# CpE 313: Microprocessor Systems Design

## Handout 05
## Pipelining

September 13, 2004
Shoukat Ali

shoukat@umr.edu

UMR UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

## MIPS Datapath

## MIPS Datapath

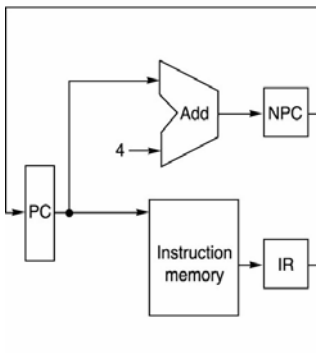
- five execution steps
  - Instruction Fetch
  - Instruction Decode and Register Fetch
  - ALU operation (Execution) *or* Memory Address Computation *or* Branch Target Computation
  - Memory Access
  - Write-back step
- ***INSTRUCTIONS TAKE FROM 4 - 5 CYCLES!***

3

## Step 1: Instruction Fetch

- use PC to get instruction and put it in the Instruction Register
- increment the PC by 4 and put the result in the NPC
  - "next program counter"
- can be described succinctly using RTL (Register-Transfer Language)

IR = Memory[PC];
NPC = PC + 4;

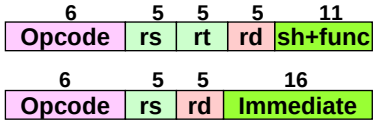



4

## Step 2: Instruction Decode and Register Fetch

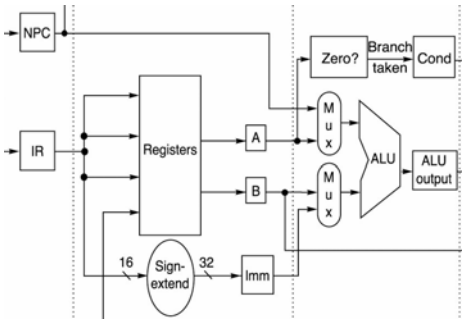
- read source registers rs and rt ***in case*** we need them
- compute the immediate value ***in case*** we need it
- RTL:

  A = Reg[IR[25-21]];
  B = Reg[IR[20-16]];
  Imm = sign-extended (IR[15-0]);




if we are wrong, extra reading doesn't help but doesn't hurt either

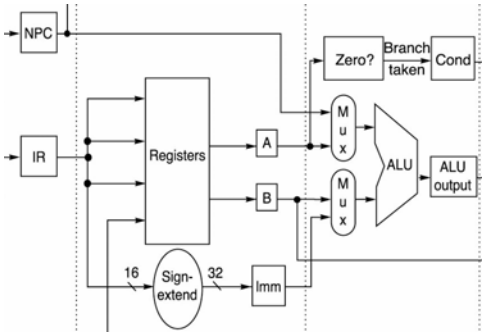
decoding part, not shown here, is done in ***parallel*** by the control unit




5

## Step 3 (Instruction Dependent)

- ALU is performing one of four functions, based on instruction type
  - computing address of memory reference:
    - ALUOut = A + imm
  - R-type:
    - ALUOut = A op B
  - Register-Immediate Op
    - ALUOut = A op imm
  - Branch instruction (assume only `beqz`)
    - ALUOut = NPC + (imm << 2)
    - Cond = 1 if (A == 0) else 0

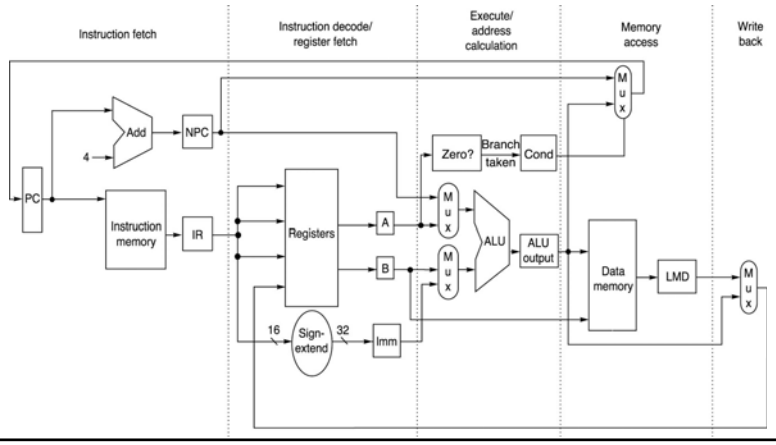



6

## Step 4 (Memory Access/Store-Branch Completion)

- regardless of inst type, PC = NPC
- loads and stores access memory
  - load: LMD = Memory[ALUOut];
  - store: Memory[ALUOut] = B;
    - ***stores done in this step***

- branch instructions complete
  - if (cond), PC = ALUOutput
  - ***branches done in this step***




7

## Step 5 (Write-Back)

- value brought in LMD by the load instruction is now written back to the destination register
  - Reg[IR[20-16]]= LMD
    - ***loads done in this step***

| 6 | 5 | 5 | 5 | 11 |
|---|---|---|---|---|
| Opcode | rs | rt | rd | sh+func |

| 6 | 5 | 5 | 16 |
|---|---|---|---|
| Opcode | rs | rd | Immediate |

- for register-register ALU instructions, the result is now written back into destination register
  - Reg[IR[15-11]]= ALUOutput
    - ***reg-reg ALU instr done in this step***

- register-immediate instructions
  - Reg[IR[20-16]]= ALUOutput

- ***all type of instructions are done by now***

8

## CPI Analysis for Un-Pipelined Case

- branch, 20%
  - IF, ID, EX, MEM
- stores, 10 %
  - IF, ID, EX, MEM
- all other instructions, 70%
  - IF, ID, EX, MEM, WB

- CPI = 5*0.70 + 4*(0.10 + 0.20) = 4.70 cycles per inst on *average*

- can we do better? can we get a smaller CPI with same clock period?
  - ***yes, if we pipeline!***

9

## Pipelining is Natural! Laundry Example

- ***definition: implementation technique where multiple instructions are overlapped in execution***

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


- Washer takes 30 minutes

- Dryer takes 30 minutes

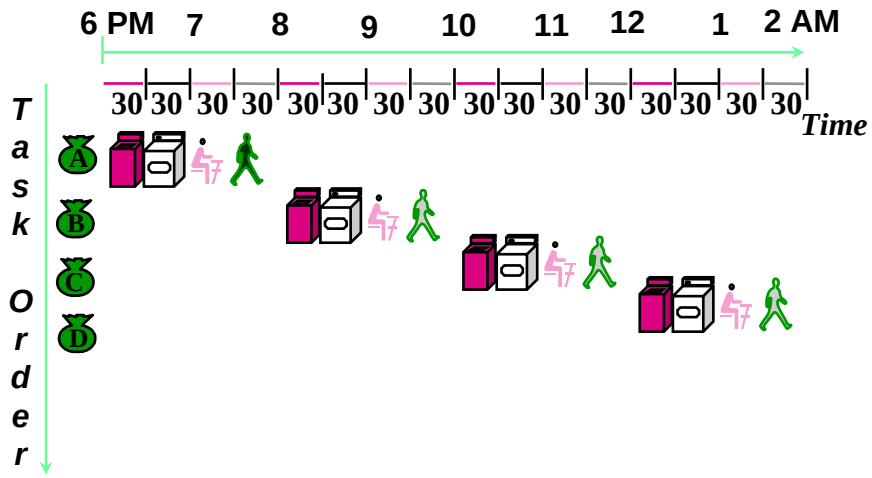
- "Folder" takes 30 minutes

- "Stasher" takes 30 minutes to
  put clothes into drawers

10

## Sequential Laundry

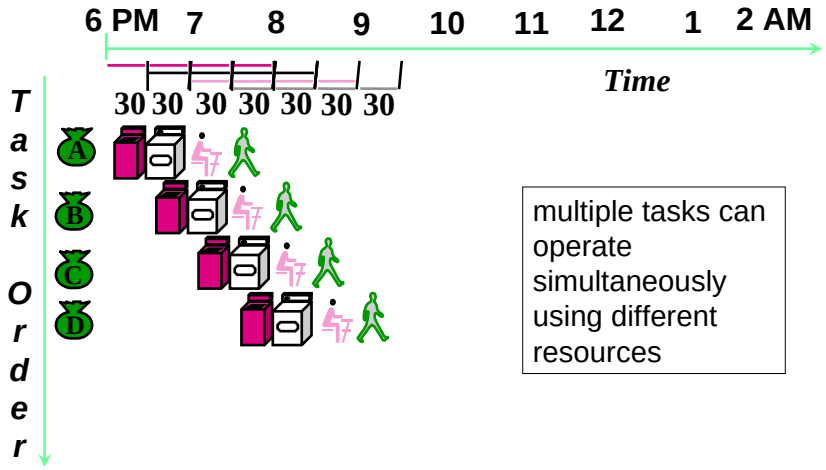
- Sequential laundry takes 8 hours for 4 loads




11

## Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads!



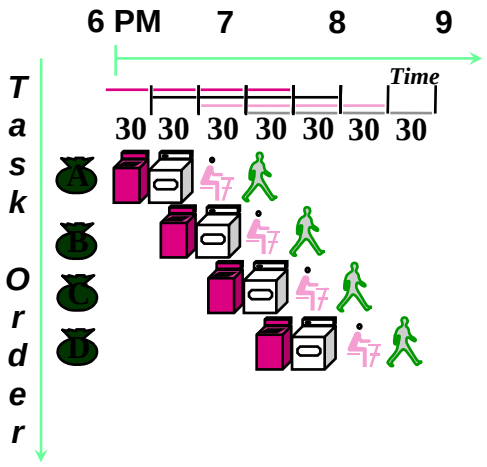

12

## Pipelining Speedup

- what is the speed up for the 4-load case?
  - = (unpipelined time/pipelined time) = 8/3.5 = 2.3
- what is the speed up for the 12-load case?
  - = (unpipelined time/pipelined time) = 24/7.5 = 3.2
- what is the speed up for the 1000-load case?
  - = (unpipelined time/pipelined time) = 2000/501.5 = 3.99
- what is the speed up for the infinite number of loads case?
  - 4

13

## Pipelining Lessons




- pipelining doesn't help latency of single task, it helps ***throughput*** of entire workload
- potential speedup = number pipe stages
  - realized for large number of tasks
  - time to "fill" pipeline reduces speedup
    - 2.3X v. 4X in this example
- how many inst in prog?
  - would pipelining make sense?

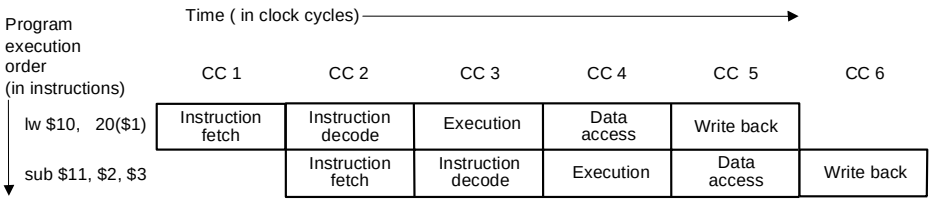
14

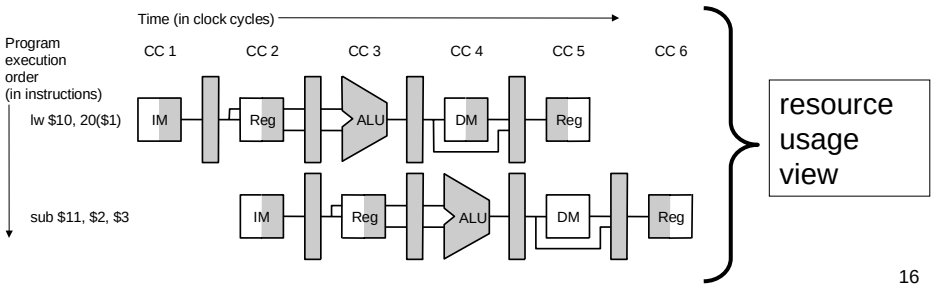
## Instruction Pipelining

- the principles of the laundry example also apply to processors
  - we can pipeline instruction execution
    - a load of laundry → an instruction
    - stages of laundry → stages of an instruction
- pipeline would make great sense because programs typically have millions of instructions to execute

15

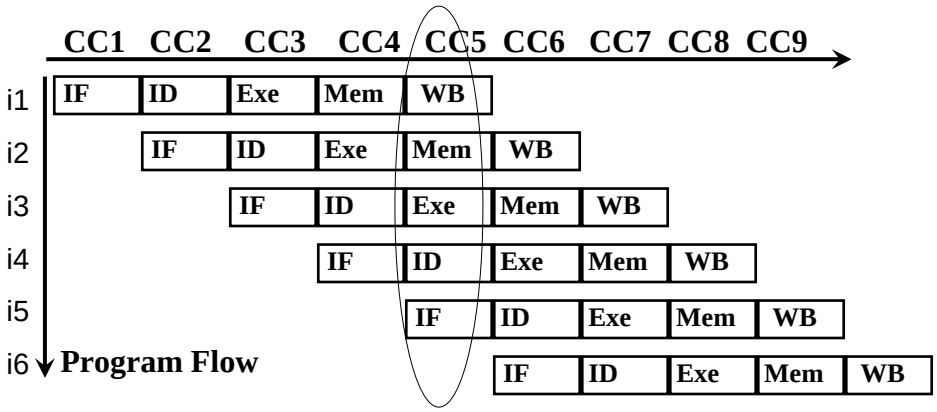
## Pipeline Diagram of Two Instructions




16

## Pipelined Instruction Execution



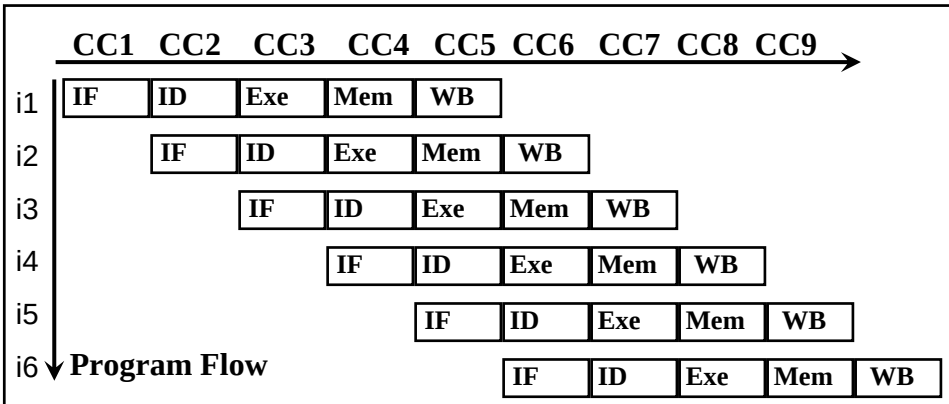

- multiple instructions are in various "stages" in a given clock cycle
  - e.g., in **CC5**, i1 is in stage 5 (WB), i2 is in stage 4 (Mem) while i3 is in stage 3 (Exe) and i4 is in stage 2 (dcd) and i5 is in stage 1 (IF)

17

## Problem!

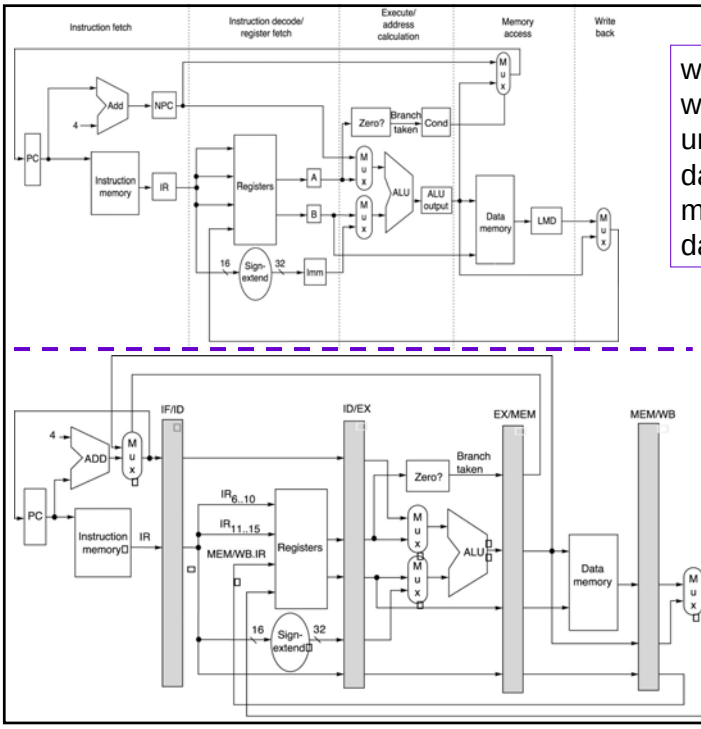
- you said before that, in stage 5,
  - value brought in LMD by the load instruction is now written back to the destination register
    - Reg[IR[20-16]]= LMD
      - ***loads done in this step***
  - for register-register ALU instructions, the result is now written back into destination register
    - Reg[IR[15-11]]= ALUOutput
      - ***reg-reg ALU instr done in this step***
  - register-immediate instructions
    - Reg[IR[20-16]]= ALUOutput
- how does i1 in CC5 access IR to determine IR[15-11] or IR[20-16]?
  - isn't IR being used by i5 in CC5?
    - *must do*: need to retain the values computed in each stage for possible use in other stages
- solution in general: use internal register called pipeline registers to move any relevant info forward as instruction moves forward in the pipeline

18




- CC2: i1 is in ID
- addr of result register rd is produced and then carried forward as i1 enters stage EX in CC3
- rd addr is not used in EX, but still carried forward to stage MEM in CC4
- rd is not used in MEM either, and is carried forward to WB in CC5, where it is used to locate the register where result should be written

19




which registers were present in unpipelined datapath but are no more in pipelined datapath?

*traveling bits:*

*NPC travels with an instruction*

*rd bits travel*

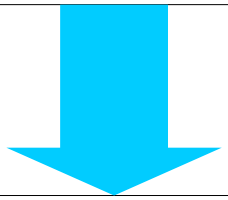
*others?*

20

## Implementing Pipelining

- *"for register-register ALU instructions, the result is now written back into destination register*
  - *Reg[IR[15-11]]= ALUOutput"*

un-pipelined datapath



- for register-register ALU instructions, the result is now written back into destination register
  - Reg[MEM/WB.IR[15-11]]= ALUOutput

pipelined datapath

21

## Pipeline Speedup

- for a pipeline of n stages

$$\text{speedup} = \frac{\text{avg inst execution time unpipelined}}{\text{avg inst execution time pipelined}}$$

$$= \frac{T}{\frac{T}{n} + \text{latch time} \times (n-1)}$$

- if latch time (or pipeline register time) = 0, speedup is n
  - otherwise, speedup decreases as n increases
- what if the stages have different lengths?

22

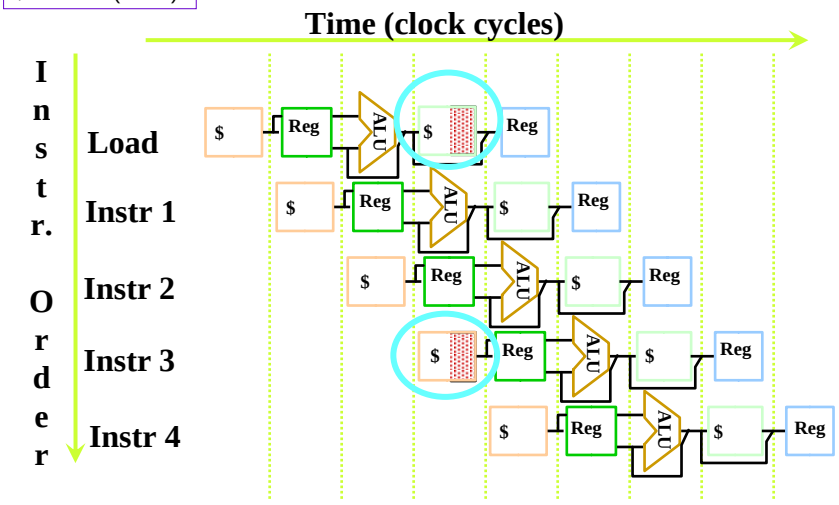
## Pipeline Hazards

- a hazard reduces the performance of the pipeline
  - *a condition that prevents the next instruction from executing in its designated clock cycle*
- three kinds:
  - structural hazards - not enough hardware resources for all combinations of instructions
  - data hazards - dependencies between instructions prevent their overlapped execution
  - control hazards - branches change the PC, which results in late code

23

## Single Cache is a Structural Hazard

$ = cache (cash!)




**Can we read same memory twice in same clock cycle?**

24

## Data Hazards and Forwarding

- what if instructions are dependent?
- in case a data hazard occurs, what do we do?
- let's look at data hazards and forwarding in detail



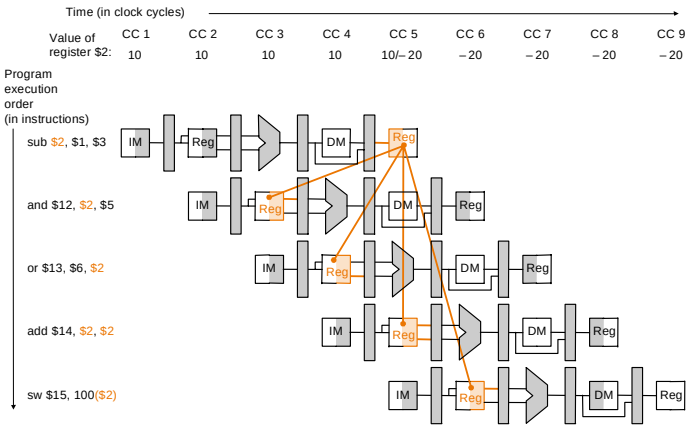
## Data Hazards: Pipeline Dependencies

- Sequence with dependencies:

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

Assume $2 had value 10 before subtraction and -20 afterwards.

Which ones are data hazards?



## Software Solution

- compiler would insert two independent instructions between the **sub** and the **and** instructions.
  - where do we find independent instructions?
    - code re-ordering ring a bell?
  - when no independent instructions can be found, compiler inserts **nop** (no operation)
- sequence with no data hazard:

```
sub   $2, $1, $3
nop
nop
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

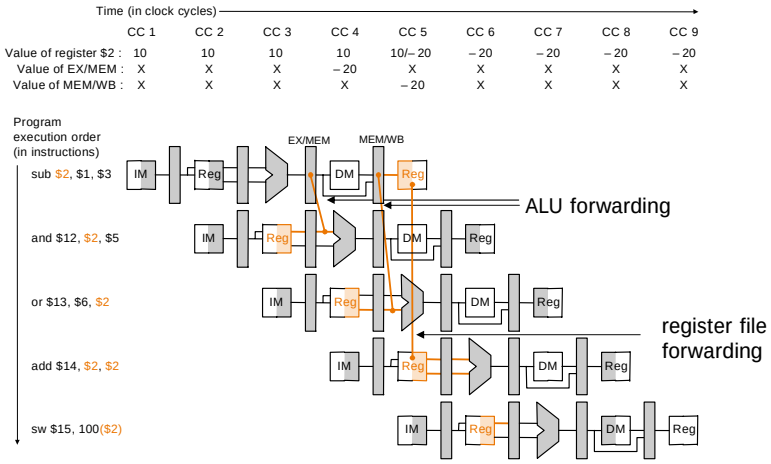
| register-register | register-register | register-register |
|---|---|---|
| load R1, B | load R2, C | load R2, C |
| load R2, C | load R1, B | load R1, B |
| load R3, D | load R3, D | load R3, D |
| mult R4,R2,R3 | mult R4,R2,R3 | sub R6, R1,R2 |
| add R5, R4,R1 | add R5, R4,R1 | add R7, R6,R3 |
| store R5, A | store R5, A | store R7, E |
| sub R6, R1,R2 | sub R6, R1,R2 | mult R4,R2,R3 |
| add R7, R6,R3 | add R7, R6,R3 | add R5, R4,R1 |
| store R7, E | store R7, E | store R5, A |

code re-ordering: one reason we preferred load/store arch over stack and accumulator

27

## Hardware Solution: Forwarding

- use temporary results, don't wait for them to be written in register file
  - ALU forwarding
  - register file forwarding to handle read/write to same register (in same clock cycle)




28