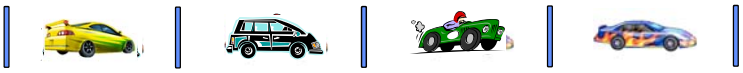
Pipelining: Its Natural!

- Car wash Example

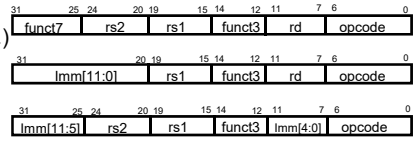


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and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

(1)

Computer Pipelines

- RISC features that facilitates efficient pipelining:
  - all instructions have the same length,
  - registers located in the same place in instruction format,
  - memory operands only in loads or stores
- We will first review a non-pipelined RISC architecture that executes:
  1) Integer R-type instructions (ex. add, sub, .)
  2) Load instructions (lw)
  3) Store instructions (sw)
  4) Conditional branch

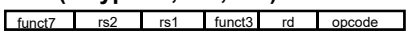


- Review – you should know about:
  - multiplexors,
  - register files,
  - ALU's
  - arithmetic Vs logical shifts
  - program counter (PC), status word (PS) and instruction register (IR),
  - Memory address registers (MAR) and memory data registers (MDR)
  - combinational Vs sequential circuits

(2)

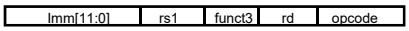
Implementing the RISC architecture (Section C.3)

Arithmetic/logic instructions (R-type rd, rs1, rs2)



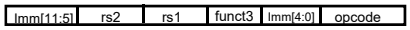
1) fetch instruction
2) read registers rs1 and rs2
3) compute result (use ALU)
4) write to register rd

Load instructions (lw rd, c(rs1))



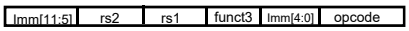
1) fetch instruction
2) read register rs1 (and rs2 ??)
3) use ALU to compute memory address = content of rs1 + c
4) read from memory
5) write to register rd

Store instructions (sw rs2, c(rs1))



1) fetch instruction
2) read registers rs1 and rs2
3) use ALU to compute memory address = content of rs1 + c
4) write to memory at address stored in rs2

Branch instructions (beq rs1, rs2, L)



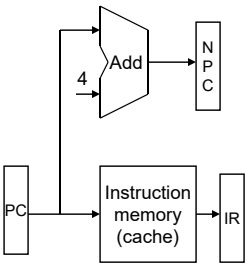
1) fetch instruction
2) read registers rs1 and rs2
3) use ALU to compute branch address = PC + offset and evaluate branch condition (rs1 == rs2)
4) update the PC (if condition satisfied)

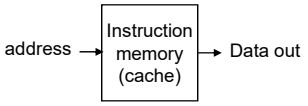
Increment the PC by 4 (in step 1)

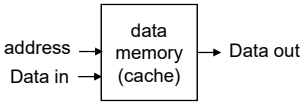
(3)

Instruction fetch

IR ← Mem[PC]
NPC ← PC + 4 /*next PC*/

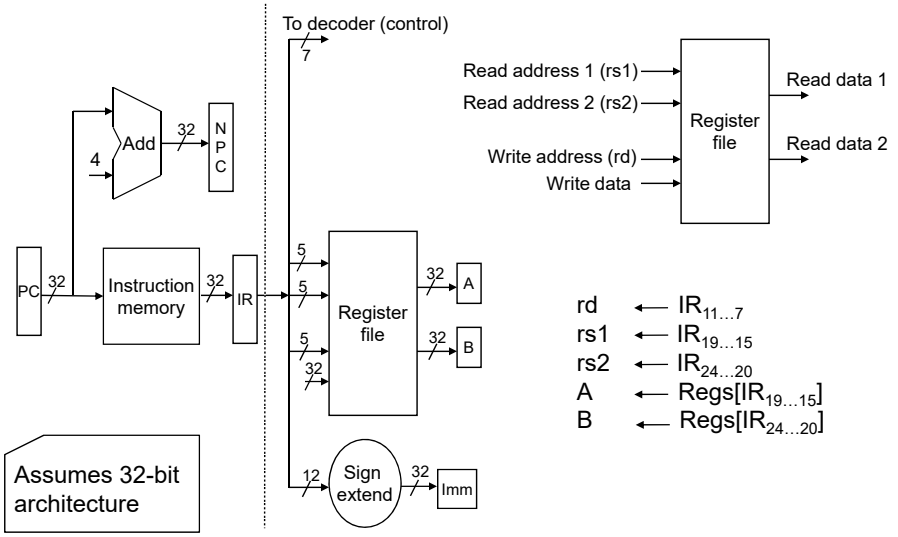






(4)

Register read

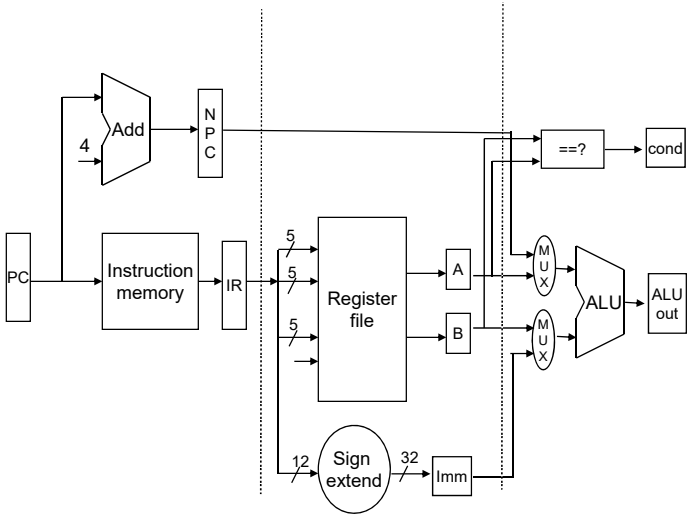


Imm ← $(IR_{31})^{20}$ ## $IR_{31..20}$ (for lw)

Imm ← $(IR_{31})^{20}$ ## $IR_{31..20}$ ## $IR_{11...7}$ (for sw and branch)

(5)

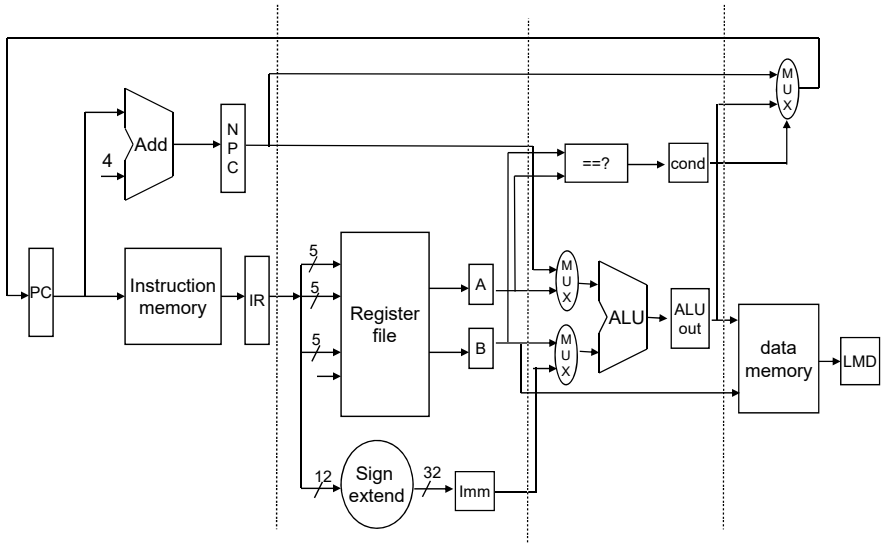
Use ALU and evaluate branch condition



(the multiplexors settings and ALU function depend on the op-code)

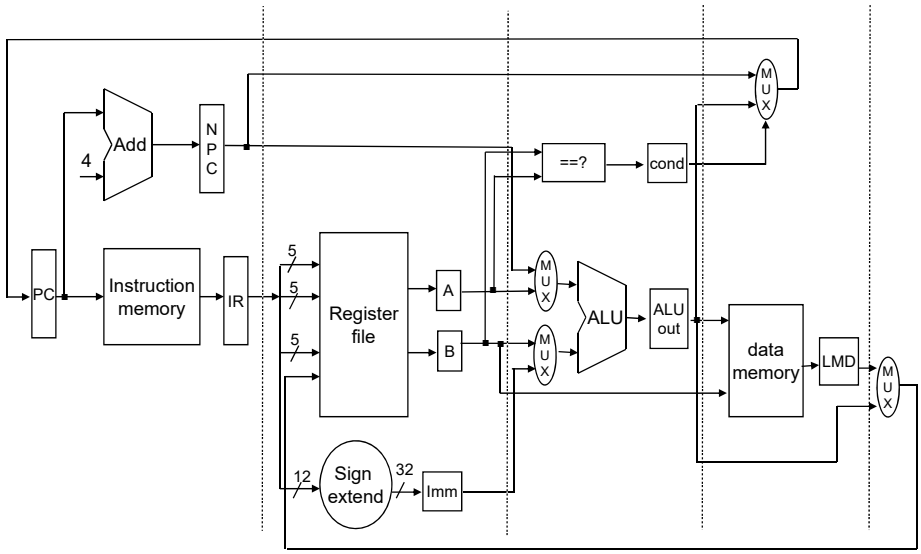
(6)

Use memory and update PC



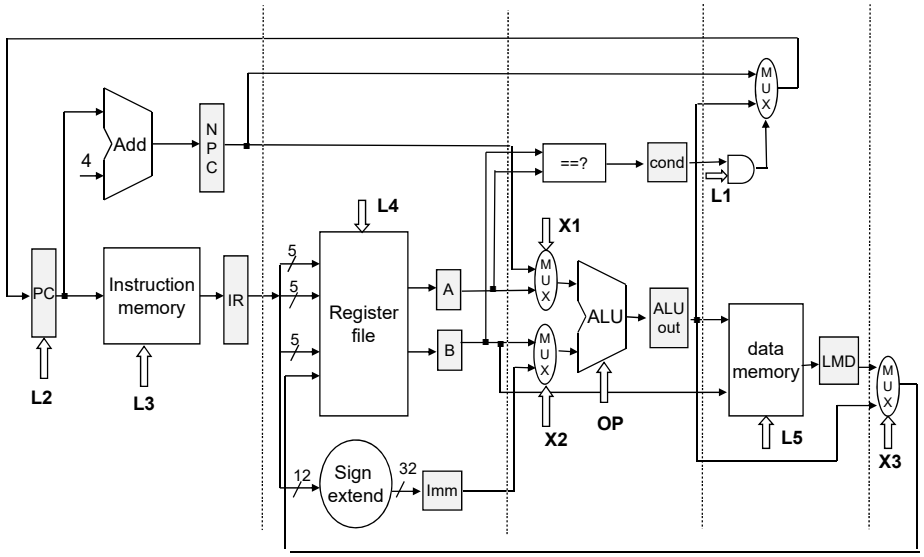
(7)

Write to register



(8)

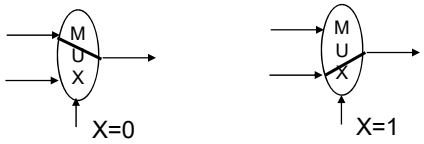
The control signals



(9)

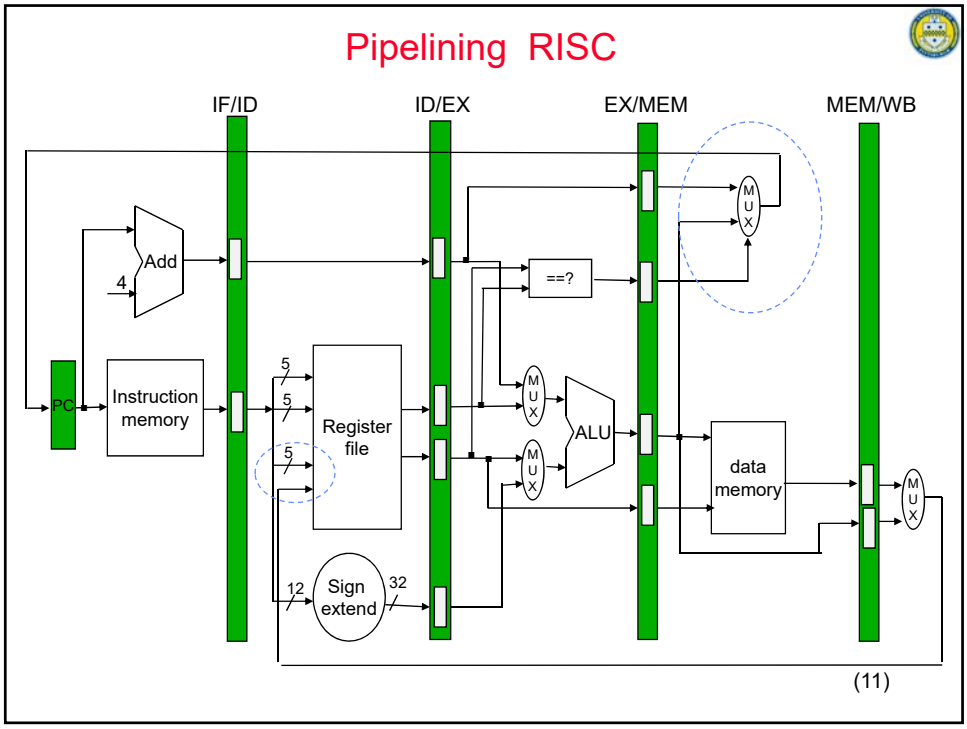
The control signals

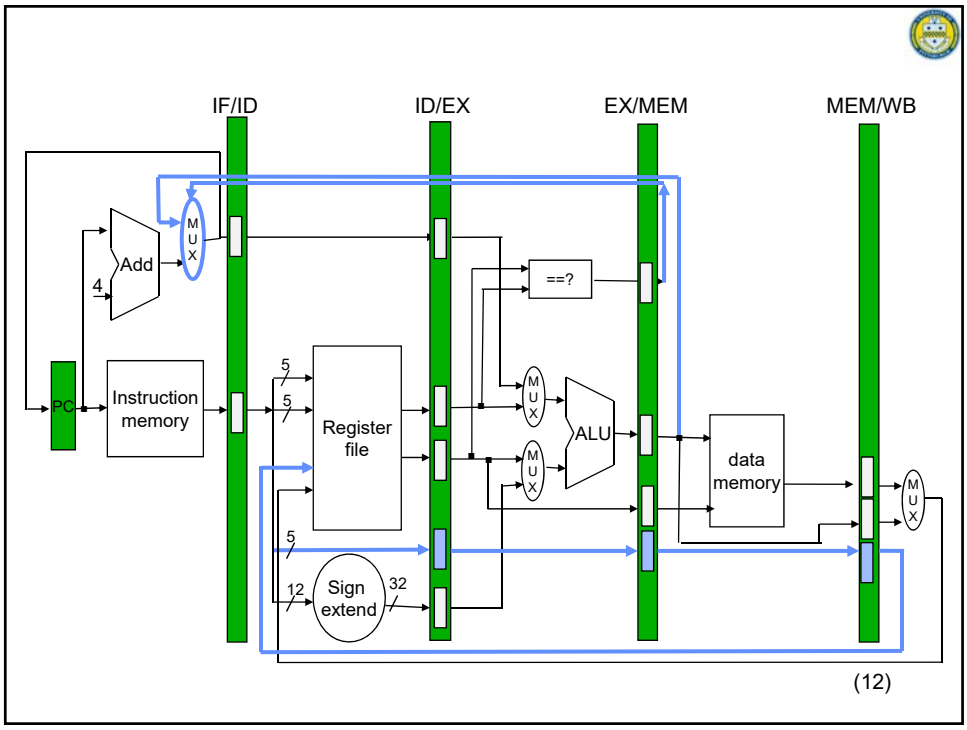
- X1, X2, X3 = select multiplexor input (one bit each)



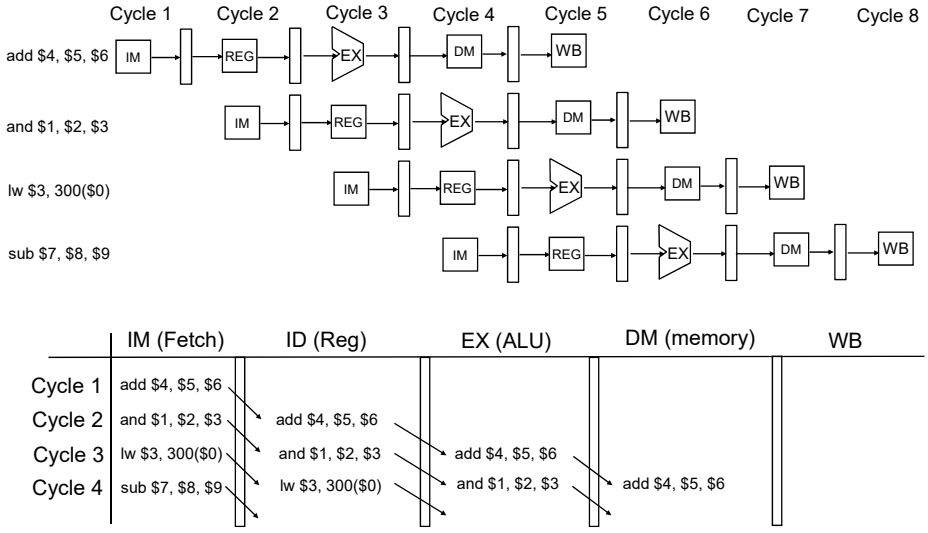
- L1 = set if the instruction is a branch (one bit)
- L2 = loads the PC (one bit)
- L3 = read the instruction memory (one bit)
- L4 = write register (one bits)
  0 = no write, 1 = write
- L5 = read/write the data memory (two bits)
  00 = no-op, 01 = read, 10 = write
- OP = ALU control (?? Bits)
  0..0 = no-op, 1..1 = add, 10.. = others

(10)


Pipelining RISC
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Instruction memory
5
5
5
Register file
==?
MUX
ALU
data memory
12
Sign extend
32
(11)


IF/ID
ID/EX
EX/MEM
MEM/WB
MUX
Add
4
PC
Instruction memory
5
5
5
Register file
==?
ALU
data memory
12
Sign extend
32
(12)

Visualizing Pipelining (two equivalent diagrams)



Pipelining puts additional demands on memory bandwidth,

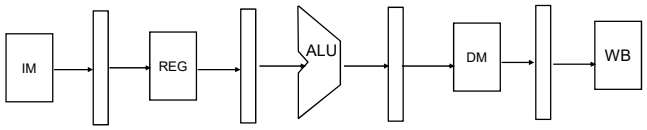
(13)

Limits to pipelining (Sections C.1 and C.2)

- Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support some combination of instructions.
  - Data hazards: An instruction depends on result of prior instruction still in the pipeline.
  - Control hazards: Pipelining of branches & other instructions *stall* the pipeline until the hazard *bubbles* in the pipeline

(14)

Structural Hazards

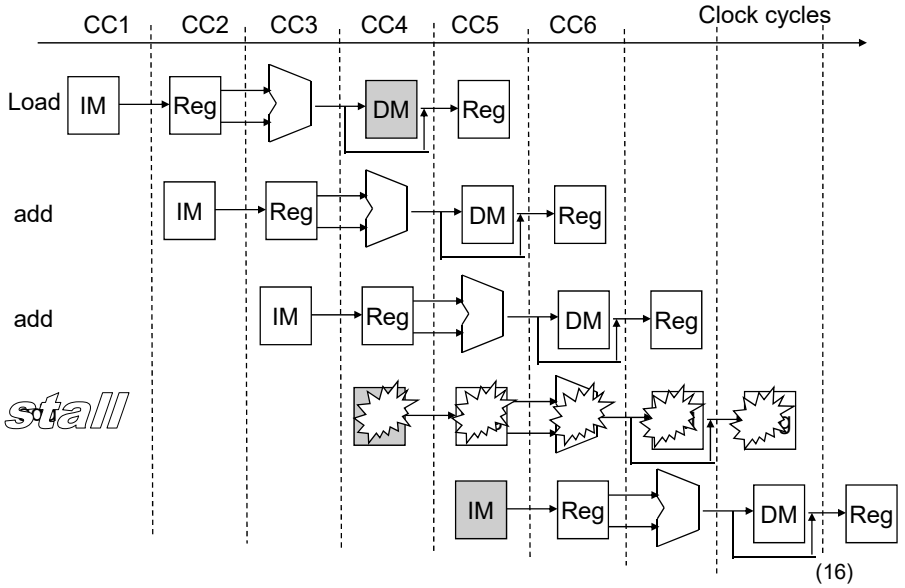


Potential problem: Both REG and WB use the register file
Solution : Read from register file during the first half of a cycle and write to register file during the second half of a cycle.

Potential problem: Both IM and DM use memory
Solution : use separate memories (caches)

(15)

Structural Hazards (assuming a single memory)



(16)

Speed Up Equation for Pipelining

$$CPI_{pipelined} = Ideal\ CPI + stall\ cycles\ per\ instruction$$

$$Speedup = \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

Example:

- Machine A: pipelined (with some depth) and dual ported memory
- Machine B: pipelined (same depth as A), but single ported memory, and a 1.05 times faster clock rate
- Ideal CPI = 1 for both, and lw/sw are 40% of instructions executed

$SpeedUp_A$ = Pipeline Depth

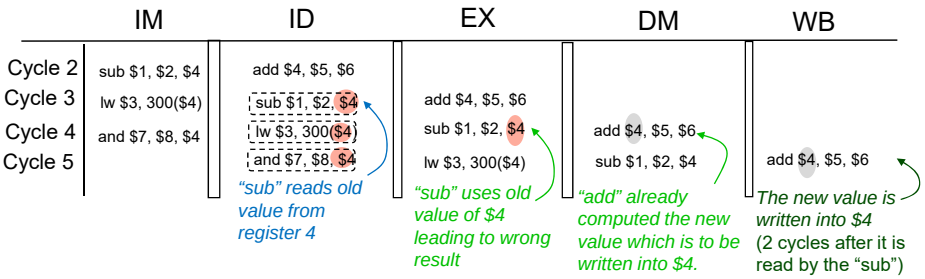
$SpeedUp_B$ = (Pipeline Depth/1.4) x 1.05

= 0.75 x Pipeline Depth

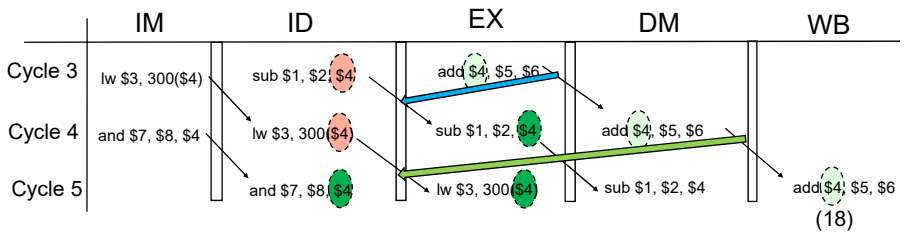
$SpeedUp_A$ / $SpeedUp_B$ = 1.33

(17)

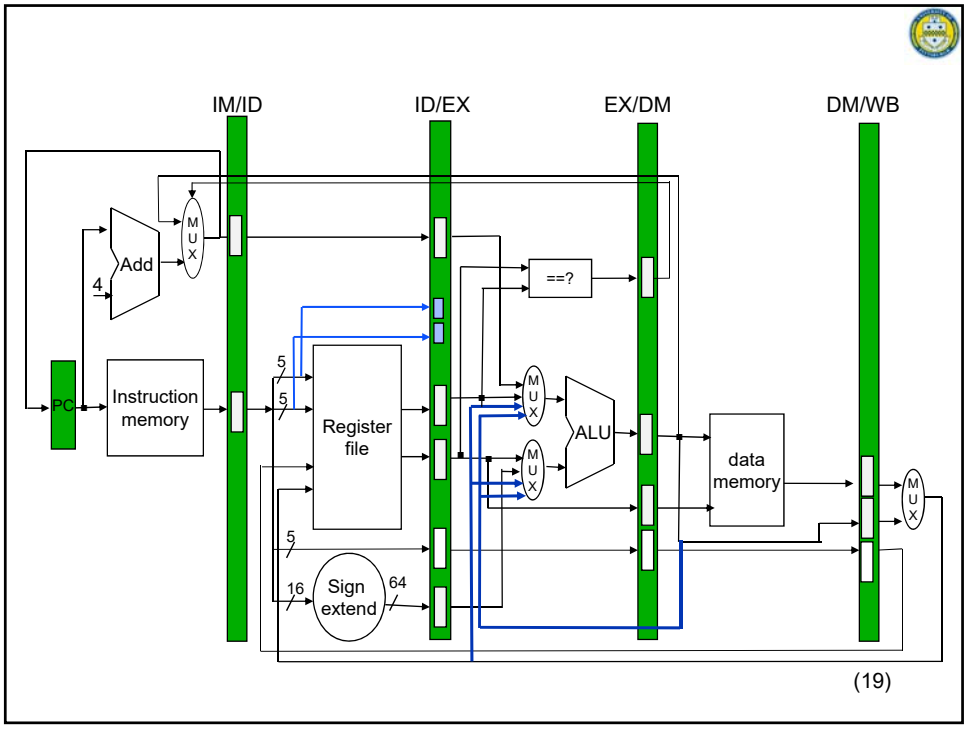
Register-to-register data Hazards



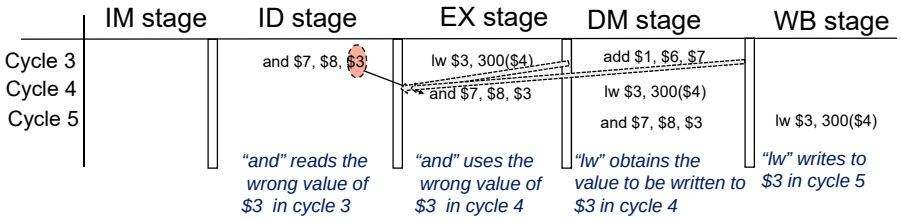
Use forwarding to mitigate data hazards?



(18)

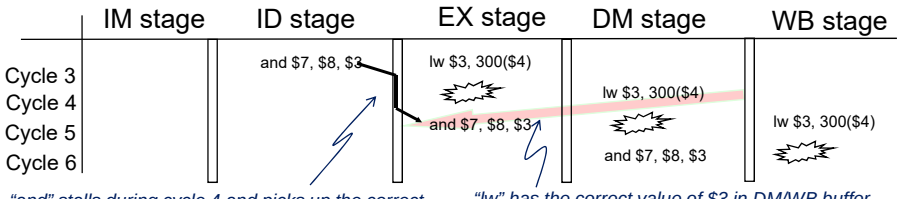


Forwarding may not be enough



Problem: *can't use forwarding at the beginning of cycle 4 since "lw" produces the data to be written in $3 at the end of cycle 4*

Solution: *need to combine forwarding with stalling the pipe.*



(20)

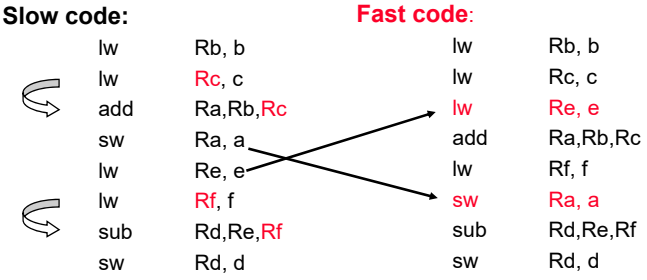
Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

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

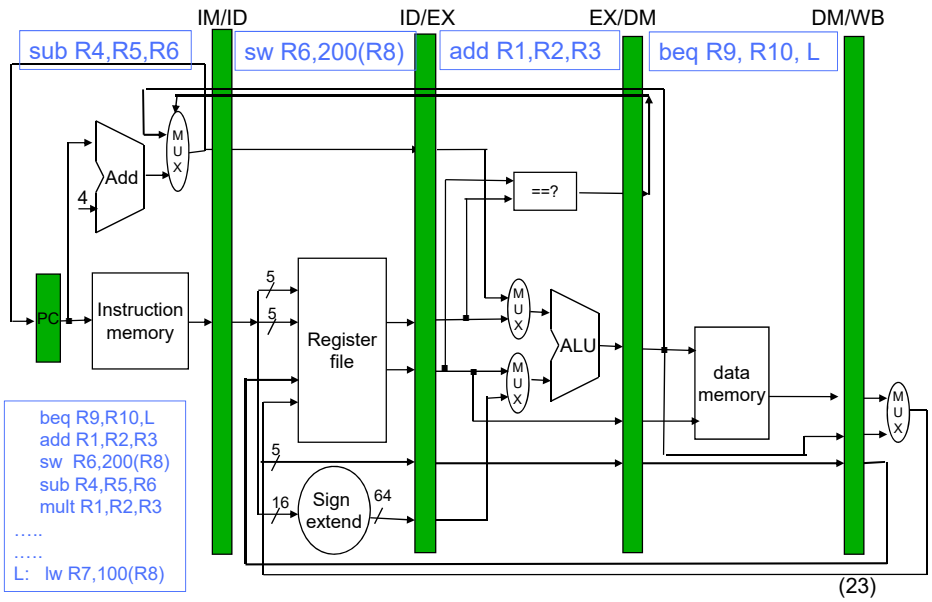


Three types hazards caused by Data Dependence

- **Read After Write (RAW)**
  $Instr_{i+k}$ reads operand before $Instr_i$ writes it
- **Write After Read (WAR)**
  $Instr_{i+k}$ writes operand *before* $Instr_i$ reads it
  - Gets wrong operand
  - Can't happen in RISC 5 stage pipeline (why?)
- **Write After Write (WAW)**
  $Instr_{i+k}$ writes operand *before* $Instr_i$ writes it
  - Leaves wrong result
  - Can't happen in RISC 5 stage pipeline (why?)
- Will see WAR and WAW in later more complicated pipes

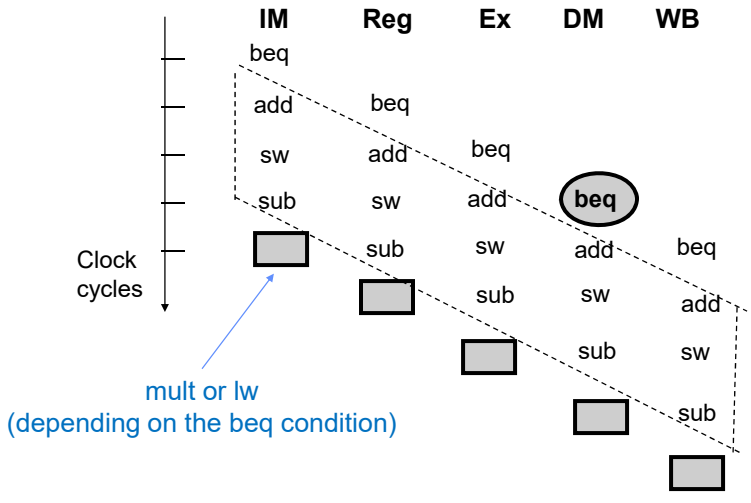
$Instr_i$ ..... $Instr_{i+k}$

Control Hazard on Branches



(23)

Control Hazard on Branches



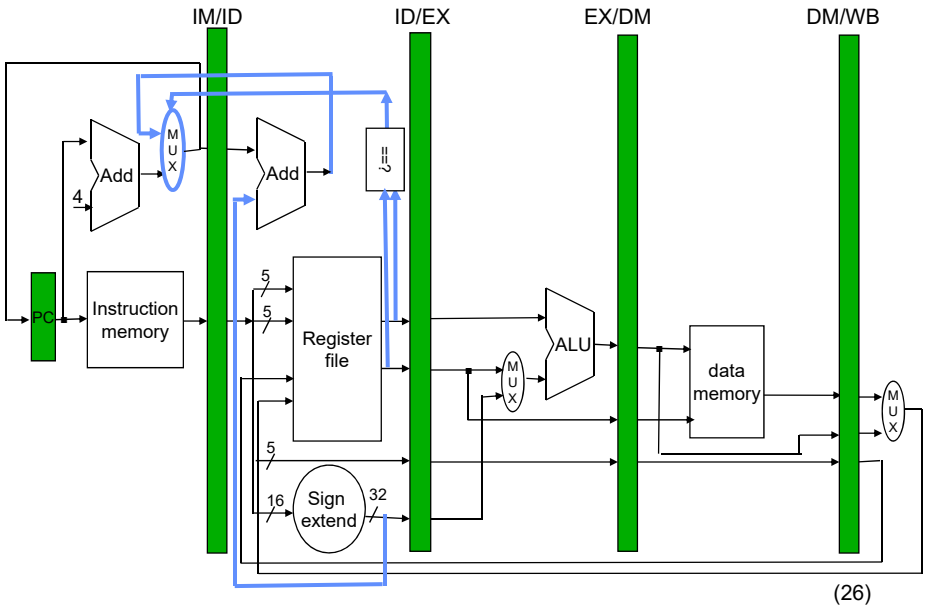
(24)

Branch Stall Impact

- If CPI = 1, 10% branch, Stall 3 cycles => new CPI = 1.3
- Two part solution:
  - Determine branch taken (or not) sooner, and
  - Compute taken branch address earlier
- RISC V branch tests if register = 0
- RISC V Solution:
  - Move Zero test to ID/EX stage
  - Adder to calculate new PC in ID/EX stage
  - 1 clock cycle penalty for branch versus 3

(25)

Early determination of Branch condition and target



(26)

Four Branch Hazard Alternatives

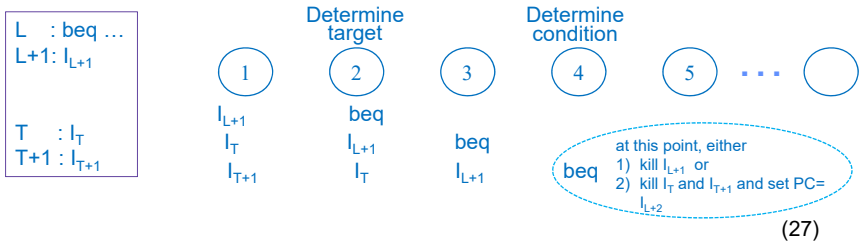
#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence (PC+4 already calculated)
- "Squash" instructions in pipeline if branch actually taken (how?)
- Advantage of late pipeline state update

#3: Predict Branch Taken

- More than 50% of MIPS branches are taken on average
- Start fetching from the branch target as soon as it is available
- Useful if target address is computed earlier than the branch condition (for example in stage 2 and stage 4, respectively, of a deep pipeline).



Four Branch Hazard Alternatives

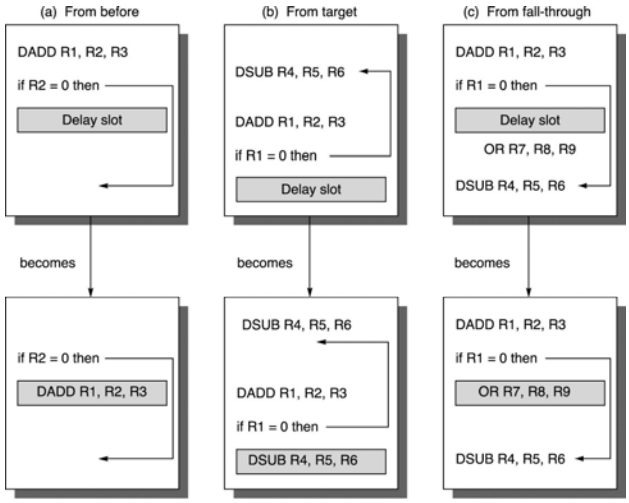
#4: Delayed Branch

- Change semantics such that branching takes place AFTER the *n* instructions following the branch execute

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

Branch delay of length *n*

- One slot delay in the 5-stage pipeline if branch condition and target are resolved in the ID stage.



- Where do we get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: should repeat instruction for correctness
  - From fall through: correct if R7 is dead after the branch

Evaluating Branch Alternatives

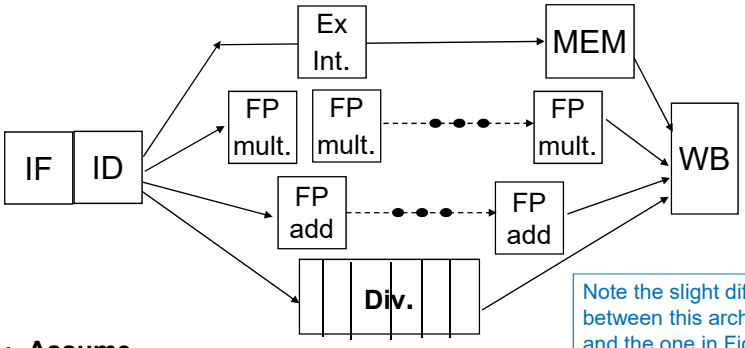
When accounting for branch hazards
CPI = 1 + Hazard frequency * Hazard penalty

- **Example**: Assume a pipeline in which the target address is known in stage 3 and the branch condition is known in stage 4. Compare the CPI for the following methods:
  - flush/stall
  - predict taken
  - predict not taken
  - delayed branch

  Assume that you know the percentages of branch instructions, and the probabilities for branch taken/not taken.
- Compilers technology can be used to increase efficiency of code if it knows branch probabilities (static branch prediction - profiling)

(30)

Multi-cycle pipelines (Section C.5)



Note the slight difference between this architecture and the one in Figure C.30 of the textbook

- **Assume**
  - 4-stage, pipelined, FP add
  - 7-stage, pipelined, FP multiply
  - 25 stage, non-pipelined divide unit
- **Latency** of an instruction, *I*, in a pipeline, *P*, is the number of bubbles that has to exist in *P* if the instruction following *I* wants to use the result of *I*.

(31)

Hazards:

- Two divide instructions will stall the pipe (structural hazards).
- May have more than one register write in one cycle (why?)
  - increase number of ports, or stall the pipeline (interlock)
- May have WAW hazard (why?)
- Out-of-order completion causes problems with exceptions,
- The long pipes causes more RAW hazards (why?)

To deal with structural Hazards and resolve competition for the WB stage:

- Stall a conflicting instruction at the ID stage
  - Use a shift register to keep track of the utilization of a stage that may suffer from structural hazard (ex. Input ports of registers)
- Stall a conflicting instruction when entering the WB stage
  - may give priority to longer instructions to reduce RAW hazards.

(32)

Examples

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
fld F4, 0(R2)	IF	ID	EX	MEM	WB												
fmul.d F0, F4, F6		IF	ID	ID	M1	M2	M3	M4	M5	M6	M7	WB					
fadd.d F2, F0, F8			IF	IF	ID	ID	ID	ID	ID	ID	ID	A1	A2	A3	A4	WB	
fsd F2, 0(R2)					IF	IF	IF	IF	IF	IF	IF	ID	EX	EX	EX	MEM	

Stall due to RAW hazards (assuming forwarding)

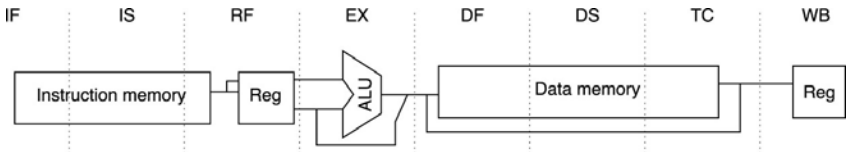
instruction	1	2	3	4	5	6	7	8	9	10	11
fmul.d F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	WB	
…		IF	ID	EX	Mem	WB					
…			IF	ID	EX	Mem	WB				
fadd.d F2,F4,F6				IF	ID	A1	A2	A3	A4	WB	
…					IF	ID	EX	Mem	WB		
fld F2, 0(R2)						IF	ID	EX	Mem	WB	
							IF	ID	Ex	Mem	WB

Structural hazards

(33)

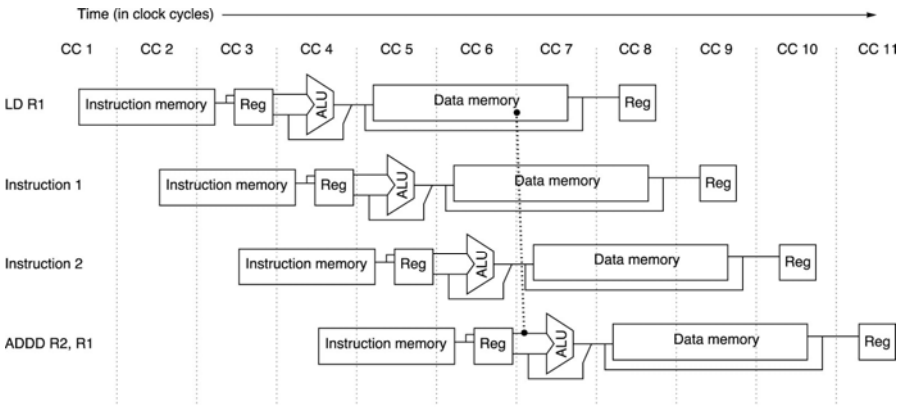
The MIPS R4000 (Section C.6)

- 64-bit instruction set (MIPS-3 ISA)
- Decomposes memory access into stages (super-pipelining)



- Uses a deeper pipeline
  - IF -- first half of instruction fetch
  - IS -- second half of instruction fetch
  - RF - instruction decode and register fetch (and check cache tag)
  - EX - execution: effective address calculation, ALU operation, branch target computation, branch condition evaluation
  - DF - first half of data fetch
  - DS - second half of data fetch
  - TC - check cache tag (to determine if it is a hit)
  - WB write back

(34)



- Forwarding from data memory output is required to instructions that are 3 or 4 cycles later (sooner instructions have to stall)
- Can speculatively forward from data memory (LD R1) to ALU (ADD R2, R1) before checking the tag to make sure that it is a hit.

(35)

To maintain precise exceptions (Section C.4) :

- May ignore the problem, or
- buffer the results and enforce the order of writes, or
- let the trap handling routine enforce the preciseness (software approach), or
- delay the issue (stall the pipe) to enforce in-order completion.

To deal with WAW:

- x = 1
  if (a==b) x = 2

- May detect hazard and hold the second instruction,

Can all hazards be detected at the decode stage?

(36)

Instruction set design and pipelining (C.4)

- Variable instruction length and execution time leads to
  - imbalance among stages,
  - complicate hazard detection and precise exceptions
- Caches have similar effects (imbalance pipes)
  - may freeze the entire pipeline on a cache miss
- Complex addressing modes
  - may change register values
  - may require multiple memory access
- self modifying instructions causes pipeline problems
- Implicitly set condition codes complicates pipeline control hazards

(37)