Some Advanced Pipelining Concepts – Appendix A (extracts)

RAW Data Hazard

Time (clock cycles)

Instr. Order

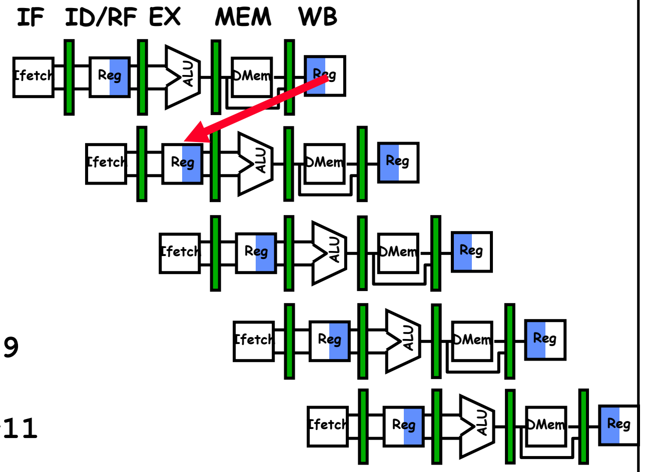
```
add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

or   r8,r1,r9

xor r10,r1,r11
```



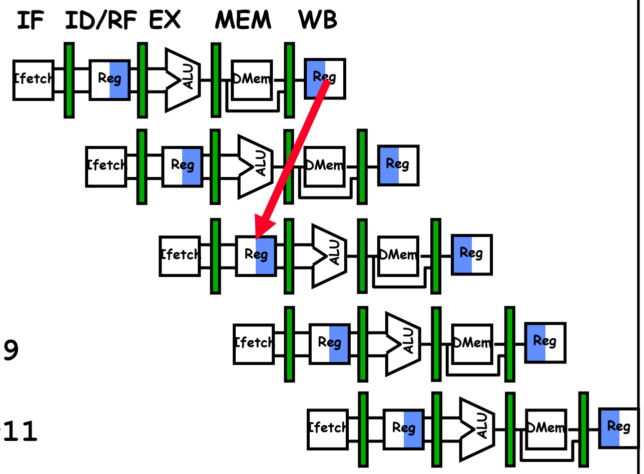
Adapted from Hennessey and Patterson, Morgan Kaufman, Pub

A two-instruction RAW hazard

Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```



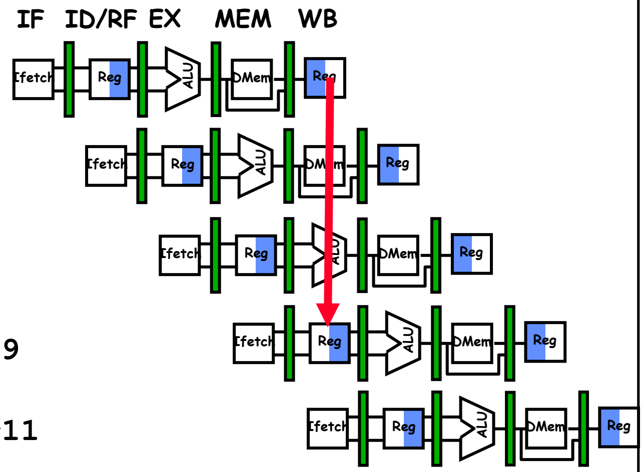
Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

A 3-instruction RAW hazard

Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```

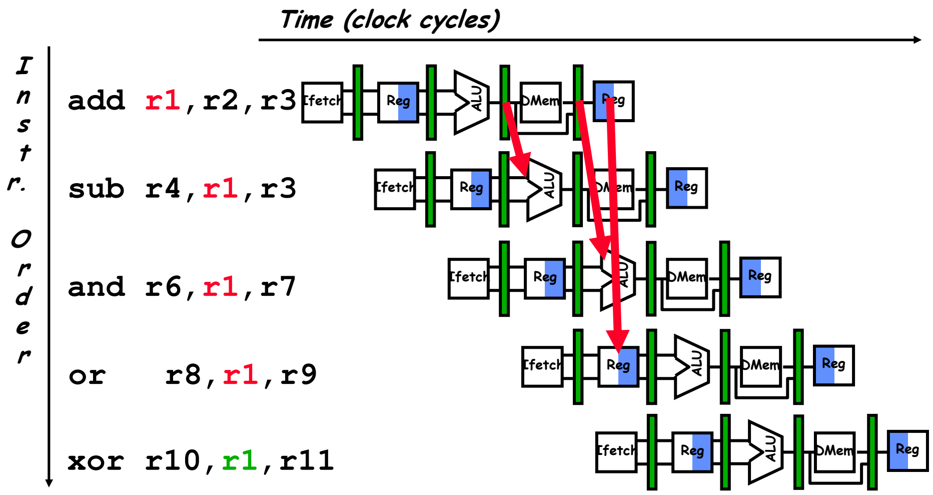


Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

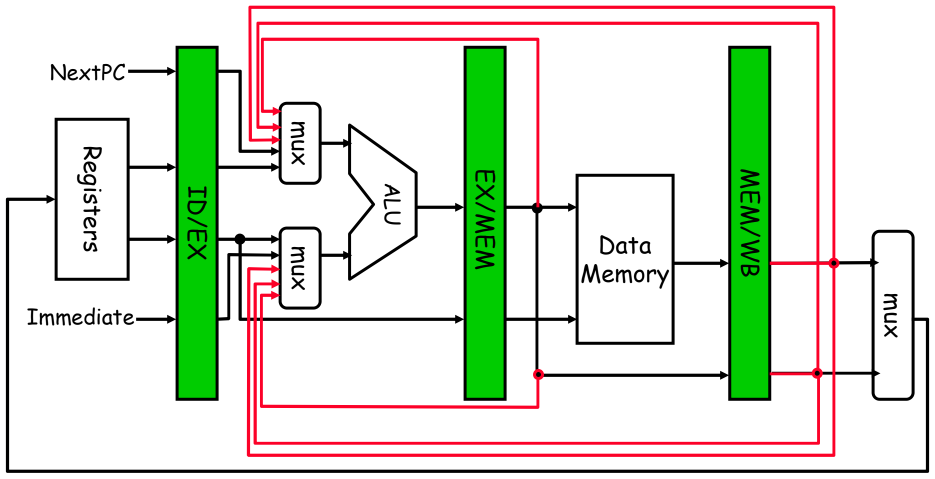
Coping with RAW hazards: Forwarding

- Addresses RAW data hazard
- The idea:
  - When a RAW hazard is present, instead of waiting until the first instruction actually writes the new value to the register…
    - » take the updated value directly when it is available in the first instruction's execution (
    - » and inject it as one of the inputs to the later instruction's EX stage
- Obviously, this requires modifying the hardware design
  - Add'l control logic to detect the hazard
  - Datapath to forward the updated value to earlier stages

Forwarding to Avoid Data Hazard

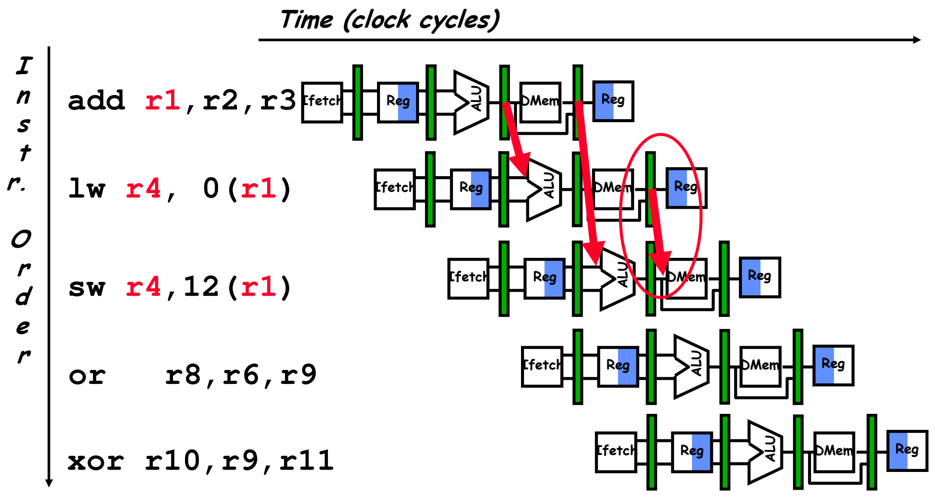


HW Change for Forwarding



Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

Forwarding to Avoid LW-SW Data Hazard



```
add r1,r2,r3
lw  r4, 0(r1)
sw  r4,12(r1)
or   r8,r6,r9
xor r10,r9,r11
```

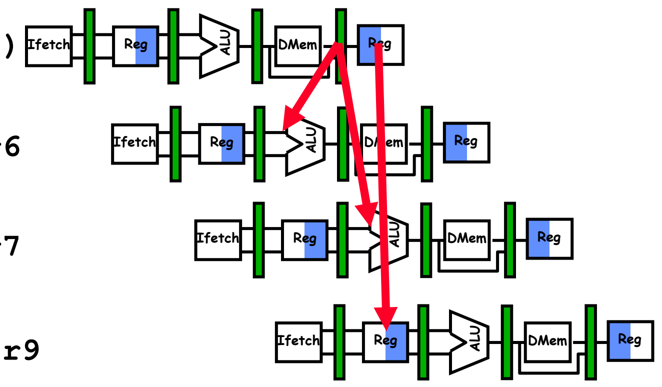
Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

Data Hazard Even with Forwarding

Time (clock cycles)

Instr. Order

```
lw r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or   r8,r1,r9
```



Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:		Fast code:	
LW	Rb,b	LW	Rb,b
LW	Rc,c	LW	Rc,c
ADD	Ra,Rb,Rc	LW	Re,e
SW	a,Ra	ADD	Ra,Rb,Rc
LW	Re,e	LW	Rf,f
LW	Rf,f	SW	a,Ra
SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
SW	d,Rd	SW	d,Rd

Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches: Three Stage Stall

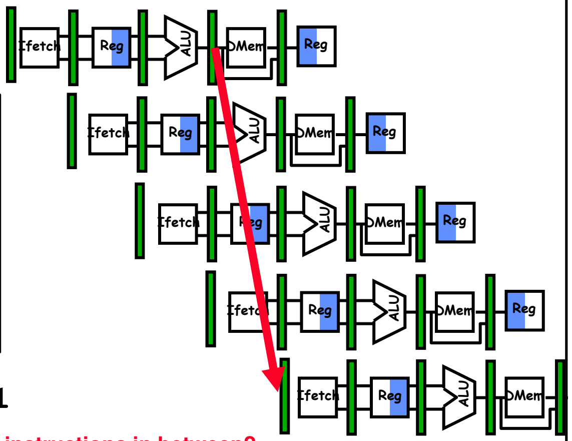
```
10: beq r1,r3,36

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?

If CPI = 1, 30% branch,
Stall 3 cycles => new CPI = 1.9!

Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

Goals of a branch stall reduction strategy

Determine if the branch is taken sooner,

AND

Compute the “taken” branch address earlier

Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

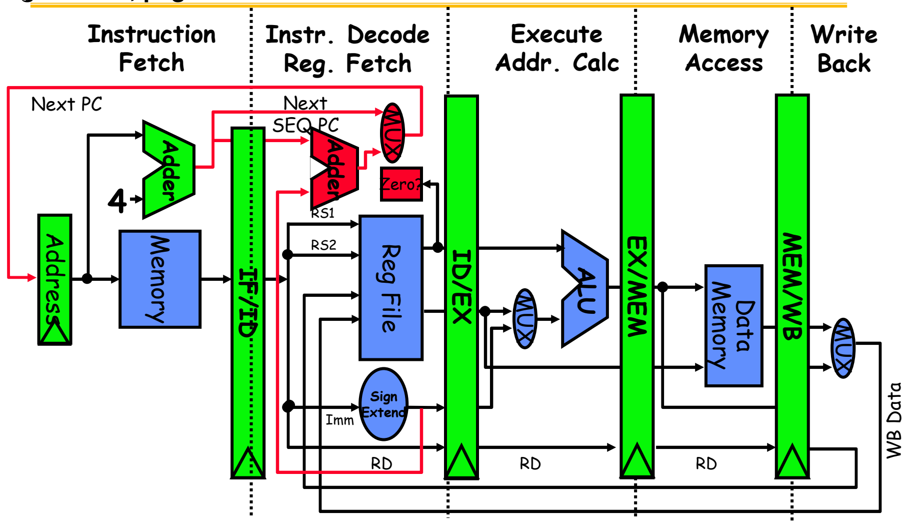
One approach

- We could change the ISA to substitute beqz (branch if reg=0) and bnez (branch if reg <> 0) for the beq and bne instructions.
- Then:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

Pipelined MIPS Datapath

Figure A.24, page A-38



- Interplay of instruction set design and cycle time.

Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub

For general Branches: 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

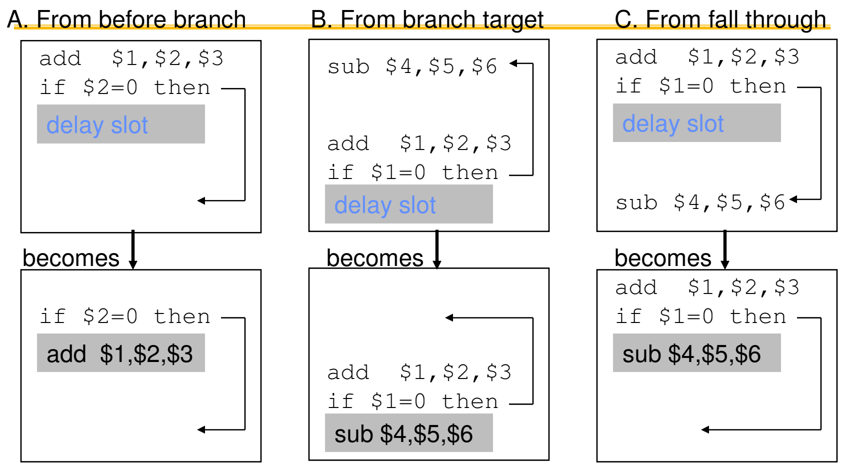
- Define branch to take place AFTER a following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Introduce a delay by changing the program to insert n instructions that would:
(a) have to be executed whether or not the branch is taken OR
(b) would not produce an incorrect result

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

Scheduling Branch Delay Slots (Fig A.14)



- **A is the best choice, fills delay slot & reduces instruction count (IC)**
- **In B, the `sub` instruction may need to be copied, increasing IC**
- **In B and C, must be okay to execute `sub` when branch fails**

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 processors go to deeper pipelines and multiple issue, the branch delay grows and needs 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**

Evaluating Branch Alternatives

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

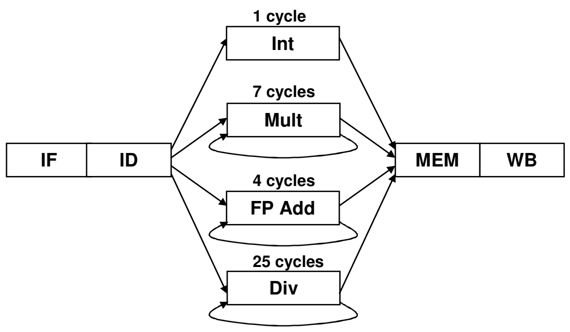
Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

Scheduling scheme	***Branch penalty***	***CPI***	***speedup v. unpipelined***	***speedup v. stall***
Stall pipeline	**3**	**1.60**	**3.1**	**1.0**
Predict taken	**1**	**1.20**	**4.2**	**1.33**
Predict not taken	**1**	**1.14**	**4.4**	**1.40**
Delayed branch	**0.5**	**1.10**	**4.5**	**1.45**

Extending the MIPS pipeline for multicycle operations

- **Floating-point operations are inherently more complex than integer operations**
- **Setting clock cycle time so that the EX phase of all instructions (including floating point) completes in one cycle will make the cycle impossibly slow**
- **To get around this:**
  1. **Allow the EX cycle to repeat as many times as needed to complete the operation, and**
  2. **Provide multiple parallel execution units**

The concept (example)



ID modified to:

- determine which EX unit an instruction should use

- hold instructions until data hazards resolved

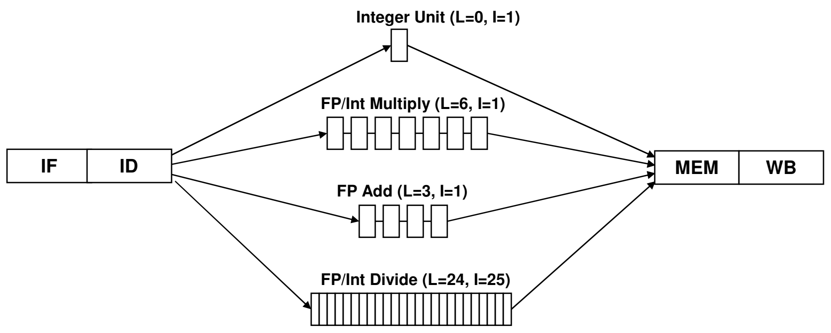
Adapted from Hennessey and Patterson, Morgan Kaufman, Pub

Latency and Initiation Interval

- Latency
  - Number of cycles that must elapse between:
    - » the end of the EX phase of the instruction of interest and
    - » An instruction that uses that instruction's result
- Initiation interval
  - Number of cycles that must elapse between two instructions that use the same EX unit

Adapted from Hennessey and Patterson, Morgan Kaufman, Pub

A more realistic approach (example)



Adapted from Hennessey and Patterson,
Morgan Kaufman, Pub