


1001 *Pipelining*

ENGR 3410 - Computer Architecture
Fall 2010

Pipelining

Example: Doing the laundry

Ann, Brian, Cathy, & Dave 
each have one load of clothes to wash, dry, and fold

Washer takes 30 minutes



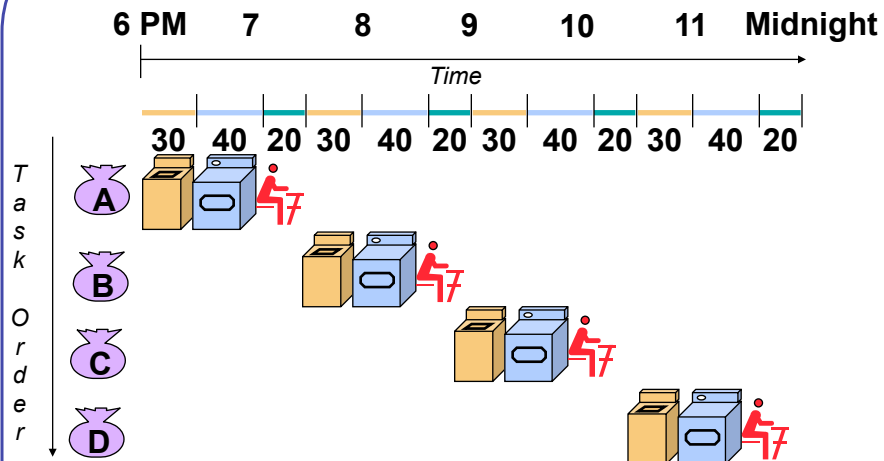
Dryer takes 40 minutes



"Folder" takes 20 minutes



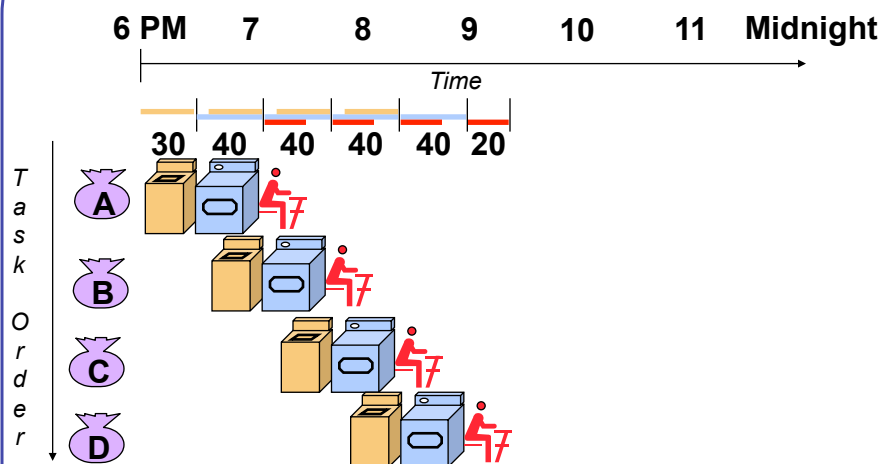
Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

2

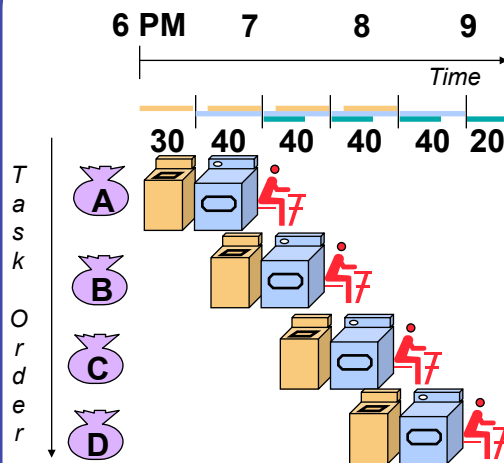
Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

3

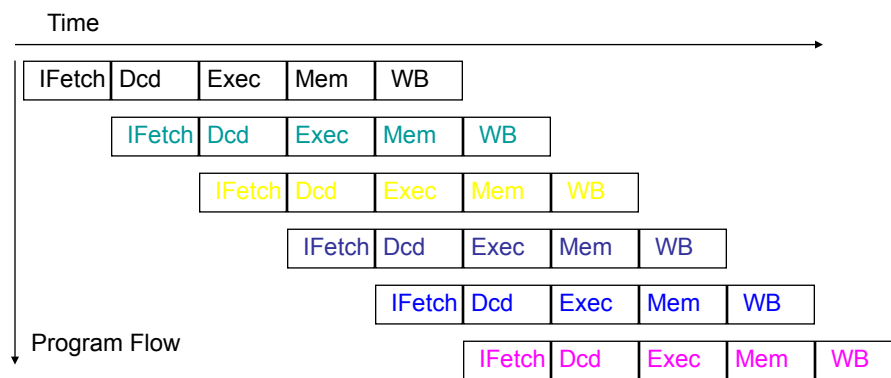
Pipelining Lessons



- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to "**fill**" pipeline and time to "**drain**" it reduces speedup
- Stall for Dependences

4

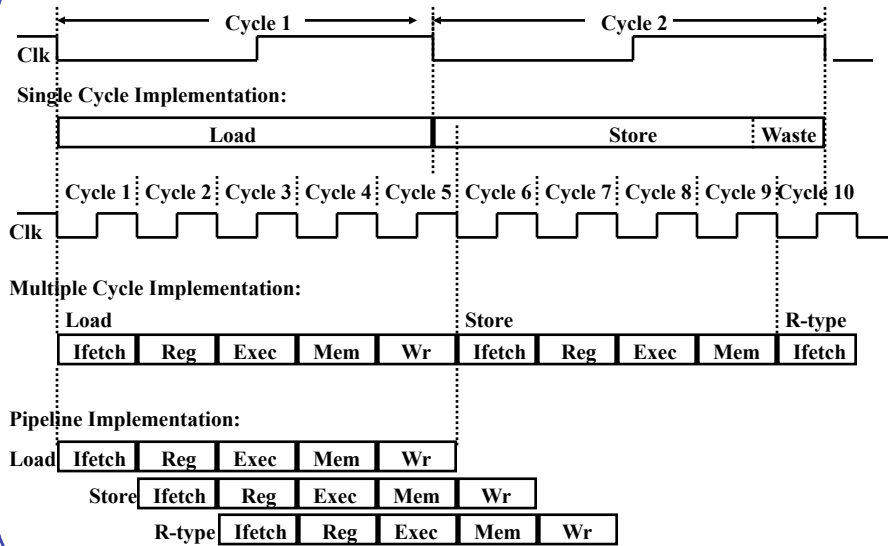
Pipelined Execution



- Now we just have to make it work

5

Single Cycle, Multiple Cycle, vs. Pipeline



6

Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
 - $45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = \text{___ ns}$
- Multicycle Machine
 - $10 \text{ ns/cycle} \times 4.0 \text{ CPI} \times 100 \text{ inst} = \text{___ ns}$
- Ideal pipelined machine
 - $10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = \text{___ ns}$

7

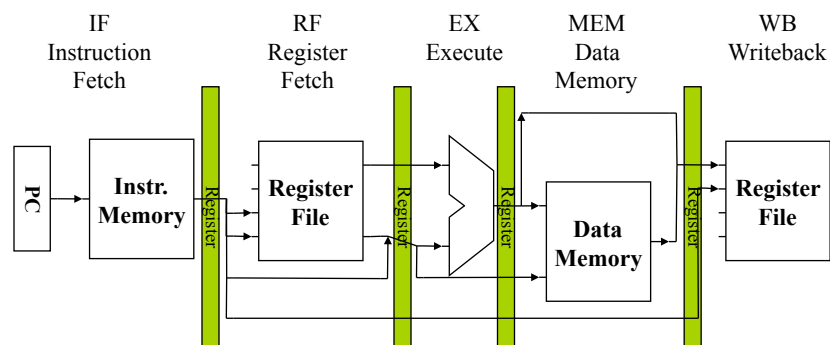
CPI for Pipelined Processors

- Ideal pipelined machine
 - $10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = \text{___ ns}$
- CPI in pipelined processor is “issue rate”. Ignore fill/drain, ignore latency.
- Example: A processor wastes 2 cycles after every branch, and 1 after every load, during which it cannot issue a new instruction. If a program has 10% branches and 30% loads, what is the CPI on this program?
-

8

Pipelined Datapath

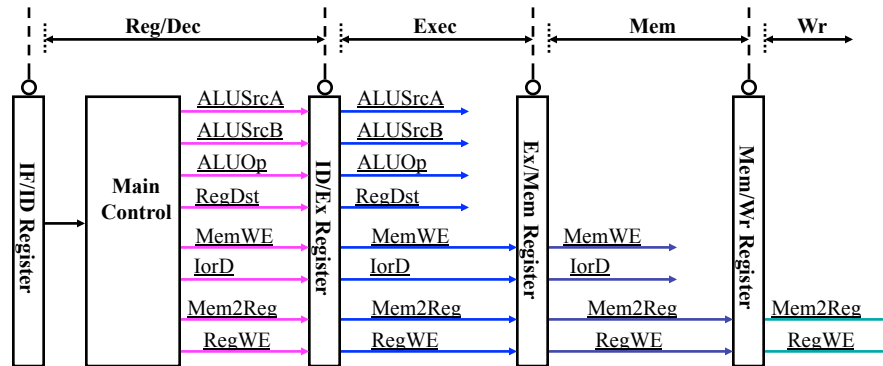
- Divide datapath into multiple pipeline stages



9

Pipelined Control

- The Main Control generates the control signals during Reg/Dec
 - Control signals for Exec (ALUSrcA, ALUSrcB, ...) are used 1 cycle later
 - Control signals for Mem (MemWE, lorD, ...) are used 2 cycles later
 - Control signals for Wr (Mem2Reg, RegWE, ...) are used 3 cycles later



10

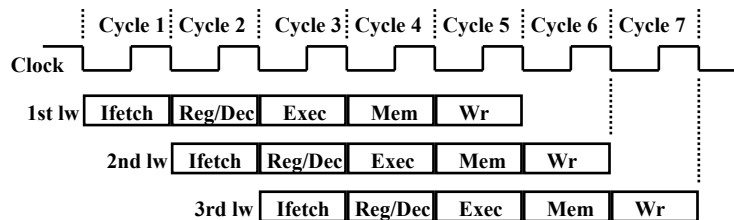
Can pipelining get us into trouble?

- Yes: **Pipeline Hazards**
 - structural hazards:** attempt to use the same resource two different ways at the same time
 - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
 - data hazards:** attempt to use item before it is ready
 - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
 - instruction depends on result of prior instruction still in the pipeline
 - control hazards:** attempt to make decision before condition evaluated
 - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
 - branch instructions
- Can always resolve hazards by **waiting**
 - pipeline control must detect the hazard
 - take action (or delay action) to resolve hazards

11

Pipelining the Load Instruction

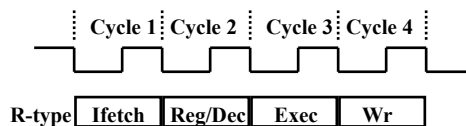
- The five independent functional units in the pipeline datapath are:
 - Instruction Memory for the **Ifetch** stage
 - Register File's Read ports (bus A and bus B) for the **Reg/Dec** stage
 - ALU for the **Exec** stage
 - Data Memory for the **Mem** stage
 - Register File's Write port (bus W) for the **Wr** stage



12

The Four Stages of R-type

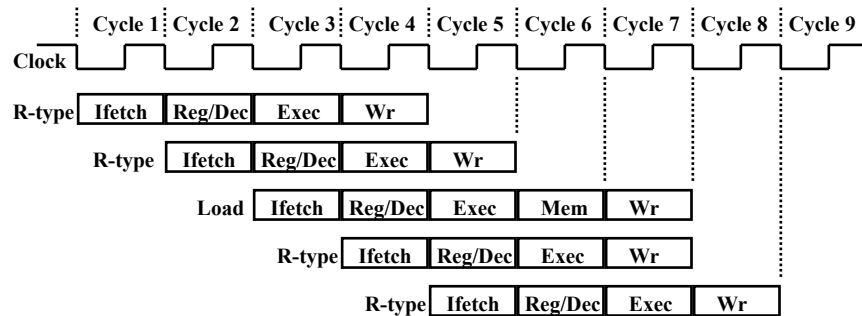
- Ifetch: Fetch the instruction from the Instruction Memory
- Reg/Dec: Register Fetch and Instruction Decode
- Exec: ALU operates on the two register operands
- Wr: Write the ALU output back to the register file



13

Structural Hazard

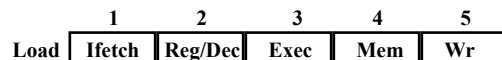
- Interaction between R-type and loads causes structural hazard on writeback



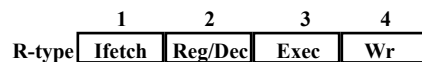
14

Important Observation

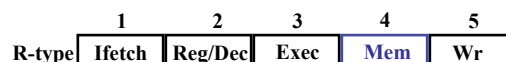
- Each functional unit can only be used **once** per instruction
- Each functional unit must be used at the **same** stage for all instructions:
 - Load uses Register File's Write Port during its **5th** stage



- R-type uses Register File's Write Port during its **4th** stage

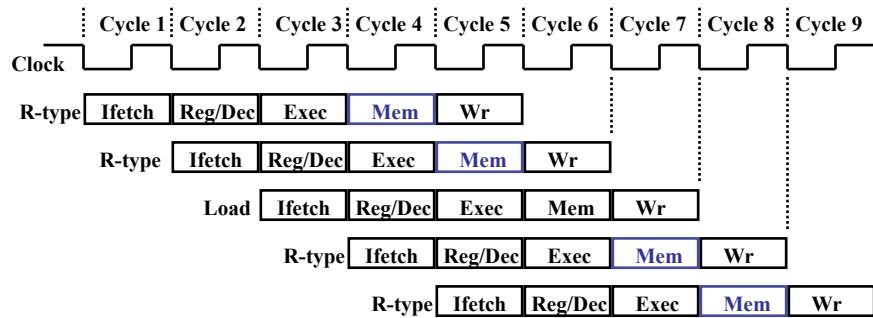


- Solution: Delay R-type's register write by one cycle:
 - Now R-type instructions also use Reg File's write port at Stage 5
 - Mem stage is a **NOOP** stage: nothing is being done.



15

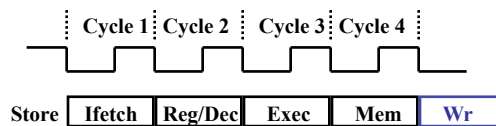
Pipelining the R-type Instruction



16

The Four Stages of Store

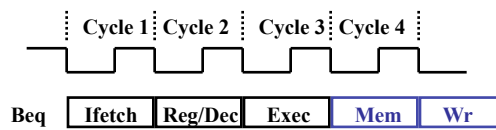
- Ifetch: Fetch the instruction from the Instruction Memory
- Reg/Dec: Register Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory
- Wr: **NOOP**
- Compatible with Load & R-type instructions



17

The Stages of Branch

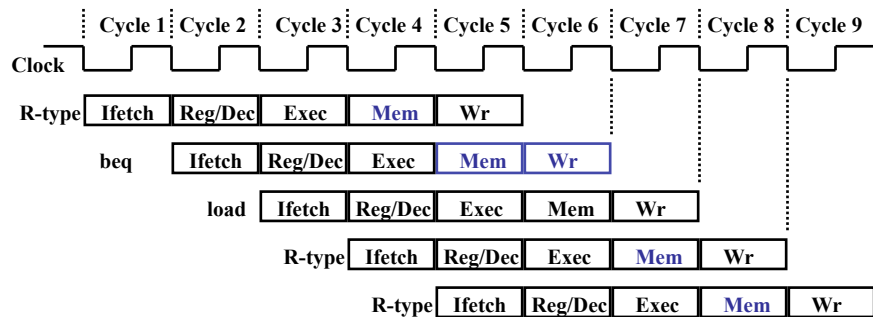
- Ifetch: Fetch the instruction from the Instruction Memory
- Reg/Dec: Register Fetch and Instruction Decode, compute branch target
- Exec: Test condition & update the PC
- Mem: **NOOP**
- Wr: **NOOP**



18

Control Hazard

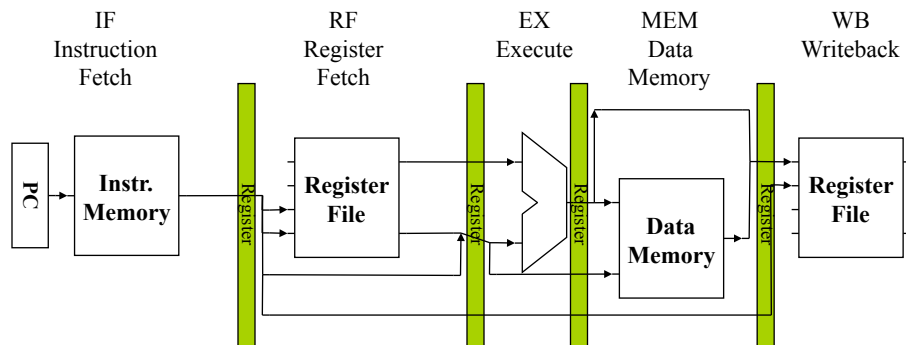
- Branch updates the PC at the end of the Exec stage.



19

Accelerate Branches

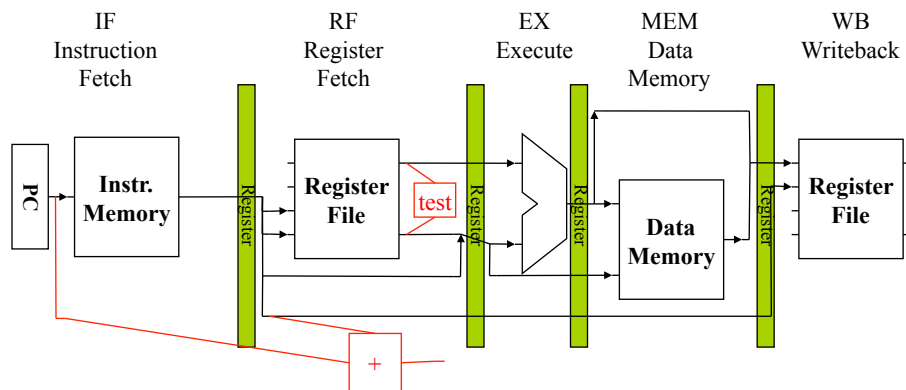
- When can we compute branch target address?
- When can we compute beq condition?



20

Accelerate Branches

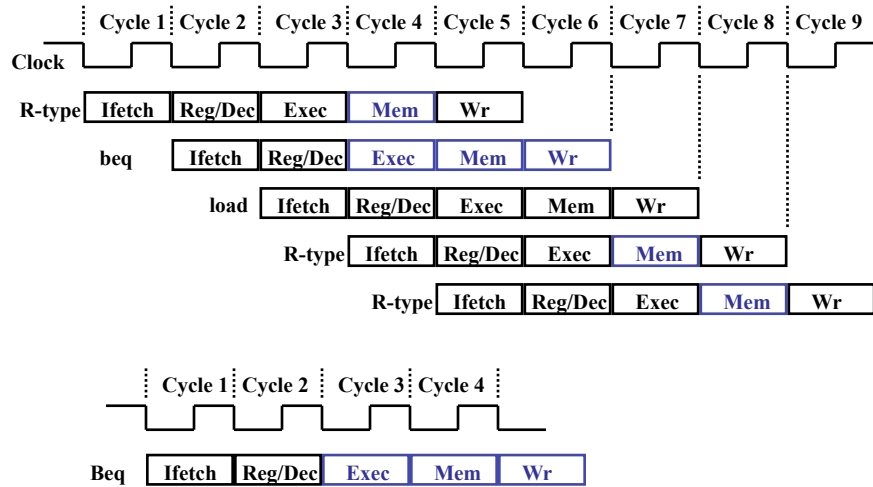
- When can we compute branch target address?
- When can we compute beq condition?



21

Control Hazard 2

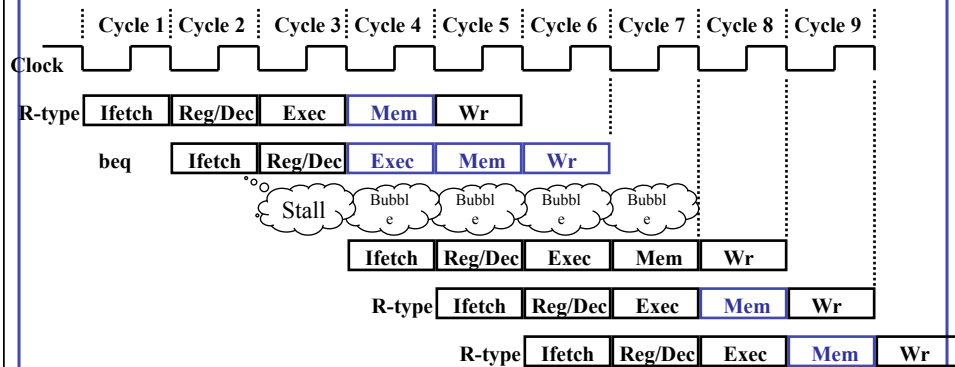
- Branch updates the PC at the end of the Reg/Dec stage.



22

Solution #1: Stall

- Delay loading next instruction, load no-op instead

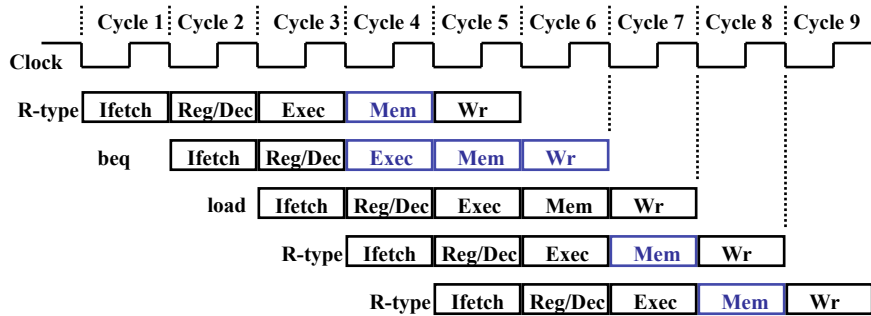


- CPI if all other instructions take 1 cycle, and branches are 20% of instructions?

23

Solution #2: Branch Prediction

- Guess all branches not taken, squash if wrong



- CPI if 50% of branches actually not taken, and branch frequency 20%?

24

Solution #3: Branch Delay Slot

- Redefine branches: Instruction directly after branch always executed
Instruction after branch is the **delay slot**

Compiler/assembler **fills** the delay slot

```
add $t1, $t0, $t0    sub $t2, $t0, $t3    add $t1, $t0, $t0    add $t1, $t0, $t0
beq $t2, $t3, FOO    add $t1, $t0, $t0    beq $t1, $t3, FOO    beq $t1, $t3, FOO
                    beq $t1, $t3, FOO
                    add $t1, $t3, $t3
                    ...
FOO:
    add $t1, $t2, $t0
```

25

Solution #3: Branch Delay Slot

- Redefine branches: Instruction directly after branch always executed
- Instruction after branch is the **delay slot**
- Compiler/assembler **fills** the delay slot

add \$t1, \$t0, \$t0	sub \$t2, \$t0, \$t3	add \$t1, \$t0, \$t0	add \$t1, \$t0, \$t0
beq \$t2, \$t3, FOO	add \$t1, \$t0, \$t0	beq \$t1, \$t3, FOO	beq \$t1, \$t3, FOO
add \$t1, \$t0, \$t0	beq \$t1, \$t3, FOO	add \$t1, \$t2, \$t0	add \$0, \$0, \$0
	sub \$t2, \$t0, \$t3	add \$t1, \$t3, \$t3	

No
wasted
cycles

No
wasted
cycles

...
FOO:
~~add \$t1, \$t2, \$t0~~
Assume 50% branch,
Wastes ½ cycle per branch

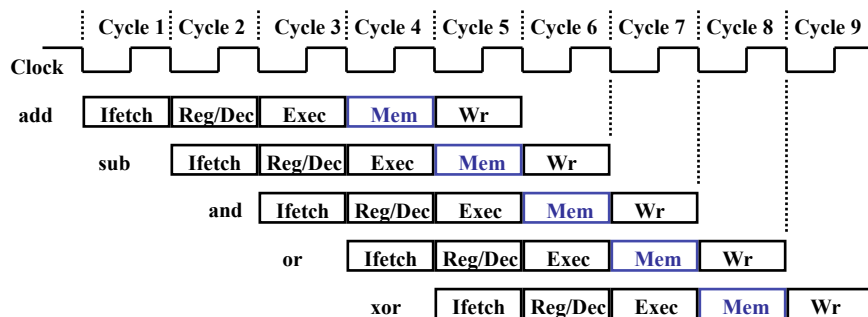
Insert noop
Wastes 1 cycle
per branch

Compare vs. stall

26

Data Hazards

- Consider the following code:
 - add \$t0, \$t1, \$t2
 - sub \$t3, \$t0, \$t4
 - and \$t5, \$t0, \$t7
 - or \$t8, \$t0, \$s0
 - xor \$s1, \$t0, \$s2

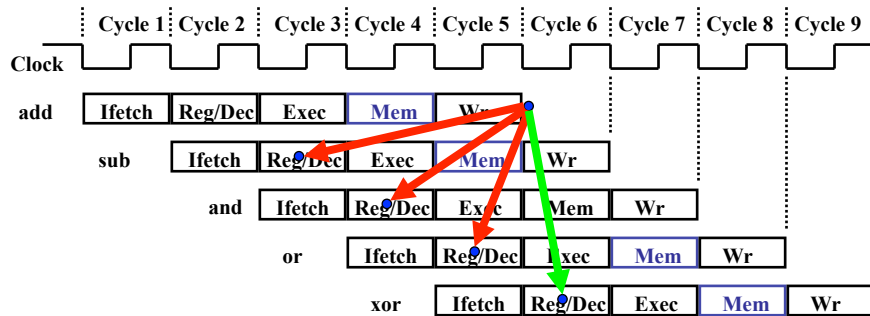


27

Data Hazards

- Consider the following code:

- add \$t0, \$t1, \$t2
- sub \$t3, \$t0, \$t4
- and \$t5, \$t0, \$t7
- or \$t8, \$t0, \$s0
- xor \$s1, \$t0, \$s2

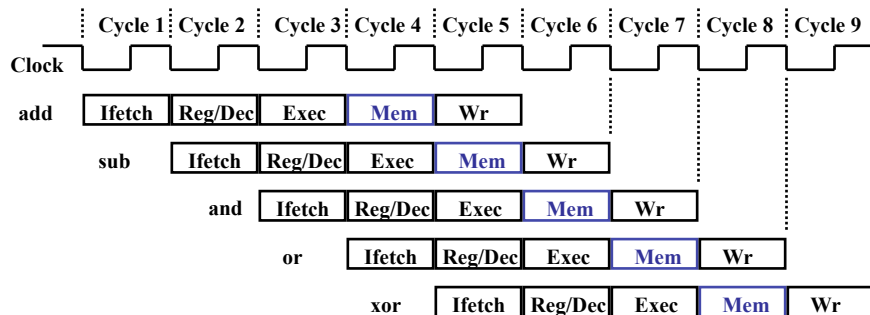


28

Design Register File Carefully

What if reads see value after write during the same cycle?

add \$t0, \$t1, \$t2
 sub \$t3, \$t0, \$t4
 and \$t5, \$t0, \$t7
 or \$t8, \$t0, \$s0
 xor \$s1, \$t0, \$s2



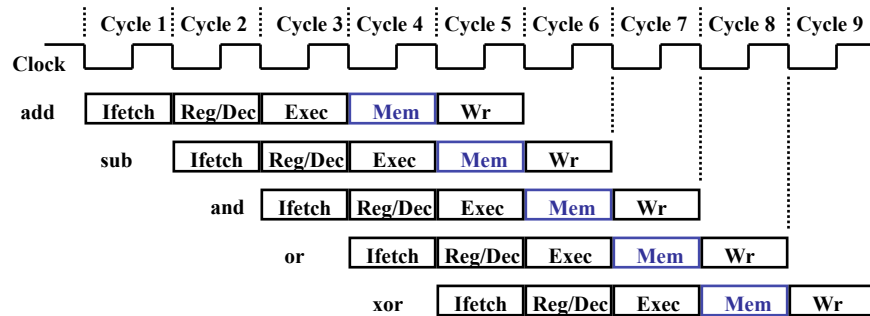
29

Forwarding

Note: data is computed by end of Cycle 3 (Exec stage of add). Add logic to pass last two values from ALU output to ALU input(s) as needed

- Forward the ALU output to later instructions

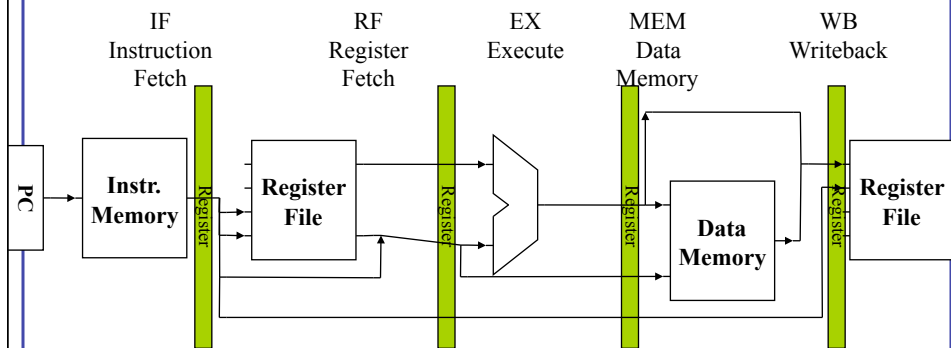
```
add $t0, $t1, $t2
sub $t3, $t0, $t4
and $t5, $t0, $t7
or $t8, $t0, $s0
xor $s1, $t0, $s2
```



30

Forwarding (cont.)

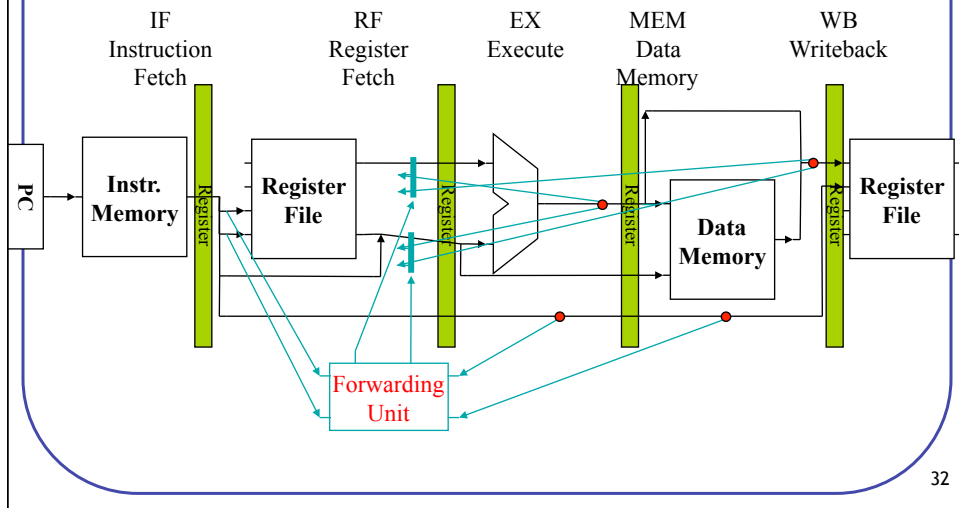
- Requires values from last two ALU operations.
- Remember destination register for operation.
- Compare sources of current instruction to destinations of previous 2.



31

Forwarding (cont.)

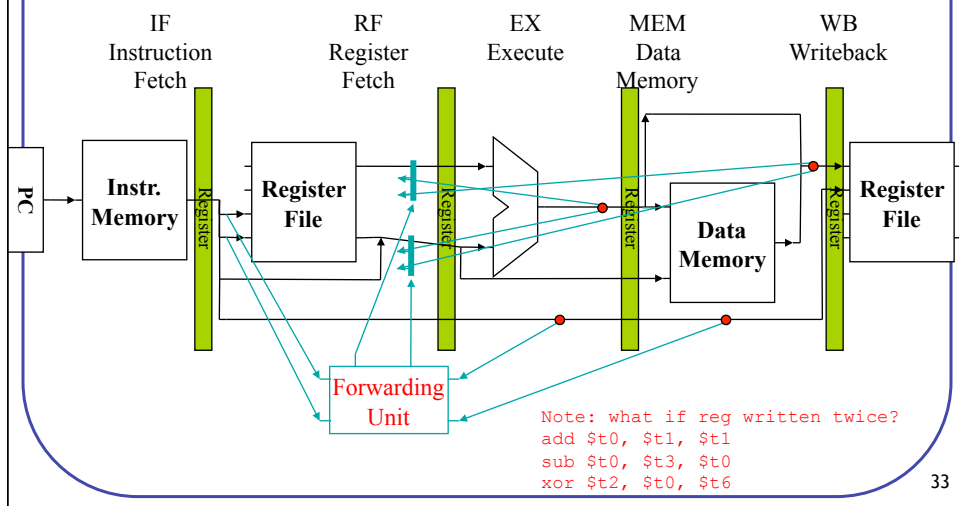
- Requires values from last two ALU operations.
- Remember destination register for operation.
- Compare sources of current instruction to destinations of previous 2.



32

Forwarding (cont.)

- Requires values from last two ALU operations.
- Remember destination register for operation.
- Compare sources of current instruction to destinations of previous 2.



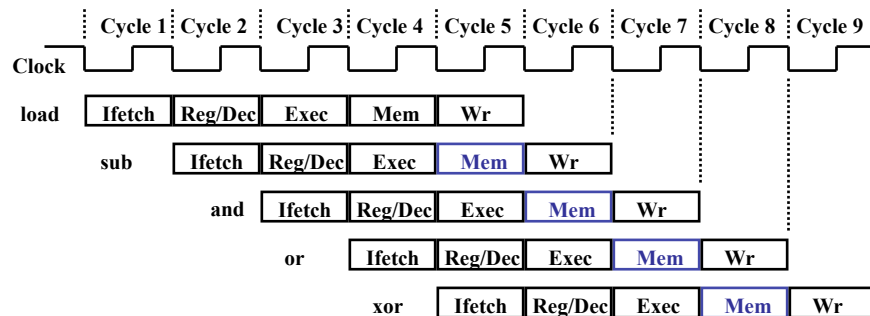
Note: what if reg written twice?
 add \$t0, \$t1, \$t1
 sub \$t0, \$t3, \$t0
 xor \$t2, \$t0, \$t6

33

Data Hazards on Loads

```

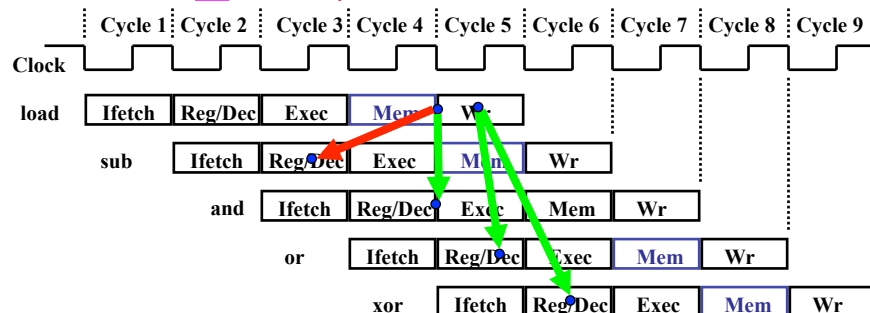
load $t0, 0($t1)      # Data being fetched in cycle 4
sub  $t3, $t0, $t4     # sub performed in cycle 4!
and  $t5, $t0, $t7     # fixed already ☺
or   $t8, $t0, $s0
xor  $s1, $t0, $s2
  
```



34

Data Hazards on Loads

- Consider the following code:
 - load \$t0, 0(\$t1)
 - sub \$t3, \$t0, \$t4 - cannot be solved. Data unavailable when needed
 - and \$t5, \$t0, \$t7 - solved by forwarding
 - or \$t8, \$t0, \$s0 - fixed by register file design (read sees same cycle write)
 - xor \$s1, \$t0, \$s2 - no problem



35

Data Hazards on Loads

- Solution:
 - Use same forwarding hardware & register file for hazards 2+ cycles later
 - Stall for a cycle (no one likes this): hazard detection logic will
 - Force compiler to not allow register reads within a cycle of load
 - Fill delay slot, or insert no-op.

36

Pipelined CPI, cycle time

- CPI, assuming compiler can fill 50% of delay slots

Instruction Type	Type Cycles	Type Frequency	Cycles * Freq
ALU		50%	
Load		20%	
Store		10%	
Branch		20%	
CPI:			

Pipelined: cycle time = 1ns.

Delay for 1M instr:

Multicycle: CPI = 4.0, cycle time = 1ns.

Delay for 1M instr:

Single cycle: CPI = 1.0, cycle time = 4.5ns. Delay for 1M instr:

37

Pipelined CPI, cycle time

- CPI, assuming compiler can fill 50% of delay slots

Instruction Type	Type Cycles	Type Frequency	Cycles * Freq
ALU	1.0	50%	0.5
Load	1.5	20%	0.3
Store	1.0	10%	0.1
Branch	1.5	20%	0.3
CPI:			1.2

- Pipelined: cycle time = 1ns. Delay for 1M instr: $(1.2 \times 10^6 + 4) \text{ns}$
- Multicycle: CPI = 4.0, cycle time = 1ns. Delay for 1M instr: $4 \times 10^6 \text{ns}$
- Single cycle: CPI = 1.0, cycle time = 4.5ns. Delay for 1M instr: $4.5 \times 10^6 \text{ns}$

38

Pipelined CPU Summary

39

Pipelined CPU Summary

- Improve cycle time by pipelining CPU
- Concerns
 - Structural Hazards - two instrs can't use same resource in a clock cycle
 - All instrs are 5 cycles, and do the same tasks on each of their cycles
 - Control Hazards - instructions after jump/branches may get executed
 - Speed branch computation to reduce control hazards
 - Compiler fills delay slot, or use branch prediction
 - Data Hazards - next instruction may need value you compute
 - Forwarding to pass value from ALU out to ALU in of next instr(s)
 - Loads have unavoidable hazards, can't be accessed in next instr.
- Significant potential speedups over single-cycle, multi-cycle CPU
 -