



ECE/CS 552: Pipeline Hazards

© Prof. Mikko Lipasti

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

Pipeline Hazards

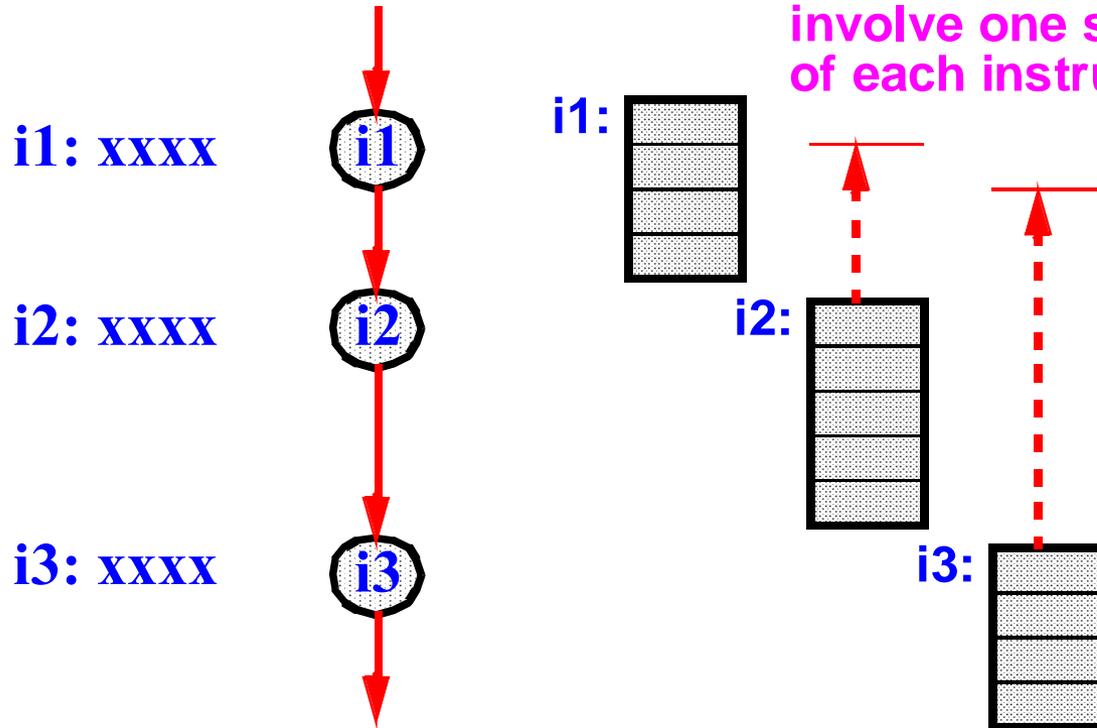
- Forecast
 - Program Dependences
 - Data Hazards
 - Stalls
 - Forwarding
 - Control Hazards
 - Exceptions

Sequential Execution Model

- MIPS ISA requires the appearance of *sequential execution*
 - *Precise exceptions*
 - True of most general purpose ISAs

Program Dependences

A true dependence between two instructions may only involve one subcomputation of each instruction.



The implied sequential precedences are an overspecification. It is sufficient but not necessary to ensure program correctness.

Program Data Dependences

- True dependence (RAW)

- j cannot execute until i produces its result

$$D(i) \cap R(j) \neq \phi$$

- Anti-dependence (WAR)

- j cannot write its result until i has read its sources

$$R(i) \cap D(j) \neq \phi$$

- Output dependence (WAW)

- j cannot write its result until i has written its result

$$D(i) \cap D(j) \neq \phi$$

Control Dependences

- Conditional branches
 - Branch must execute to determine which instruction to fetch next
 - Instructions following a conditional branch are control dependent on the branch instruction

Example (quicksort/MIPS)

```
#      for (; (j < high) && (array[j] < array[low]) ; ++j );  
#      $10 = j  
#      $9 = high  
#      $6 = array  
#      $8 = low
```

```
bge     done, $10, $9  
mul     $15, $10, 4  
addu    $24, $6, $15  
lw      $25, 0($24)  
mul     $13, $8, 4  
addu    $14, $6, $13  
lw      $15, 0($14)  
bge     done, $25, $15
```

cont:

```
addu    $10, $10, 1
```

...

done:

```
addu    $11, $11, -1
```

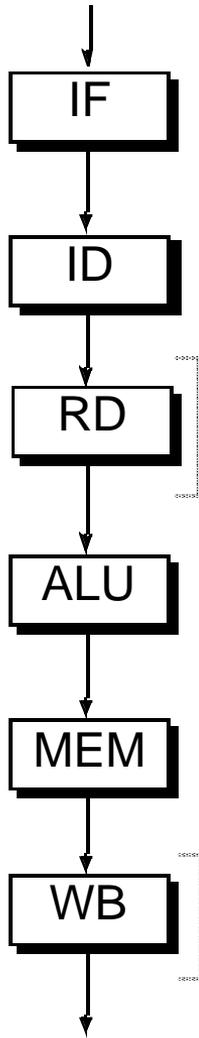
Pipeline Hazards

- Pipeline hazards
 - Potential violations of program dependences
 - Must ensure program dependences are not violated
- Hazard resolution
 - Static: compiler/programmer guarantees correctness
 - Dynamic: hardware performs checks at runtime
- Pipeline interlock
 - Hardware mechanism for dynamic hazard resolution
 - Must detect and enforce dependences at runtime

Pipeline Hazards

- Necessary conditions:
 - WAR: write stage earlier than read stage
 - Is this possible in IF-RD-EX-MEM-WB ?
 - WAW: write stage earlier than write stage
 - Is this possible in IF-RD-EX-MEM-WB ?
 - RAW: read stage earlier than write stage
 - Is this possible in IF-RD-EX-MEM-WB?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipeline Hazard Analysis



- Memory hazards
 - RAW: Yes/No?
 - WAR: Yes/No?
 - WAW: Yes/No?
- Register hazards
 - RAW: Yes/No?
 - WAR: Yes/No?
 - WAW: Yes/No?

RAW Hazard

- Earlier instruction produces a value used by a later instruction:
 - add \$1, \$2, \$3
 - sub \$4, \$5, \$1

| | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cycle: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 1 | 1 | 1 |
| Instr: | | | | | | | | | | 0 | 1 | 2 | 3 |
| add | F | D | X | M | W | | | | | | | | |
| sub | | F | D | X | M | W | | | | | | | |

RAW Hazard - Stall

- Detect dependence and stall:
 - add \$1, \$2, \$3
 - sub \$4, \$5, \$1

| | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cycle: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 1 | 1 | 1 |
| Instr: | | | | | | | | | | 0 | 1 | 2 | 3 |
| add | F | D | X | M | W | | | | | | | | |
| sub | | | | | | F | D | X | M | W | | | |

Control Dependence

- One instruction affects which executes next
 - sw \$4, 0(\$5)
 - bne \$2, \$3, loop
 - sub \$6, \$7, \$8

| | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cycle: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 1 | 1 | 1 |
| Instr: | | | | | | | | | | 0 | 1 | 2 | 3 |
| sw | F | D | X | M | W | | | | | | | | |
| bne | | F | D | X | M | W | | | | | | | |
| sub | | | F | D | X | M | W | | | | | | |



Control Dependence - Stall

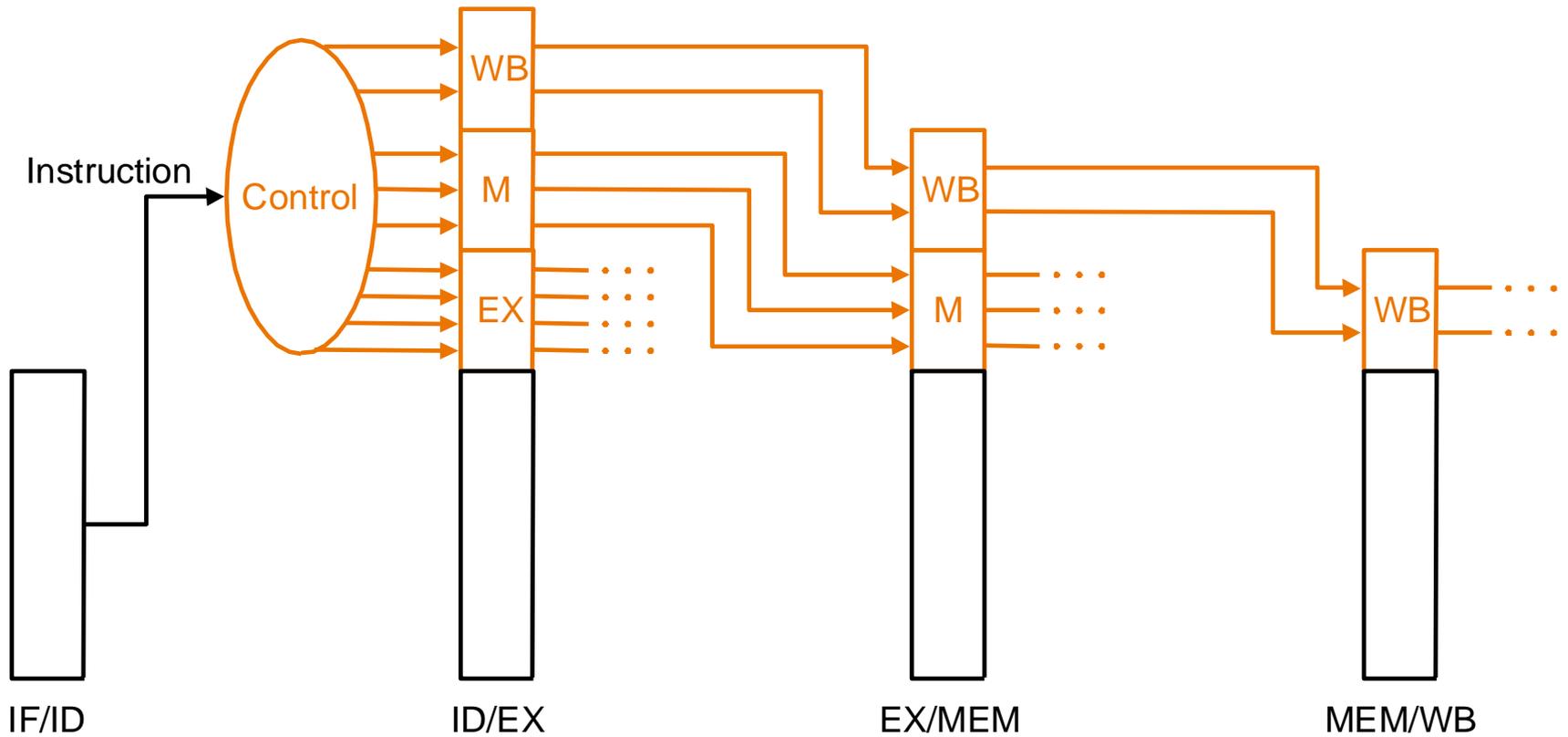
- Detect dependence and stall
 - sw \$4, 0(\$5)
 - bne \$2, \$3, loop
 - sub \$6, \$7, \$8

| | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cycle: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 1 | 1 | 1 |
| Instr: | | | | | | | | | | 0 | 1 | 2 | 3 |
| sw | F | D | X | M | W | | | | | | | | |
| bne | | F | D | X | M | W | | | | | | | |
| sub | | | | | F | D | X | M | W | | | | |

Pipelined Control

- Controlled by different instructions
- Decode instructions and pass the signals down the pipe
- Control sequencing is embedded in the pipeline
 - No explicit FSM
 - Instead, distributed FSM

Pipelined Control



RAW Hazards

- Must first detect RAW hazards
 - Pipeline analysis proves that WAR/WAW don't occur

ID/EX.WriteRegister = IF/ID.ReadRegister1

ID/EX.WriteRegister = IF/ID.ReadRegister2

EX/MEM.WriteRegister = IF/ID.ReadRegister1

EX/MEM.WriteRegister = IF/ID.ReadRegister2

MEM/WB.WriteRegister = IF/ID.ReadRegister1

MEM/WB.WriteRegister = IF/ID.ReadRegister2

RAW Hazards

- Not all hazards because
 - WriteRegister not used (e.g. sw)
 - ReadRegister not used (e.g. addi, jump)
 - Do something only if necessary

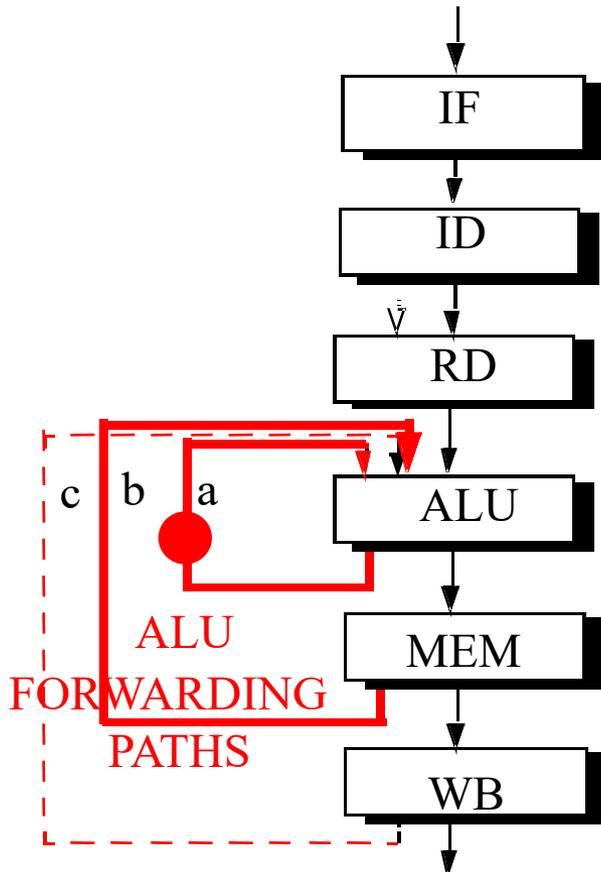
RAW Hazards

- Hazard Detection Unit
 - Several 5-bit comparators
- Response? Stall pipeline
 - Instructions in IF and ID stay
 - IF/ID pipeline latch not updated
 - Send 'nop' down pipeline (called a bubble)
 - PCWrite, IF/IDWrite, and nop mux

RAW Hazard Forwarding

- A better response – forwarding
 - Also called bypassing
- Comparators ensure register is read after it is written
- Instead of stalling until write occurs
 - Use mux to select forwarded value rather than register value
 - Control mux with hazard detection logic

Forwarding Paths (ALU instructions)

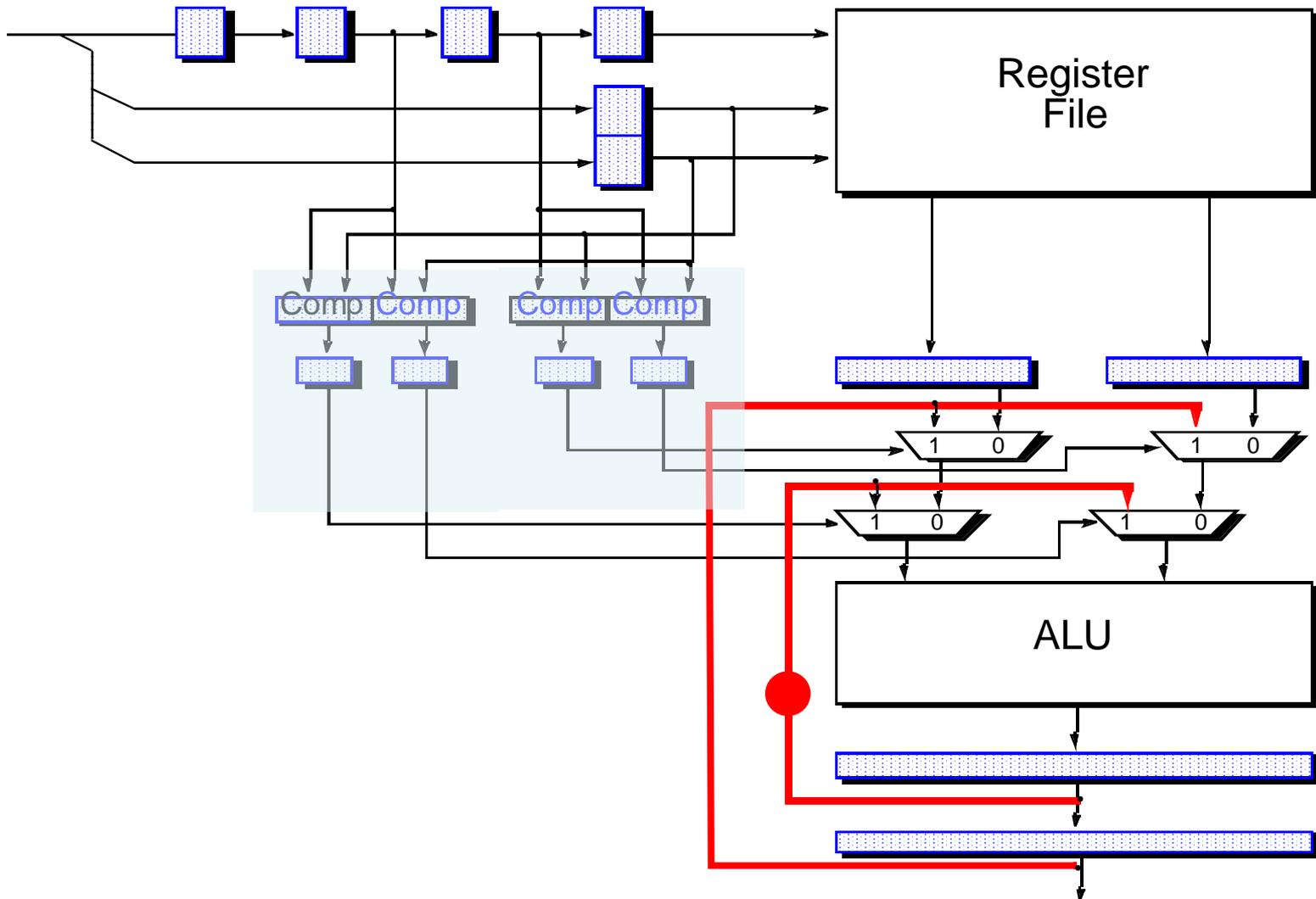


| | | |
|---|---|---|
| $i+1: \leftarrow R1$ | $i+2: \leftarrow R1$ | $i+3: \leftarrow R1$ |
| $i: R1 \leftarrow$ | $i+1: \leftarrow R1$ | $i+2: \leftarrow R1$ |
| | $i: R1 \leftarrow$ | $i+1: \leftarrow$ |
| $(i \rightarrow i+1)$ Forwarding via Path a | $(i \rightarrow i+2)$ Forwarding via Path b | $(i \rightarrow i+3)$ i writes $R1$ before $i+3$ reads $R1$ |

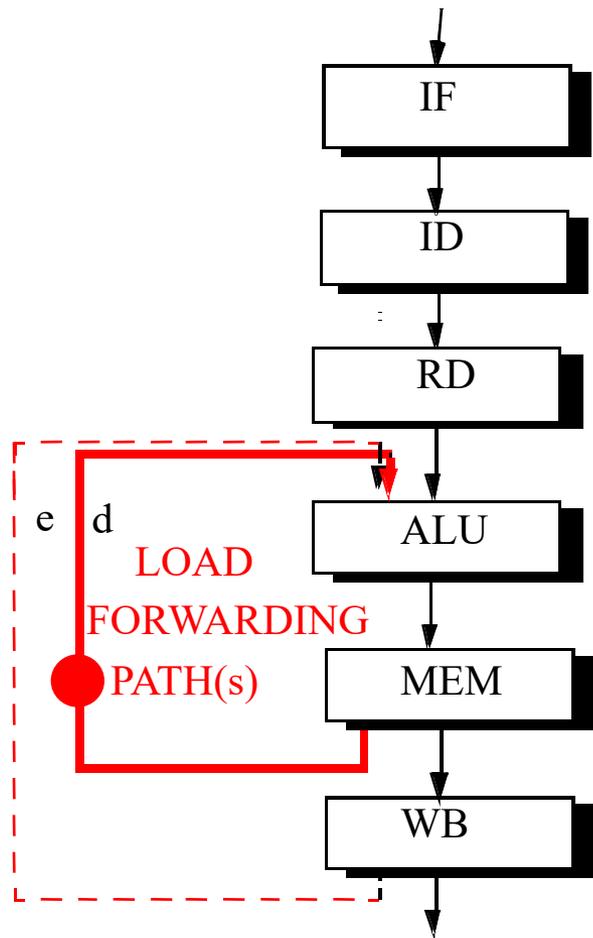
Write before Read RF

- Register file design
 - 2-phase clocks common
 - Write RF on first phase
 - Read RF on second phase
- Hence, same cycle:
 - Write \$1
 - Read \$1
- No bypass needed
 - If read before write or DFF-based, need bypass

ALU Forwarding



Forwarding Paths (Load instructions)



| | | |
|--------------------------------------|---|--|
| | | |
| | | |
| $i+1: \leftarrow R1$ | $i+1: \leftarrow R1$ | $i+2: \leftarrow R1$ |
| $i:R1 \leftarrow MEM[]$ | — | $i+1: \leftarrow R1$ |
| | $i:R1 \leftarrow MEM[]$ | — |
| | | $i:R1 \leftarrow MEM[]$ |
| $(i \rightarrow i+1)$ Stall $i+1$ | $(i \rightarrow i+1)$ Forwarding via Path d | $(i \rightarrow i+2)$ i writes $R1$ before $i+2$ reads $R1$ |

Control Flow Hazards

- Control flow instructions
 - branches, jumps, jals, returns
 - Can't fetch until branch outcome known
 - Too late for next IF

Control Flow Hazards

- What to do?
 - Always stall
 - Easy to implement
 - Performs poorly
 - $1/6^{\text{th}}$ instructions are branches
 - each branch takes 3 cycles
 - $\text{CPI} = 1 + 3 \times 1/6 = 1.5$ (lower bound)

Control Flow Hazards

- Predict branch not taken
- Send sequential instructions down pipeline
- Kill instructions later if incorrect
- Must stop memory accesses and RF writes
- Late flush of instructions on misprediction
 - Complex
 - Global signal (wire delay)

Control Flow Hazards

- Even better but more complex
 - Predict taken
 - Predict both (eager execution)
 - Predict one or the other dynamically
 - Adapt to program branch patterns
 - Lots of chip real estate these days
 - Core i7, ARM A15
 - Current research topic
 - More later, covered in detail in ECE752

Control Flow Hazards

- Another option: delayed branches
 - Always execute following instruction
 - “delay slot” (later example on MIPS pipeline)
 - Put useful instruction there, otherwise ‘nop’
- A mistake to cement this into ISA
 - Just a stopgap (one cycle, one instruction)
 - Superscalar processors (later)
 - Delay slot just gets in the way

Exceptions and Pipelining

- add \$1, \$2, \$3 overflows
- A surprise branch
 - Earlier instructions flow to completion
 - Kill later instructions
 - Save PC in EPC, set PC to EX handler, etc.
- Costs a lot of designer sanity
 - 554 teams that try this sometimes fail

Exceptions

- Even worse: in one cycle
 - I/O interrupt
 - User trap to OS (EX)
 - Illegal instruction (ID)
 - Arithmetic overflow
 - Hardware error
 - Etc.
- Interrupt priorities must be supported

Pipeline Hazards

- Program Dependences
- Data Hazards
 - Stalls
 - Forwarding
- Control Hazards
- Exceptions