Review

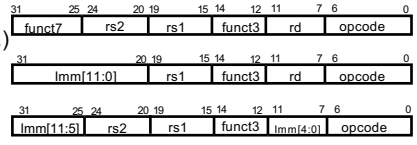
- Last Class:
  - Review of instruction set architecture (ISA)
  - Instruction format
  - Addressing mode
- Today's class:
  - Review of pipeline
  - Single cycle processor → CCT problem
  - Pipelined execution → improved ILP
  - Hazards hurt performance
  - Multi-cycle pipelines
- Announcement and reminder
  - Office hour starts today (virtual, zoom link on syllabus)

(1)

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



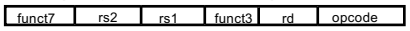
- Review – you should know about:
  - multiplexors,
  - register files,
  - ALU's
  - arithmetic Vs logical shifts
  - program counter (PC), status word (PS) and instruction register (IR),
  - Memory address registers (MAR) and memory data registers (MDR)
  - combinational Vs sequential circuits

(2)

2

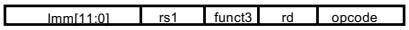
Implementing the RISC architecture (Section C.3)

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



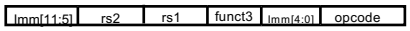
1) fetch instruction
2) read registers rs1 and rs2
3) compute result (use ALU)
4) write to register rd

Load instructions (lw rd, c(rs1))



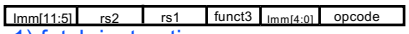
1) fetch instruction
2) read register rs1 (and rs2 ??)
3) use ALU to compute memory address = content of rs1 + c
4) read from memory
5) write to register rd

Store instructions (sw rs2, c(rs1))

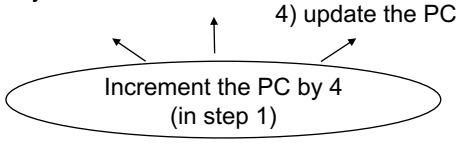


1) fetch instruction
2) read registers rs1 and rs2
3) use ALU to compute memory address = content of rs1 + c
4) write to memory at address stored in rs2

Branch instructions (beq rs1, rs2, L)



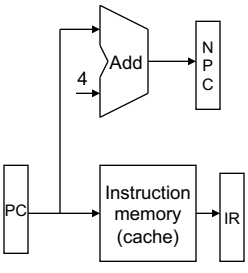
1) fetch instruction
2) read registers rs1 and rs2
3) use ALU to compute branch address = PC + offset and evaluate branch condition (rs1 == rs2)
4) update the PC (if condition satisfied)

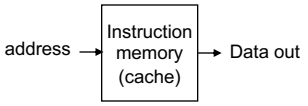


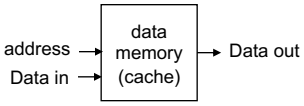
(3)

Instruction fetch

IR ← Mem[PC]
NPC ← PC + 4 /*next PC*/

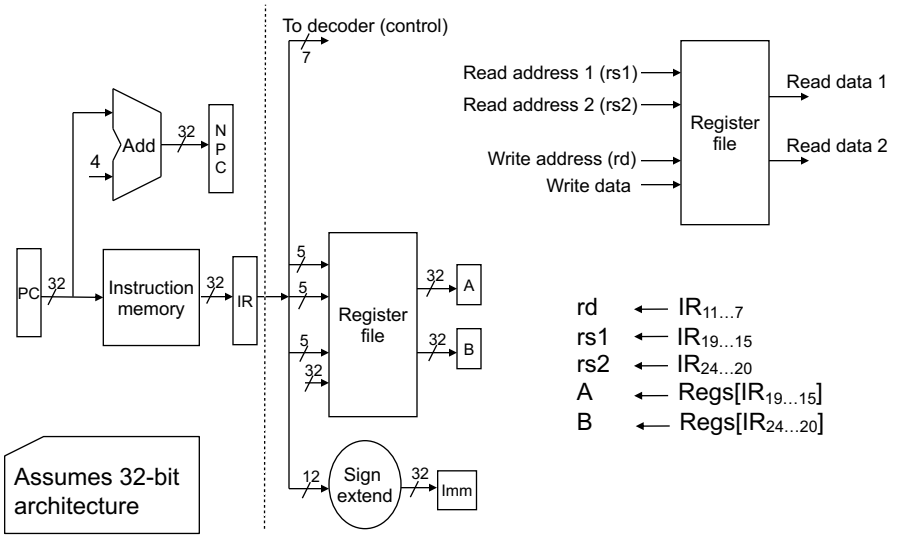






(4)

Register read

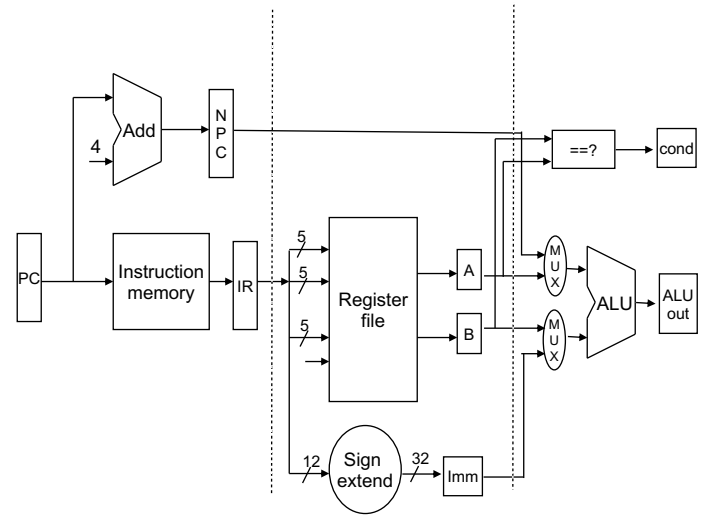


Imm ← $(IR_{31})^{20}$ ## $IR_{31..20}$ (for lw)

Imm ← $(IR_{31})^{20}$ ## $IR_{31..20}$ ## $IR_{11...7}$ (for sw and branch) (5)

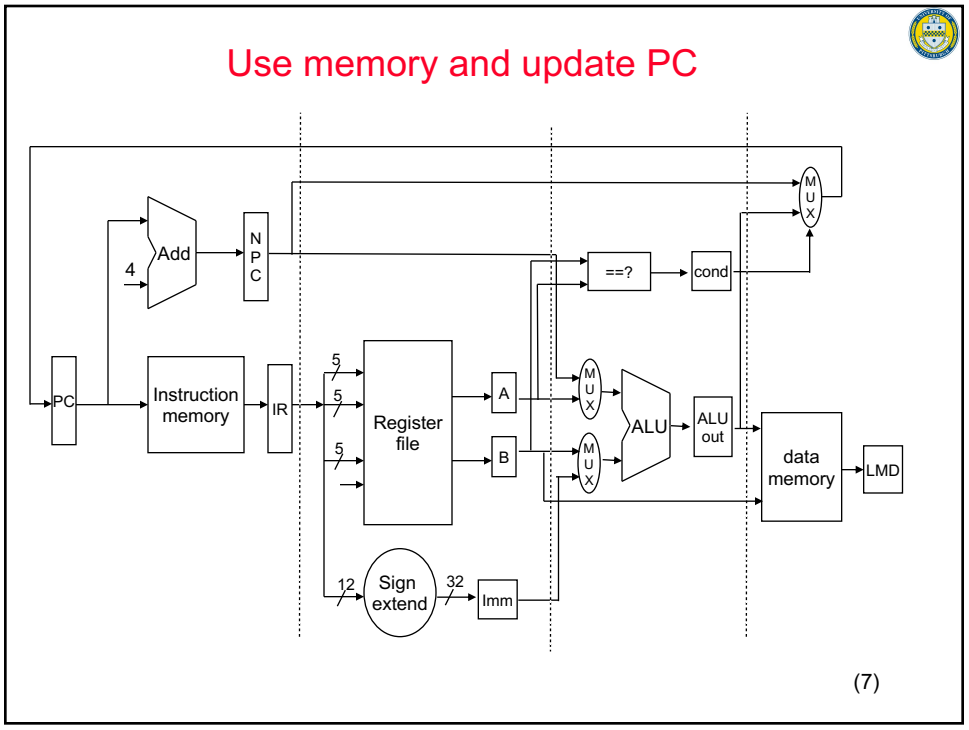
5

Use ALU and evaluate branch condition

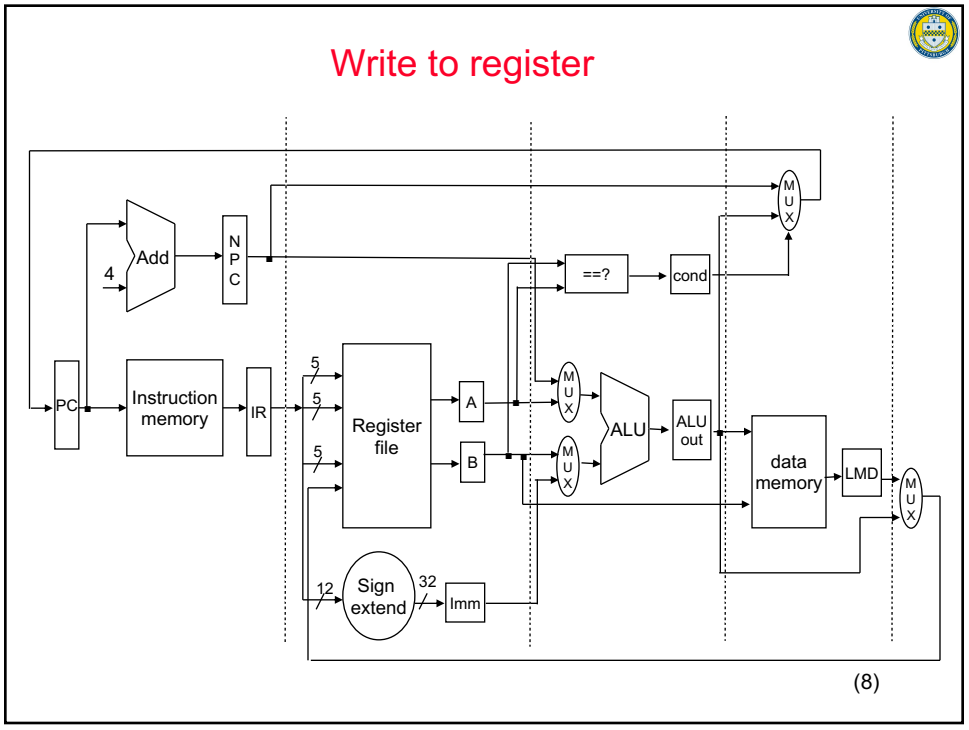


(the multiplexors settings and ALU function depend on the op-code) (6)

6

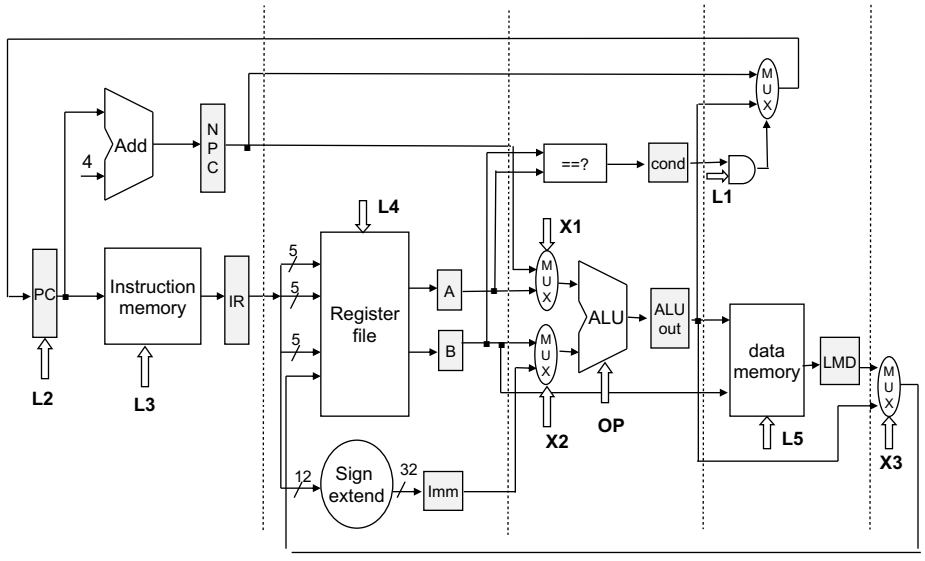

Use memory and update PC
Add
4
N
P
C
M
U
X
==?
cond
PC
Instruction
memory
IR
5
5
5
Register
file
A
B
ALU
ALU
out
data
memory
LMD
12
Sign
extend
32
Imm
(7)

7


Write to register
Add
4
N
P
C
M
U
X
==?
cond
PC
Instruction
memory
IR
5
5
5
Register
file
A
B
ALU
ALU
out
data
memory
LMD
12
Sign
extend
32
Imm
(8)

8

The control signals

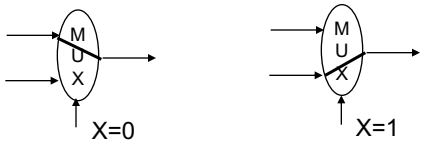


(9)

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)

10

Control Signal Table

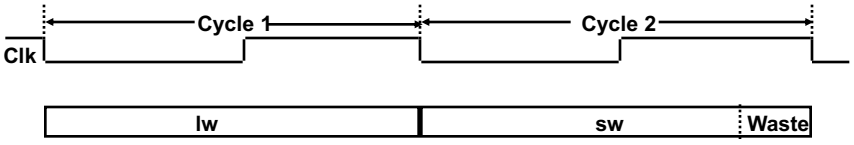
Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUOp for R-type instructions depends on the instructions' func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq *and* the ALU's Zero output is true.

(11)

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point (FP) multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

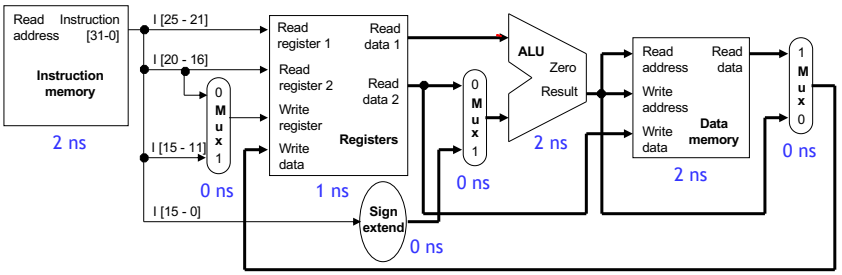
- Single cycle datapath is simple and easy to understand

(12)

CCT Problem

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, lw $t0, –4($sp) needs 8ns, assuming the delays shown here.

reading the instruction memory	2ns	
reading the base register $sp	1ns	
computing memory address $sp-4	2ns	8ns
reading the data memory	2ns	
storing data back to $t0	1ns	

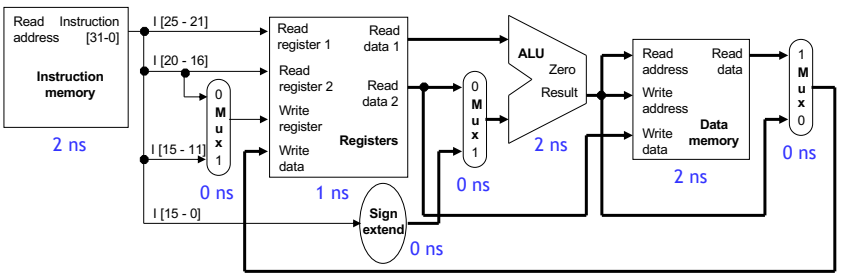


(13)

CCT Problem

- If we make the cycle time 8ns then *every* instruction will take 8ns, even if they don't need that much time.
- For example, the instruction add $s4, $t1, $t2 really needs just 6ns.

reading the instruction memory	2ns	
reading registers $t1 and $t2	1ns	
computing $t1 + $t2	2ns	6ns
storing the result into $s0	1ns	



(14)

Another Example

- With these same component delays, a sw instruction would need 7ns, and beq would need just 5ns.
- Let's consider the gcc instruction mix:

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be:

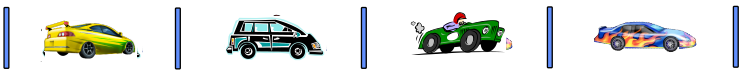
(48% x 6ns) + (22% x 8ns) + (11% x 7ns) + (19% x 5ns) = 6.36ns

- The single-cycle datapath is about 1.26 times slower!

(15)

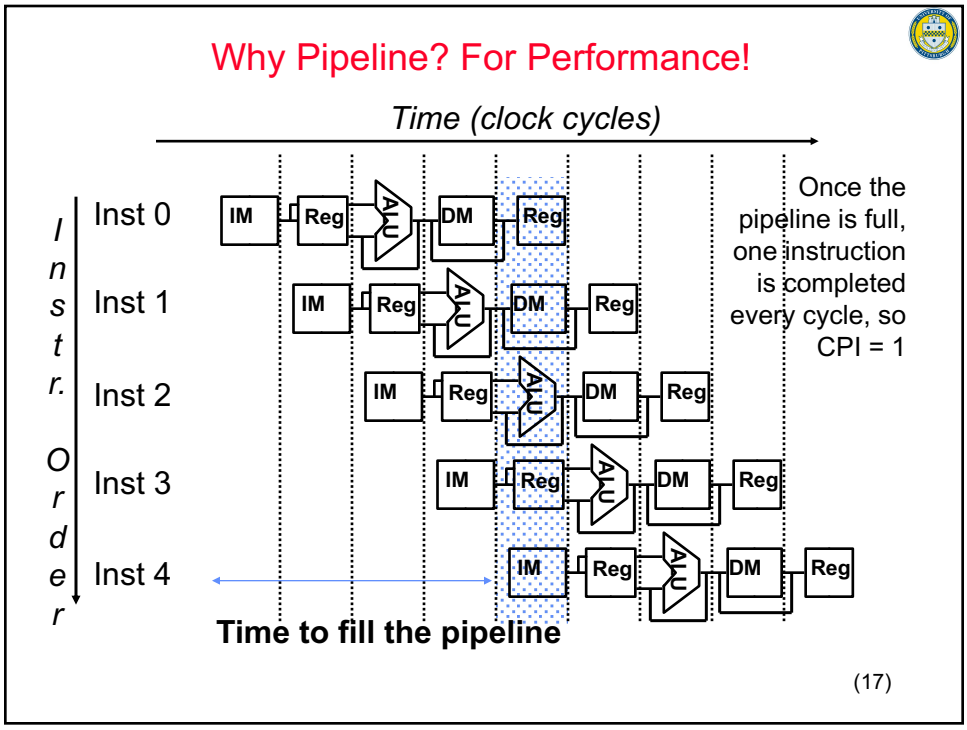
Pipelining: Its Natural!

- Car wash Example

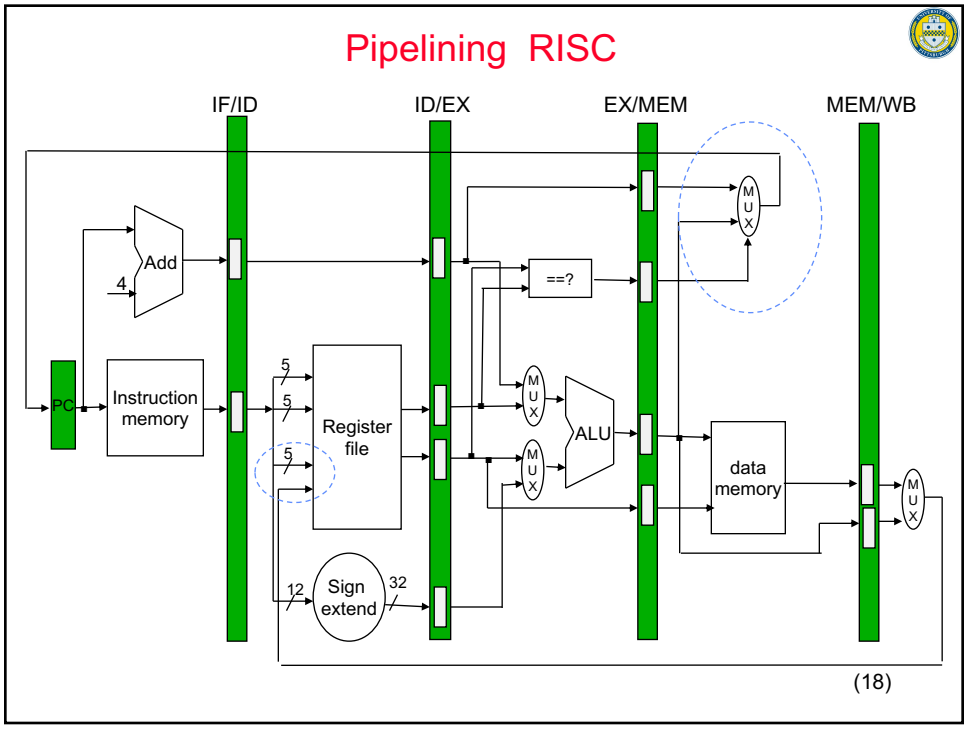


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and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

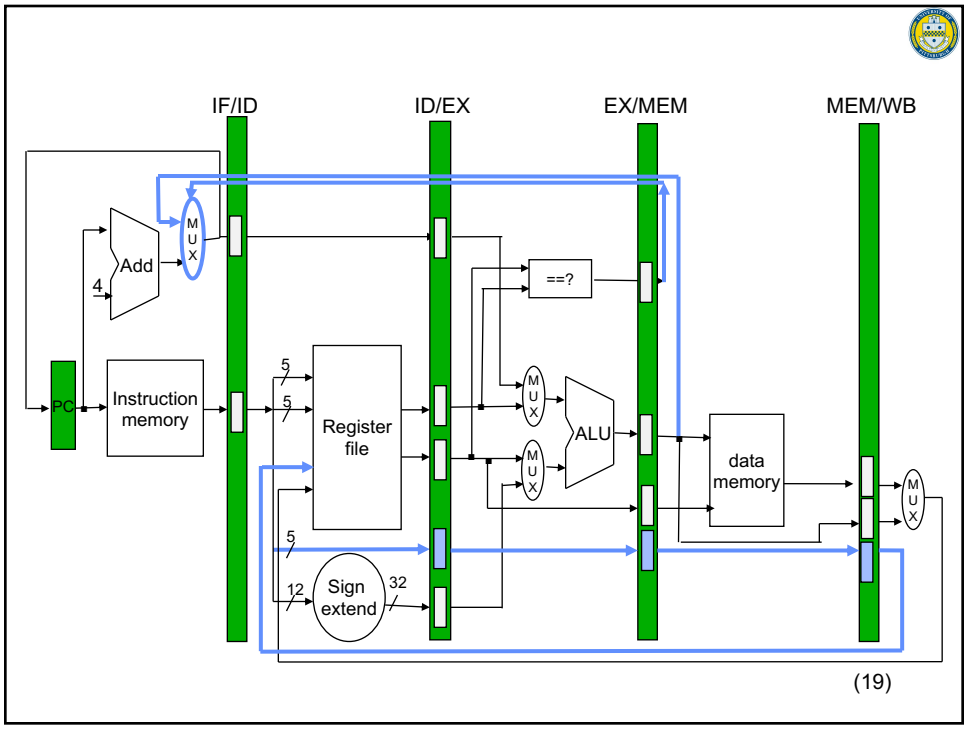
(16)


Why Pipeline? For Performance!
Time (clock cycles)
Instr. Order
Inst 0
Inst 1
Inst 2
Inst 3
Inst 4
IM
Reg
ALU
DM
Reg
Once the pipeline is full, one instruction is completed every cycle, so CPI = 1
Time to fill the pipeline
(17)

17

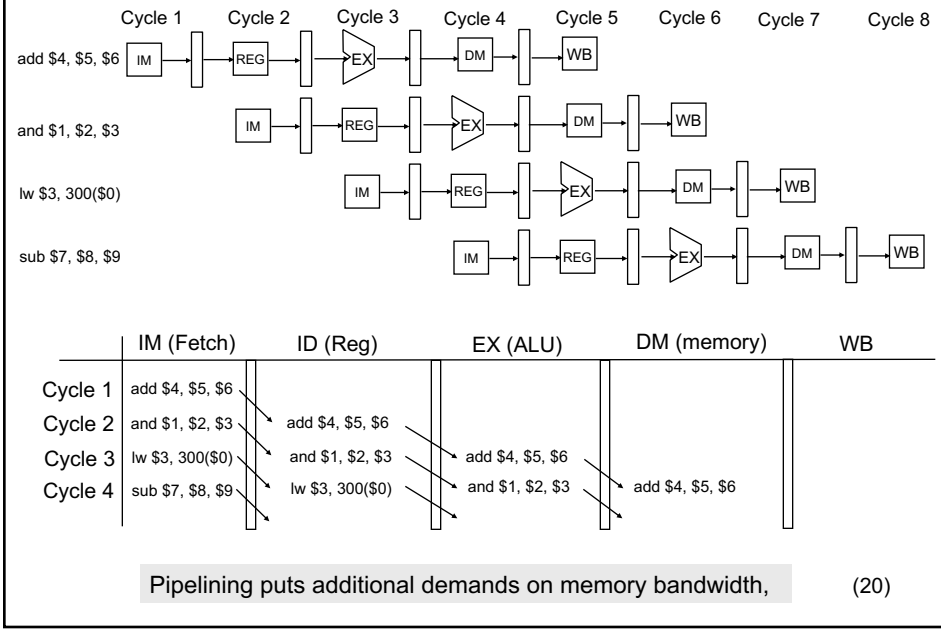

Pipelining RISC
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Instruction memory
5
5
5
Register file
12
Sign extend
32
==?
M U X
ALU
data memory
(18)

18



19

Visualizing Pipelining (two equivalent diagrams)



Pipelining puts additional demands on memory bandwidth, (20)

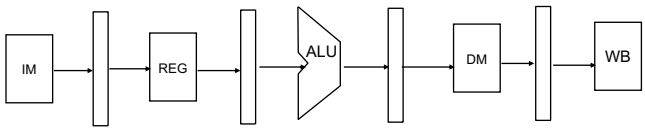
20

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
- Dependencies backward in time cause hazards
- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

(21)

Structural Hazards

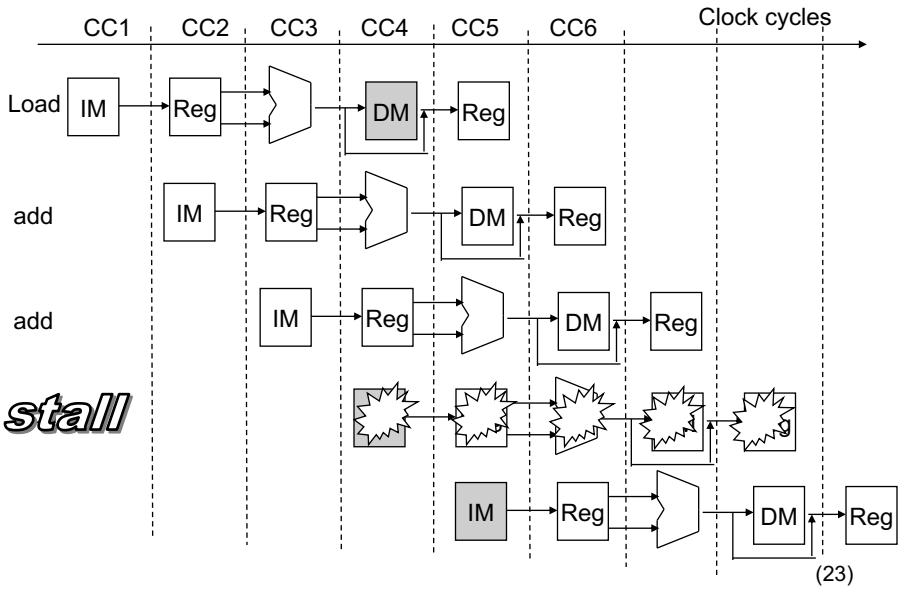


Potential problem: Both REG and WB use the register file
Solution : Read from register file during the first half of a cycle and write to register file during the second half of a cycle.

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

(22)

Structural Hazards (assuming a single memory)



(23)

23

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

$$SpeedUp_A = \text{Pipeline Depth}$$
$$SpeedUp_B = (\text{Pipeline Depth}/1.4) \times 1.05$$
$$= 0.75 \times \text{Pipeline Depth}$$
$$SpeedUpA\ /\ SpeedUpB = 1.33$$

(24)

24

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?)

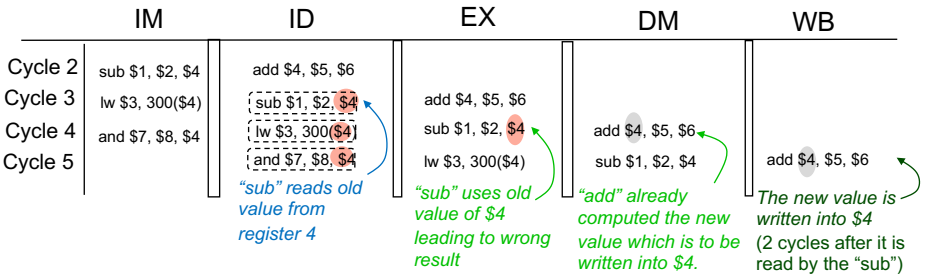
- **Write After Write (WAW)**
  $Instr_{i+k}$ writes operand *before* $Instr_i$ writes it
  - Leaves wrong result
  - Can't happen in RISC 5 stage pipeline (why?)

- Forwarding mitigates data hazards (load-use hazard?)
- Will see WAR and WAW in later more complicated pipes

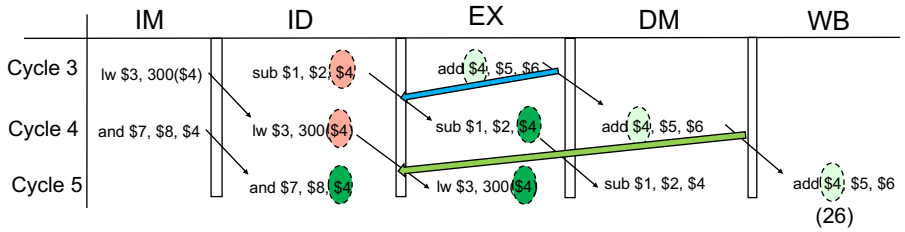
$Instr_i$
.....
$Instr_{i+k}$
.....

(25)

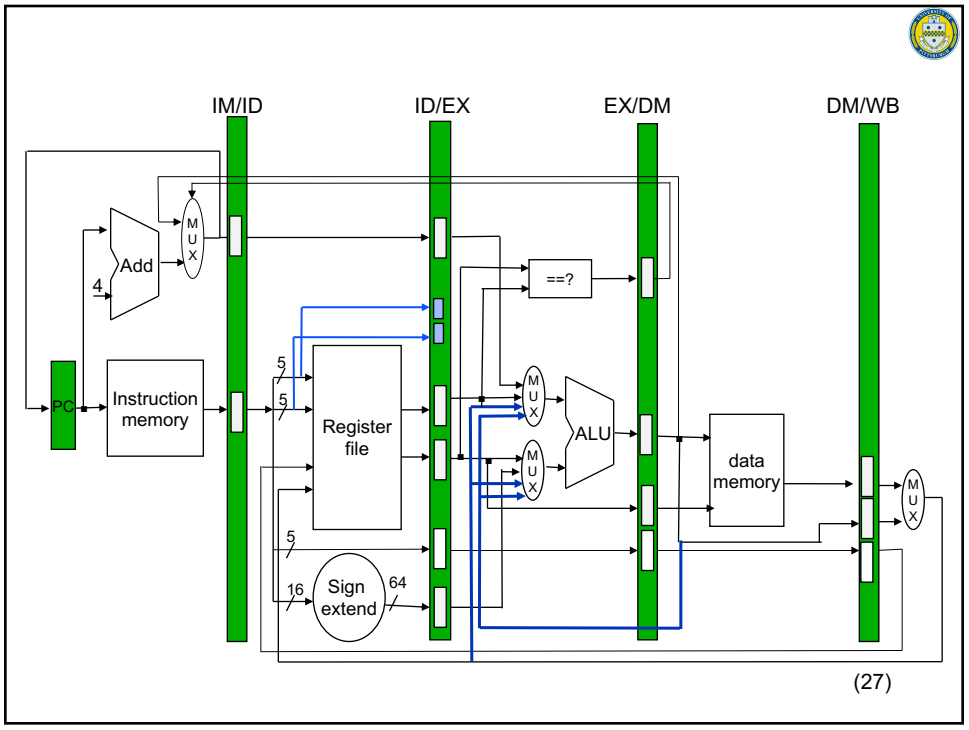
Register-to-register data Hazards



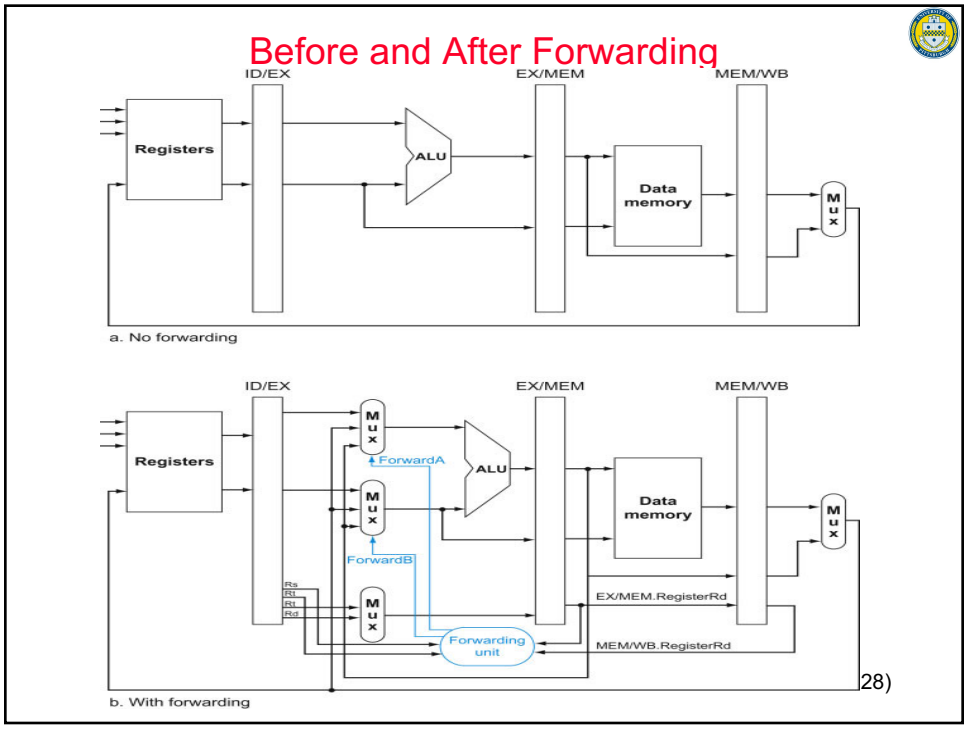
Use forwarding to mitigate data hazards?



(26)

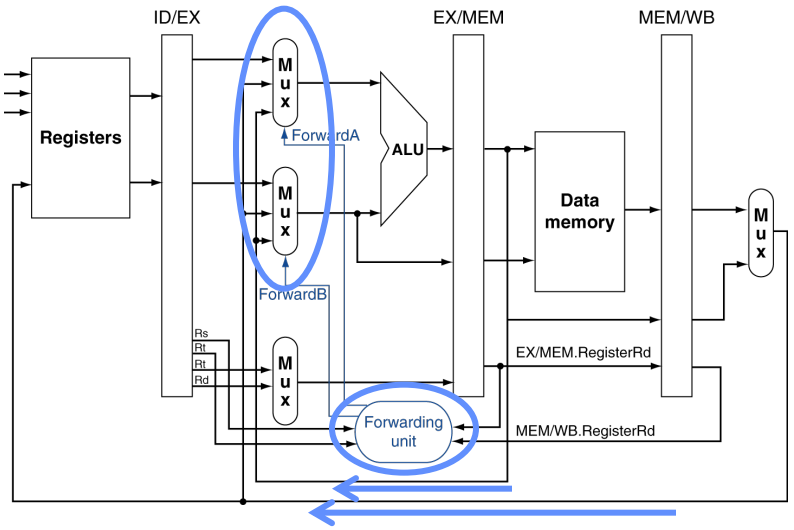

IM/ID
ID/EX
EX/DM
DM/WB
Add
4
M
U
X
==?
PC
Instruction memory
5
5
Register file
ALU
data memory
5
16
Sign extend
64
(27)

27


Before and After Forwarding
ID/EX
EX/MEM
MEM/WB
Registers
ALU
Data memory
a. No forwarding
ForwardA
ForwardB
EX/MEM.RegisterRd
MEM/WB.RegisterRd
Forwarding unit
b. With forwarding
28)

28

Forwarding Logic



(29)

29

Control Values for the Forwarding Multiplexors

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

(30)

30

Data Forwarding Control Conditions

1. EX/MEM Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

2. MEM/WB Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

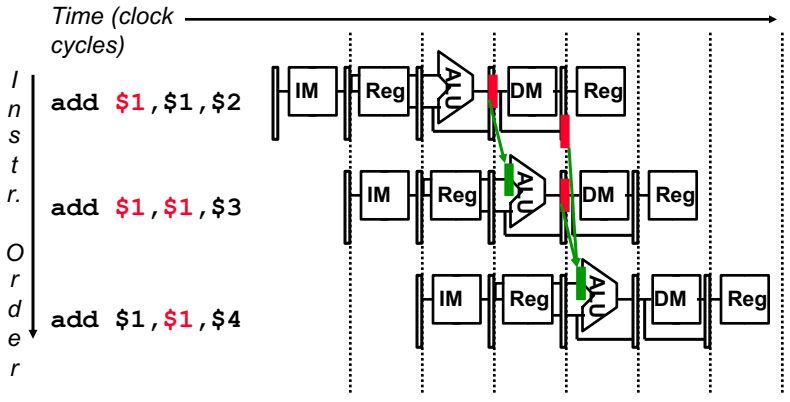
Forwards the result from the second previous instr. to either input of the ALU

(31)

31

Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?



(32)

32

Corrected Data Forwarding Control Conditions

1. EX/MEM Forward Unit:

```
if (EX/MEM.RegWrite
…
```

Forwards the result from the previous instr. to either input of the ALU

2. MEM/WB Forward Unit:

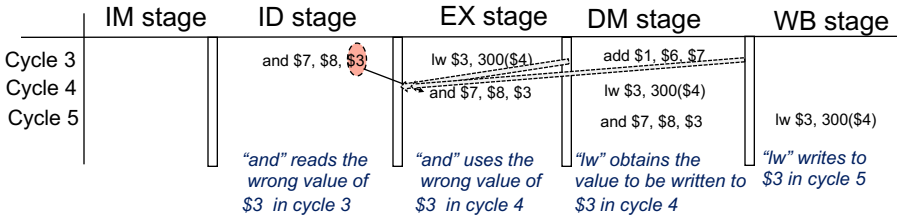
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and !(EX/MEM.RegWrite and (EX/MEM RegisterRD != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
```

Forwards the result from the previous or second previous instr. to either input of the ALU

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and !(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

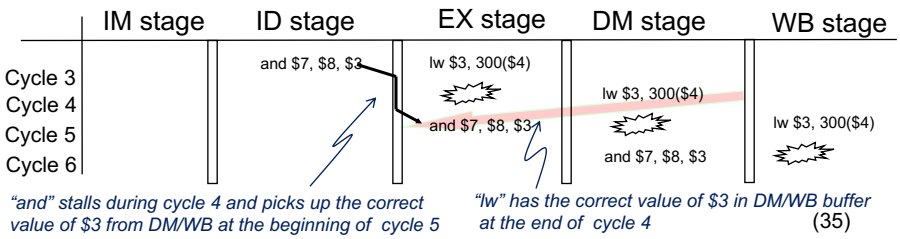
(33)

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.*



(35)

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:

```
lw    Rb, b
lw    Rc, c
add   Ra,Rb,Rc
sw    Ra, a
lw    Re, e
lw    Rf, f
sub   Rd,Re,Rf
sw    Rd, d
```

Fast code:

```
lw    Rb, b
lw    Rc, c
lw    Re, e
add   Ra,Rb,Rc
lw    Rf, f
sw    Ra, a
sub   Rd,Re,Rf
sw    Rd, d
```

(36)