | EX | MEM | WB | | | |
| i+2 | | | IF | ID | EX | MEM | WB | | |
| i+3 | | | | IF | ID | EX | MEM | WB | |
| i+4 | | | | | IF | ID | EX | MEM | WB |

Pipelining really isn't this simple

# *A Naive Pipeline Implementation*

**Figure C.28**

**of 5th edition**






Pipelining really isn't this simple

# *Hazards*

Hazards

Structural Hazards

Data Hazards

Control Hazards

# Handling Hazards

Pipeline interlock logic

- Detects hazard and takes appropriate action

Simplest solution: stall

- Increases CPI
- Decreases performance

Other solutions are harder, but have better performance

# *Structural Hazards*

When two *different* instructions want to use the *same* hardware resource in the *same* cycle

Stall (cause bubble)

+ Low cost, simple

  Increases CPI

  Use for rare events

  E.g., ??

Duplicate Resource

+ Good performance

  Increases cost (and maybe cycle time for interconnect)

  Use for cheap resources

  E.g., ALU and PC adder

# *Structural Hazards, cont.*

Pipeline Resource

+ Good performance

  Often complex to do

  Use when simple to do

  E.g., write & read registers every cycle

Structural hazards are avoided if each instruction uses a resource

- At most once
- Always in the same pipeline stage
- For one cycle
- ($\Rightarrow$ no cycle where two instructions use the same resource)

# Structural Hazard Example

Loads/stores (MEM) use same memory port as instrn fetches (IF)

30% of all instructions are loads and stores

Assume $CPI_{old}$ is 1.5

```
      1   2   3   4   5   6   7   8   9
i     IF  ID  EX  MEM WB <— a load
i+1       IF  ID  EX  MEM WB
i+2           IF  ID  EX  MEM WB
i+3               **  IF  ID  EX  MEM WB
i+4                       IF  ID  EX MEM WB
```

How much faster could a new machine with two memory ports be?

# Data Hazards

When two different instructions use the same location, it must appear as if instructions execute one at a time and in the specified order

```
i      ADD r1,r2,
i+1    SUB r2,,r1
i+2    OR  r1,--,
```

Read-After-Write (RAW, data-dependence)

A true dependence

MOST IMPORTANT

Write-After-Read (WAR, anti-dependence)

Write-After-Write (WAW, output-dependence)

NOT: Read-After-Read (RAR)

# Example Read-After-Write Hazards

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | r1 written | |
| ADD r1,_,_ | IF | ID | EX | MEM | WB | |
| | | | | | NOT OK! | |
| SUB _, r1,_ | | IF | ID | EX | MEM | WB |
| | | | r1 read | | | |
| | | | | | r1 written | |
| LW r1,_,_ | IF | ID | EX | MEM | WB | |
| | | | | | NOT OK! | |
| SUB _, r1,_ | | IF | ID | EX | MEM | WB |
| | | | r1 read | | | |
| | | | | memory written | | |
| SW r1,100(r0) | IF | ID | EX | MEM | WB | |
| | | | | | CORRECT! | |
| LW r2,100(r0) | | IF | ID | EX | MEM | WB |
| | | | | | memory read | |

(Unless LW instrn is at address 100(r0))

# *RAW Solutions*

Solutions must first detect RAW, and then ...

Stall

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | r1 written | | | |
| ADD r1,_,_ | IF | ID | EX | MEM | WB | | | |
| SUB _, r1,_ | | IF | ID | stall | stall | EX | MEM | WB |
| | | | | | r1 read | | | |

(Assumes registers written then read each cycle)

+ Low cost, simple

  Increases CPI (plus 2 per stall in 5 stage pipeline)

  Use for rare events

# *RAW Solutions*

Bypass/Forward/ShortCircuit

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | data available | | r1 written | |
| ADD r1,_,_ | IF | ID | EX | MEM | WB | |
| SUB _, r1,_ | | IF | ID | EX | MEM | WB |
| | | | r1 read | data used | | |

Use data before it is in register

- \+ Reduces (avoids) stalls
- More complex
- Critical for common RAW hazards

# Bypass, cont.

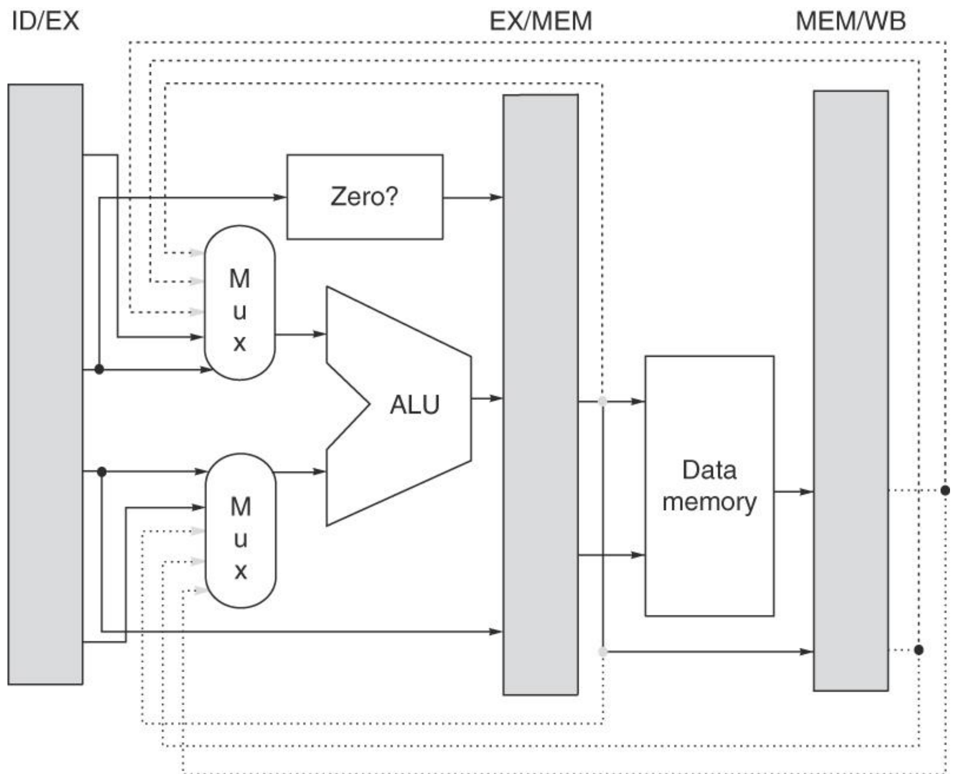



**Figure C.27**

**5th edition**



- Additional hardware
  - Muxes supply correct result to ALU
- Additional control
  - Interlock logic must control muxes

# RAW Solutions, cont.

Hybrid solution sometimes required:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | data available | r1 written | | |
| LW r1,_,_ | IF | ID | EX | MEM | WB | | |
| SUB _, r1,_ | | IF | ID | stall | EX | MEM | WB |
| | | | | r1 read | data used | | |

One cycle bubble if result of load used by next instruction

Pipeline scheduling at compile time

Moves instructions to eliminate stalls

# Pipeline Scheduling Example

Before:

```
a = b + c;   LW   Rb,b
             LW   Rc,c
                        <- stall
             ADD  Ra,Rb,Rc
             SW   a, Ra

d = e - f;   LW   Re,e
             LW   Rf,f
                        <- stall
             SUB  Rd,Re, Rf
             SW   d, Rd
```

After:

```
a = b + c;   LW   Rb,b
             LW   Rc,c
             LW   Re,e
             ADD  Ra,Rb,Rc

d = e - f;   LW   Rf,f
             SW   a, Ra
             SUB  Rd,Re,Rf
             SW   d, Rd
```

# Other Data Hazards

```
i     ADD r1,r2,
i+1   SUB r2,,r1
i+2   OR  r1,,
```

Write-After-Read (WAR, anti-dependence)

```
i     MULT , (r2), r1 /* RX mult */
i+1   LW   , (r1)+ /* autoincrement */
```

Write-After-Write (WAW, output-dependence)

```
i     DIVF fr1, ,  /* slow */
i+1
i+2   ADDF fr1, ,  /* fast */
```

# Control Hazards

When an instruction affects which instructions are executed *next* --
branches, jumps, calls

```
i     BEQZ r1,#8
i+1   SUB  ,,
       . . .
i+8   OR   ,,
i+9   ADD  ,,


1    2    3    4    5    6    7    8    9
i    IF   ID   EX   MEM  WB
i+1       IF (aborted)
i+8            IF   ID   EX   MEM  WB
i+9                 IF   ID   EX   MEM
```

Handling control hazards is very important

# *Handling Control Hazards*

Branch Prediction

- Guess the direction of the branch
- Minimize penalty when right
- May increase penalty when wrong

Techniques

- Static – At compile time
- Dynamic – At run time

Static Techniques

- Predict NotTaken
- Predict Taken
- Delayed Branches

Dynamic techniques and more powerful static techniques later…

# Handling Control Hazards, cont.

Predict NOT-TAKEN Always

NotTaken:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| i | IF | ID | EX | MEM | WB | | | |
| i+1 | | IF | ID | EX | MEM | WB | | |
| i+2 | | | IF | ID | EX | MEM | WB | |
| i+3 | | | | IF | ID | EX | MEM | WB |

Taken:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| i | IF | ID | EX | MEM | WB | | | |
| i+1 | | IF (aborted) | | | | | | |
| i+8 | | | IF | ID | EX | MEM | WB | |
| i+9 | | | | IF | ID | EX | MEM | WB |

Don't change machine state until branch outcome is known

Basic pipeline: State always changes late (WB)

# Handling Control Hazards, cont.

Predict TAKEN Always

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| i | IF | ID | EX | MEM | WB | | | |
| i+8 | | ‘IF’ | ID | EX | MEM | WB | | |
| i+9 | | | IF | ID | EX | MEM | WB | |
| i+10 | | | | IF | ID | EX | MEM | WB |

Must know what address to fetch at BEFORE branch is decoded

Not practical for our basic pipeline

## *Handling Control Hazards, cont.*

Delayed branch

- Execute next instruction regardless (of whether branch is taken)
- What do we execute in the DELAY SLOT?

# Delay Slots

Fill from before branch

When:

Helps:

Fill from target

When:

Helps:

Fill from fall through

When:

Helps:

# *Delay Slots (Cont.)*

Cancelling or nullifying branch

- Instruction includes direction of prediction
- Delay instruction squashed if wrong prediction
- Allows second and third case of previous slide to be more aggressive

# *Comparison of Branch Schemes*

Suppose 14% of all instructions are branches

Suppose 65% of all branches are taken

Suppose 50% of delay slots usefully filled

CPIpenalty = % branches $\times$

(% Taken $\times$ Taken-Penalty + % Not-Taken $\times$ Not-Taken penalty)

| Branch Scheme | Taken Penalty | Not-Taken Penalty | CPI Penalty |
|---|---|---|---|
| Basic Branch | 1 | 1 | .14 |
| Not-Taken | 1 | 0 | .09 |
| Taken0 | 0 | 1 | .05 |
| Taken1 | 1 | 1 | .14 |
| Delayed Branch | .5 | .5 | .07 |

# Real Processors

MIPS R4000: 3 cycle branch penalty

- First cycle: cancelling delayed branch (cancel if not taken)
- Next two cycles: Predict not taken

Recent architectures:

- Because of deeper pipelines, delayed branches not very useful
- Processors rely more on hardware prediction (will see later) or may include both delayed and nondelayed branches

# *Interrupts*

Interrupts (a.k.a. faults, exceptions, traps) often require

- Surprise jump
- Linking of return address
- Saving of PSW (including CCs)
- State change (e.g., to kernel mode)

Some examples

- Arithmetic overflow
- I/O device request
- O.S. call
- Page fault

Make pipelining hard

# *One Classification of Interrupts*

1a. Synchronous

function of program and memory state

(e.g., arithmetic overflow, page fault)

1b. Asynchronous

external device or hardware malfunction

(printer ready, bus error)

# Handling Interrupts

Precise Interrupts (Sequential Semantics)

- Complete instrns before offending one
- Squash (effects of) instrns after
- Save PC
- Force trap instrn into IF

Must handle simultaneous interrupts

- IF –
- ID –
- EX –
- MEM –
- WB –

Which interrupt should be handled first?

# Interrupts, cont.

Example: Data Page Fault

```
      1    2    3    4    5    6    7    8
i     IF   ID   EX   MEM  WB
i+1        IF   ID   EX   MEM  WB <— page fault (MEM)
i+2             IF   ID   EX   MEM  WB <— squash
i+3                  IF   ID   EX   MEM  WB <— squash
i+4                       IF   ID   EX   MEM  WB <— squash
i+5                  trap —> IF   ID   EX   MEM  WB
i+6             trap handler —> IF   ID   EX   MEM  WB
```

Preceding instruction already complete

Squash succeeding instructions

    Prevent from modifying state

'Trap' instruction jumps to trap handler

Hardware saves PC in IAR

Trap handler must save IAR

# *Interrupts, cont.*

Example: Arithmetic Exception

```
     1   2   3   4   5   6   7   8
i    IF  ID  EX  MEM WB
i+1      IF  ID  EX  MEM WB
i+2          IF  ID  EX  MEM WB <- Exception (EX)
i+3              IF  ID  EX  MEM WB <- squash
i+4                  IF  ID  EX  MEM WB <- squash
i+5              trap -> IF  ID  EX  MEM WB
i+6          trap handler -> IF  ID  EX  MEM WB
```

Let preceding instructions complete

Squash succeeding instruction

# Interrupts, cont.

Example: Illegal Opcode

```
        1    2    3    4    5    6    7    8
i       IF   ID   EX   MEM  WB
i+1          IF   ID   EX   MEM  WB
i+2               IF   ID   EX   MEM  WB
i+3                    IF   ID   EX   MEM  WB <- ill. op (ID)
i+4                         IF   ID   EX   MEM  WB <- squash
i+5                    trap -> IF   ID   EX   MEM  WB
i+6               trap handler -> IF   ID   EX   MEM  WB
```

Let preceding instructions complete

Squash succeeding instruction

# *Interrupts, cont.*

Example: Out-of-order Interrupts

```
      1    2    3    4    5    6    7    8
i     IF   ID   EX   MEM  WB   <- page fault (MEM)
i+1        IF   ID   EX   MEM  WB   <- page fault (IF)
i+2             IF   ID   EX   MEM  WB
i+3                  IF   ID   EX   MEM  WB
```

Which page fault should we take?

For precise interrupts – Post interrupts on a status vector associated with instruction, disable later writes in pipeline

- Check interrupt bit on entering WB
- Longer latency

For imprecise interrupts – Handle immediately

- Interrupts may occur in different order than on a sequential machine
- May cause implementation headaches

# *Interrupts, cont.*

Other complications

- Odd bits of state (e.g., CCs)
- Earlywrites (e.g., autoincrement)
- Outoforder execution

Interrupts come at random times

- The frequent case isn't everything
- The rare case MUST work correctly

# *Multicycle Operations*

- Not all operations complete in one cycle
  - Floating point arithmetic is inherently slower than integer arithmetic
  - 2 to 4 cycles for multiply or add
  - 20 to 50 cycles for divide
- Extend basic 5-stage pipeline
  - EX stage may repeat multiple times
  - Multiple function units
  - Not pipelined for now

# *Handling Multicycle Operations*

Four Functional Units

- EX: Integer unit
- E*: FP/integer multiplier
- E+: FP adder
- E/: FP/integer divider

Assume

- EX takes one cycle & all FP units take 4
- Separate integer and FP registers
- All FP arithmetic from FP registers

Worry about

- Structural hazards
- RAW hazards & forwarding
- WAR & WAW between integer & FP ops

# Simple Multicycle Example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| int | IF | ID | EX | MEM | WB | | | | | | |
| fp* | | IF | ID | E* | E* | E* | E* | MEM | WB | | |
| int | | | IF | ID | EX | MEM | WB? | (1) | | | |
| fp/ | | | | IF | ID | E/ | E/ | E/ | E/ | MEM | WB |
| int | | | | | IF | ID | EX | ** | MEM | WB | (2) |
| fp/ | | | | | (3) | IF | ID | ** | ** | E/ | E/ |
| int | | | | | | (4) | IF | ** | ** | ID | EX |

Notes

(1) WAW possible only if?

(2) Stall forced by?

(3) Stall forced by?

(4) Stall forced by?

# *FP Instruction Issue*

Check for RAW data hazard (in ID)

- Wait until source registers are not used as destinations by instructions in EX that will not be available when needed

Check for forwarding

- Bypass data from other stages, if necessary

Check for structural hazard in function unit

- Wait until function unit is free (in ID)

Check for structural hazard in MEM / WB

- Instructions stall in ID
- Instructions stall before MEM
  - Static priority (e.g., FU with longest latency)

# FP Instruction Issue (Cont.)

Check for WAW hazards

```
DIVF F0, F2, F4
SUBF F0, F8, F10
```

SUBF completes first

(1) Stall SUBF

(2) Abort DIVF's WB

WAR hazards?

# More Multicycle Operations

Problems with Interrupts

```
DIVF F0, F2, F4
ADDF F2, F8, F10
SUBF F6, F4, F10
```

ADDF and SUBF complete before DIVF

Out-of-order completion

- Possible imprecise interrupt

What happens if DIVF generates an exception after ADDF and SUBF complete??

We'll discuss solutions later