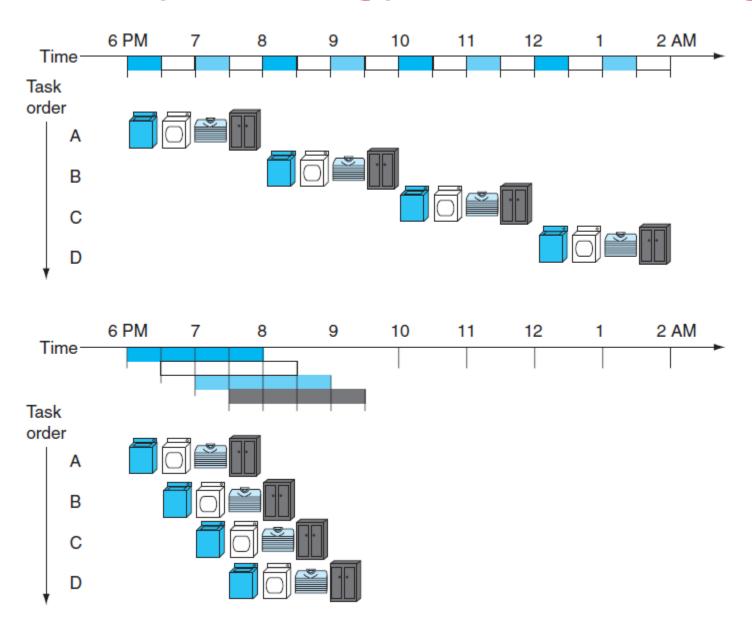# Lecture – 17

Pipelining

- An implementation technique in which multiple instructions are overlapped in execution.

- It is used to make the processor fast.

- Pipelining does not reduce the time to complete a single task but increases the throughput and the improvement in the throughput decreases the total time to complete the task.

- The speed up due to pipelining is equal to the number of stages in the pipeline if

  1. All the stages take about the same amount of time.

  2. The number of tasks is large compared to the number of stages.
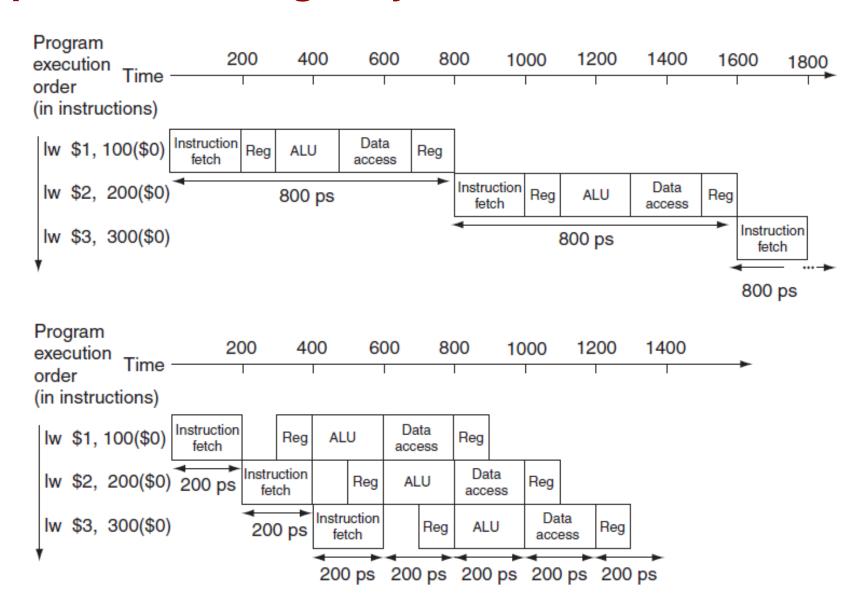
Laundry Analogy for Pipelining

MIPS Pipeline

- Fetch instruction from the memory.

- Read register while decoding the instruction.

- Execute the operation or calculate an address.

- Access an operand in data memory.

- Write the result into a register.

Pipelined vs. Single-Cycle Instruction Execution

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
Load Word (LW)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store Word (SW)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (Beq)	200 ps	100 ps	200 ps			500 ps

Pipelined vs. Single-Cycle Instruction Execution

Speed Up of Pipelining

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- Time between the instruction must be 160 ps. We get 200 ps due to imperfectly balanced stage.

- With respect to total execution time, we get 2400 ps / 1400 ps = 1.7. This is due to small number of instruction.

- Suppose, we have 1,0000,003 instructions. So, speed up = 1,0000,000 x 200 + 1400 / 1,0000,000 x 800 + 2400 ≈ 4.00

Pipelining MIPS

- **all instructions are same length**

  -so fetch and decode stages are similar for all instructions

- **just a few instruction formats**

  -simplifies instruction decode and makes it possible in one stage

- **memory operands appear only in load/stores**

  -so memory access can be deferred to exactly one later stage

- **operands are aligned in memory**

  -one data transfer instruction requires one memory access stage

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called **hazards**.

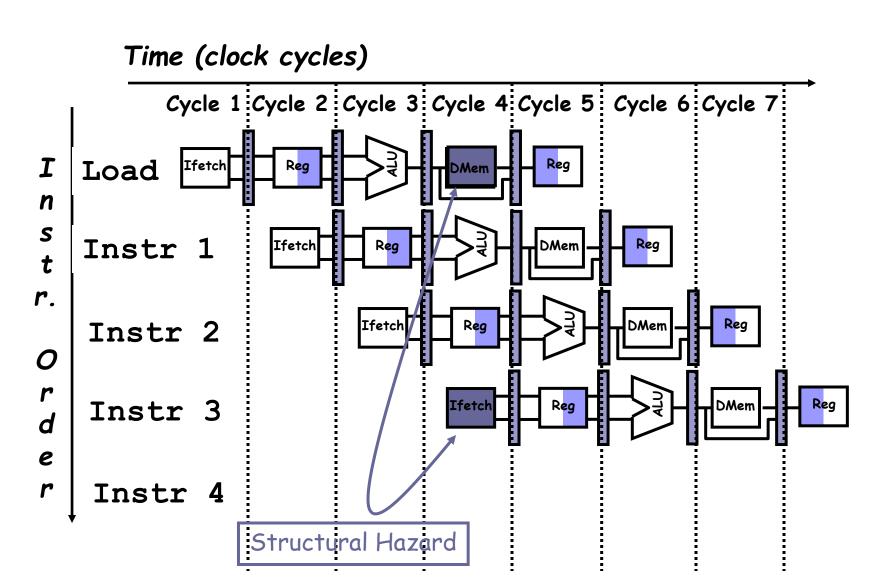There are three different types of hazards:

1. Structural Hazards
2. Data Hazards
3. Control Hazards

Structural Hazards

- An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardwire cannot support the combination of instruction that are set to execute in the given clock cycle.

- E.g., suppose *single – not separate –* instruction and data memory in pipeline below with *one read port*
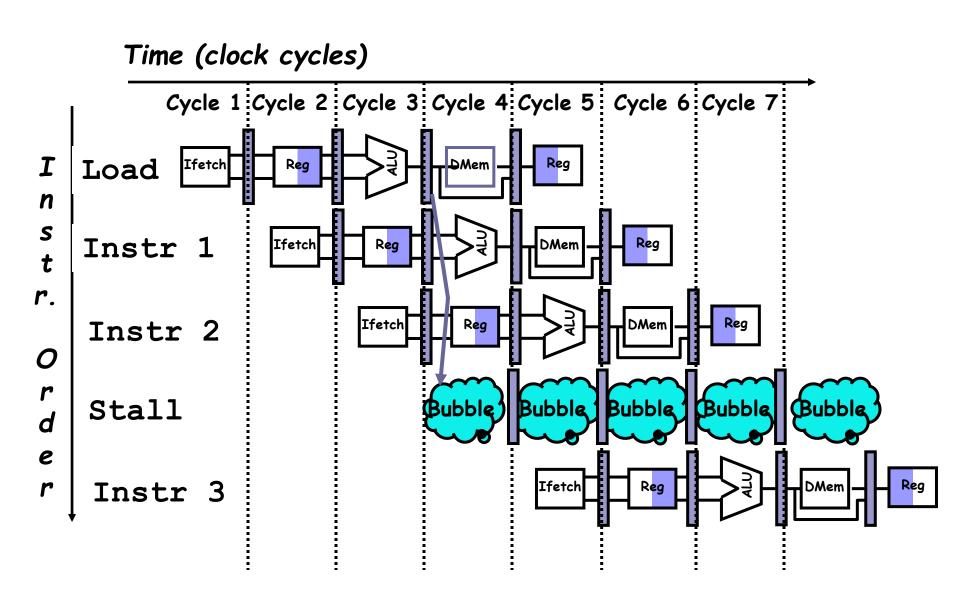 - then a structural hazard between first and fourth `lw` instructions

Structural Hazard

Resolving Structural Hazards

- **Solution 1: Wait**
  - Must detect the hazard
  - Must detect the mechanism to stall

- **Solution 2: Throw more hardware at the problem**
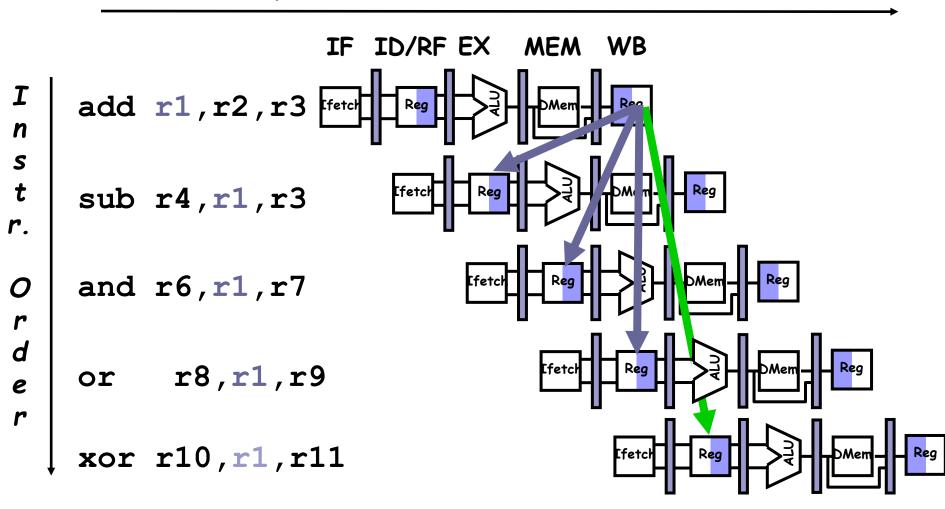
Detecting and Resolving Structural Hazard

Time (clock cycles)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7

Instr. Order

Load — Ifetch | Reg | ALU | DMem | Reg

Instr 1 — Ifetch | Reg | ALU | DMem | Reg

Instr 2 — Ifetch | Reg | ALU | DMem | Reg

Stall — Bubble | Bubble | Bubble | Bubble | Bubble

Instr 3 — Ifetch | Reg | ALU | DMem | Reg

Data Hazards

- An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

- Instruction needs data from the result of a previous instruction still executing in pipeline
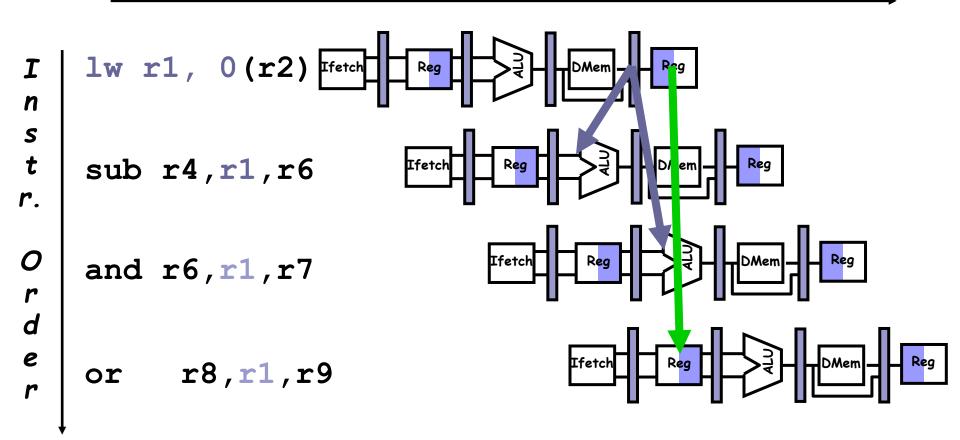
Data Hazards
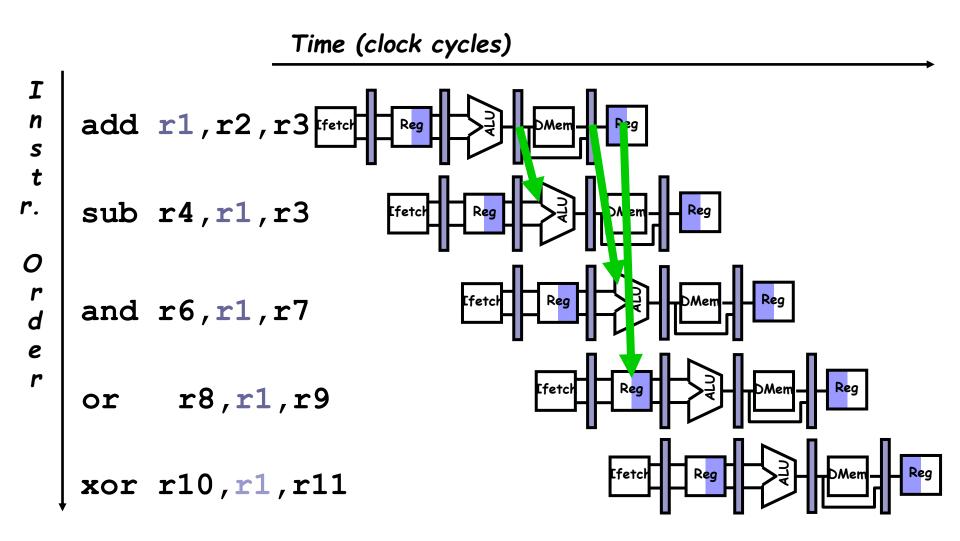
Forwarding to Avoid Data Hazard

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

Data Hazard Even with Forwarding (Load Data Hazard)

Time (clock cycles)

Instr. Order

`lw r1, 0(r2)`	Ifetch / Reg / ALU / DMem / Reg
`sub r4,r1,r6`	Ifetch / Reg / ALU / DMem / Reg
`and r6,r1,r7`	Ifetch / Reg / ALU / DMem / Reg
`or  r8,r1,r9`	Ifetch / Reg / ALU / DMem / Reg
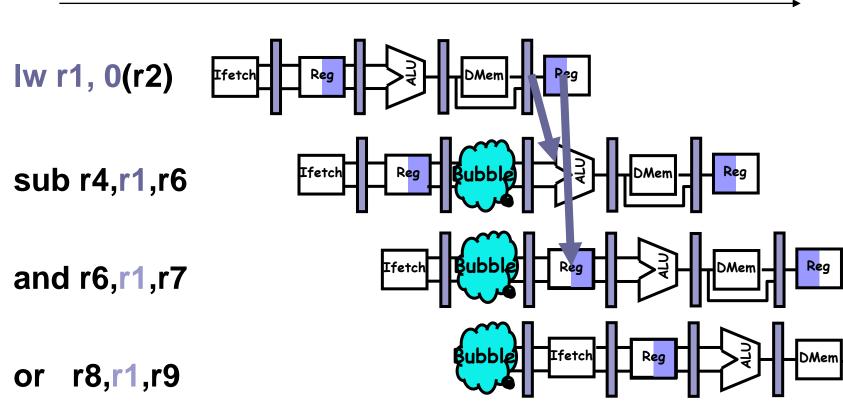
Resolving the Load Data Hazard

Time (clock cycles)

Instr. Order

lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9
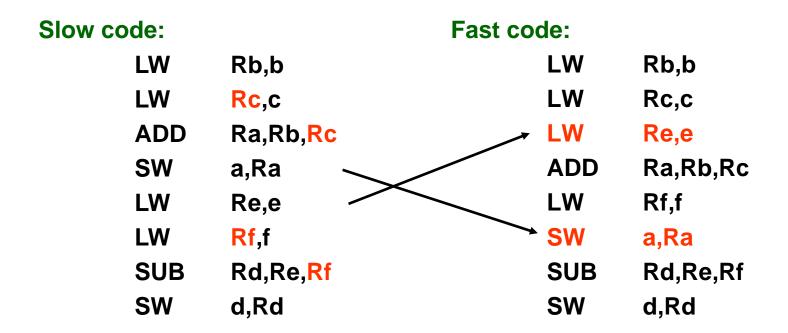
Software Scheduling to Avoid Load Hazards

Try producing fast code for

       a = b + c;

       d = e – f;

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

Slow code:

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

Fast code:

LW	Rb,b
LW	Rc,c
LW	Re,e
ADD	Ra,Rb,Rc
LW	Rf,f
SW	a,Ra
SUB	Rd,Re,Rf
SW	d,Rd
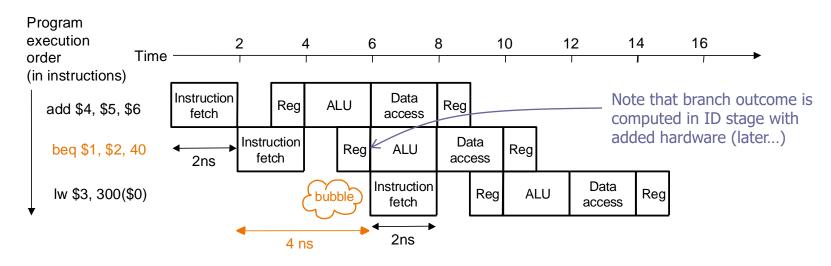
Control Hazards/ Branch Hazards

- An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed.

- Arises from the need to make a decision based on the result of one instruction while others are executing.

Control Hazards

- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- **Solution 1** *Stall* **the pipeline**
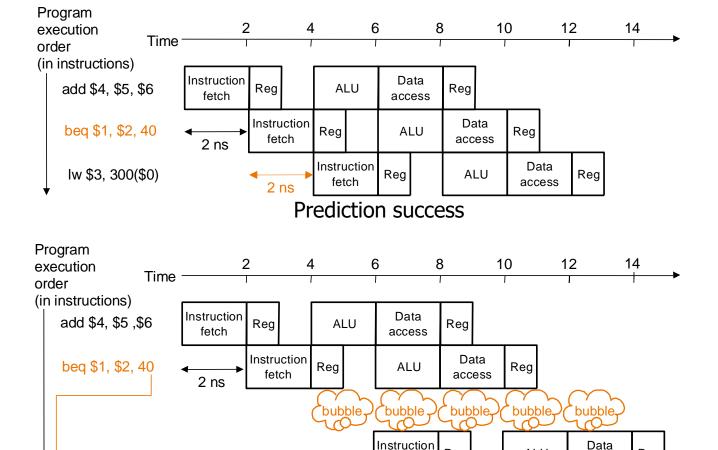
Pipeline stall

Control Hazards

- **Solution 2** *Predict* **branch outcome**

  e.g., predict *branch-not-taken* :

Prediction success
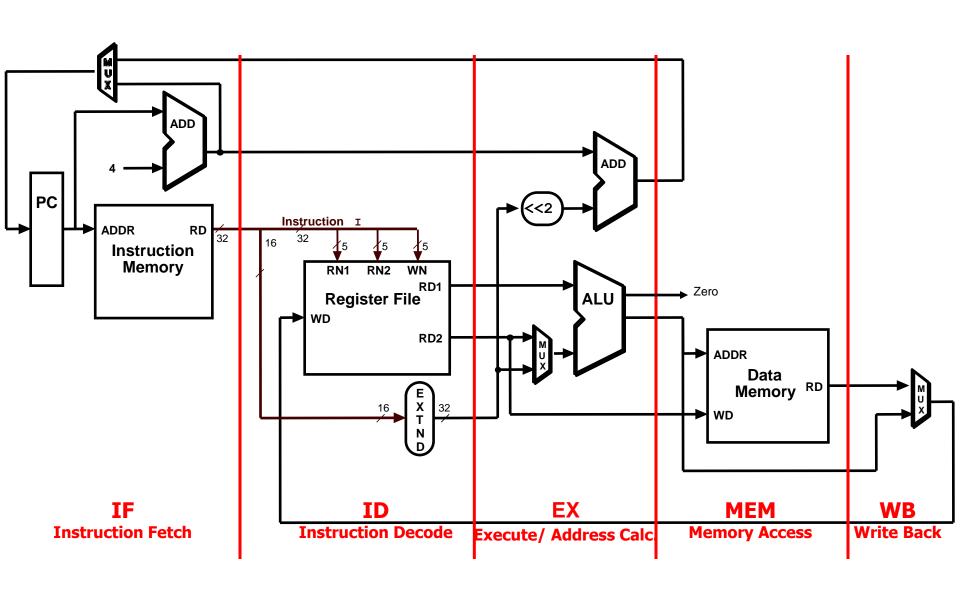
Prediction failure: undo (=flush) `lw`
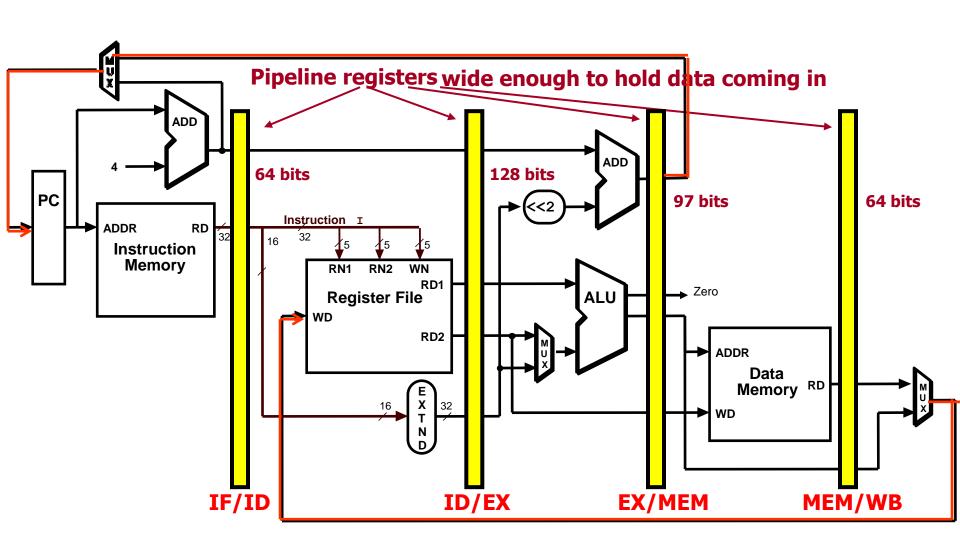
Single-Cycle Datapath "Steps"

IF
Instruction Fetch

ID
Instruction Decode

EX
Execute/ Address Calc

MEM
Memory Access

WB
Write Back
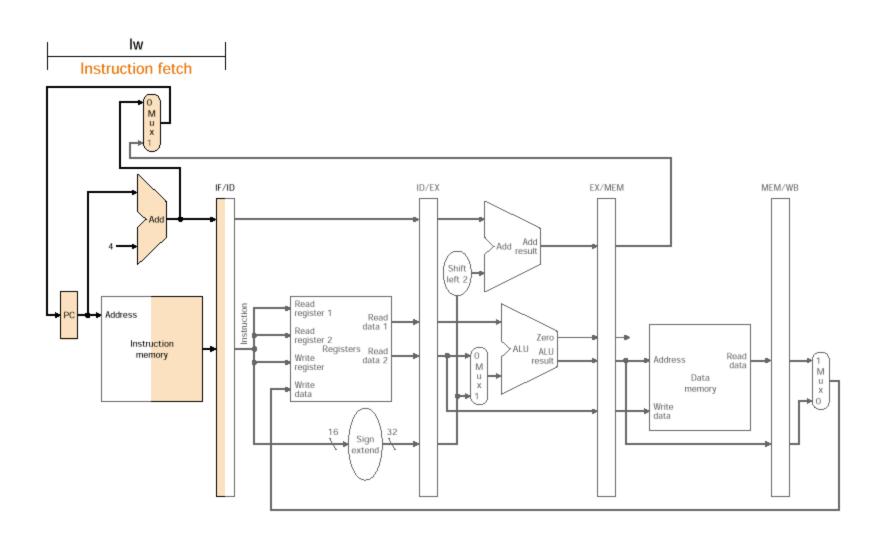
Pipelined Datapath

Pipeline registers wide enough to hold data coming in

PC

ADD

4

ADDR   RD

Instruction Memory

64 bits

32

16   32

Instruction I

5   5   5

RN1   RN2   WN

Register File

WD

RD1

RD2

16

E X T N D

32

128 bits

<<2

ADD

ALU

MUX

Zero

97 bits

ADDR

Data Memory   RD

WD

64 bits

MUX

IF/ID

ID/EX

EX/MEM

MEM/WB
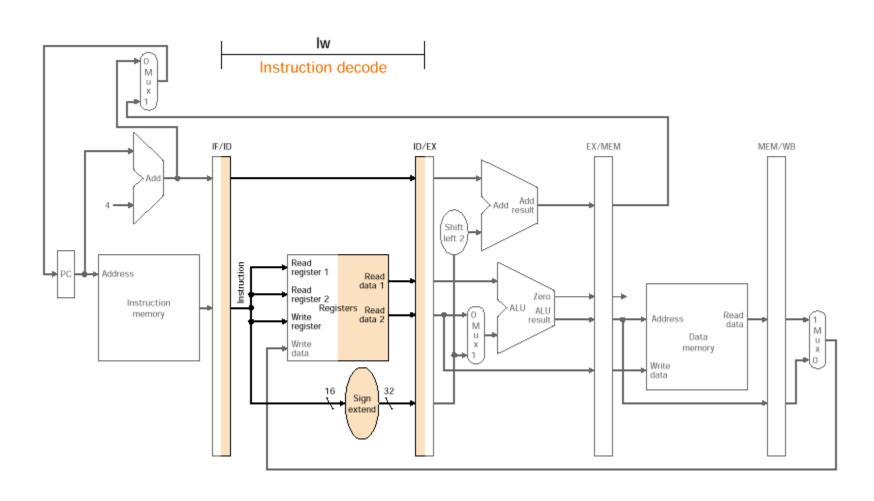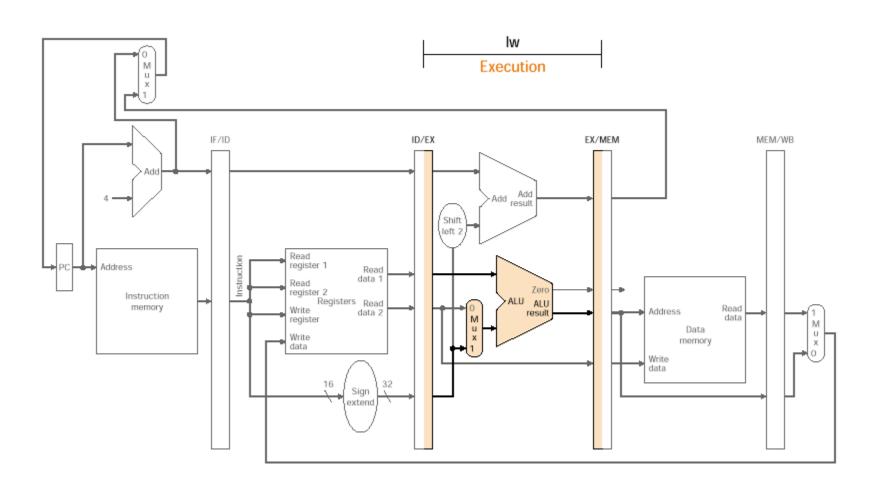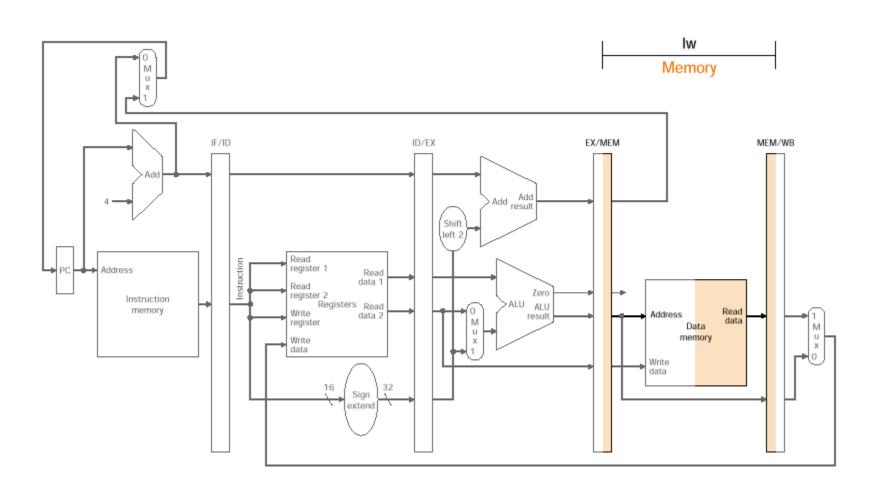
Pipelined Datapath
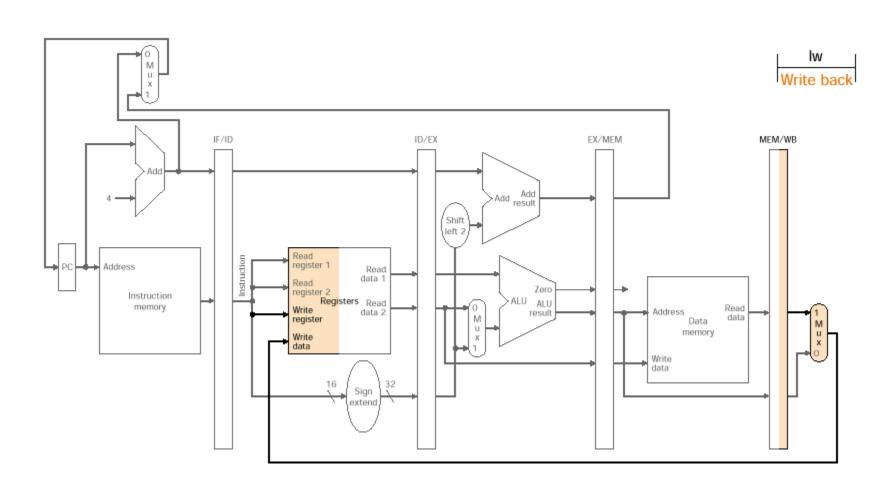
- No pipeline register at the end of write back stage. In this stage we write a register in the register file and any later instruction can get this data by reading this register. So no need for redundant register.

- Each component of the data path is associated with a single stage.

- Each register contains a portion of the instruction needed for that stage and later stages.
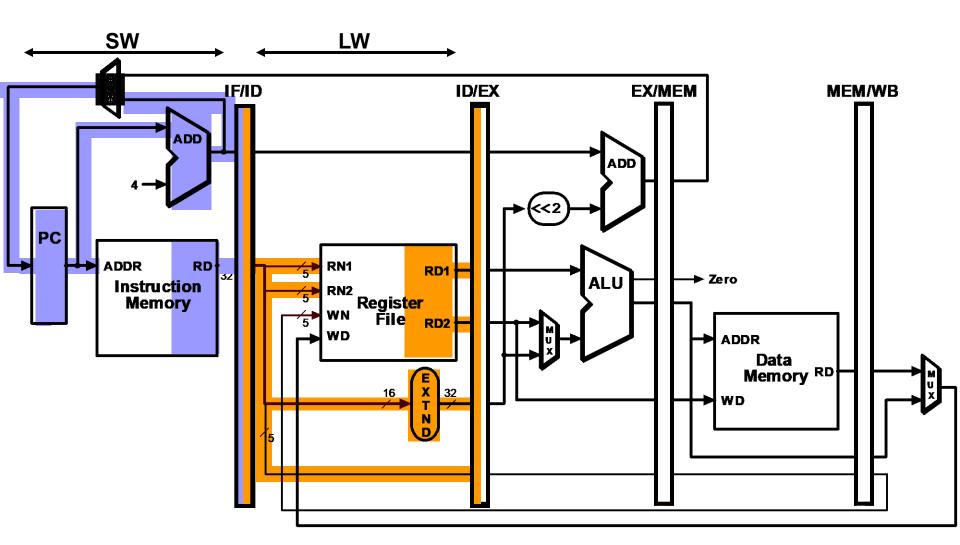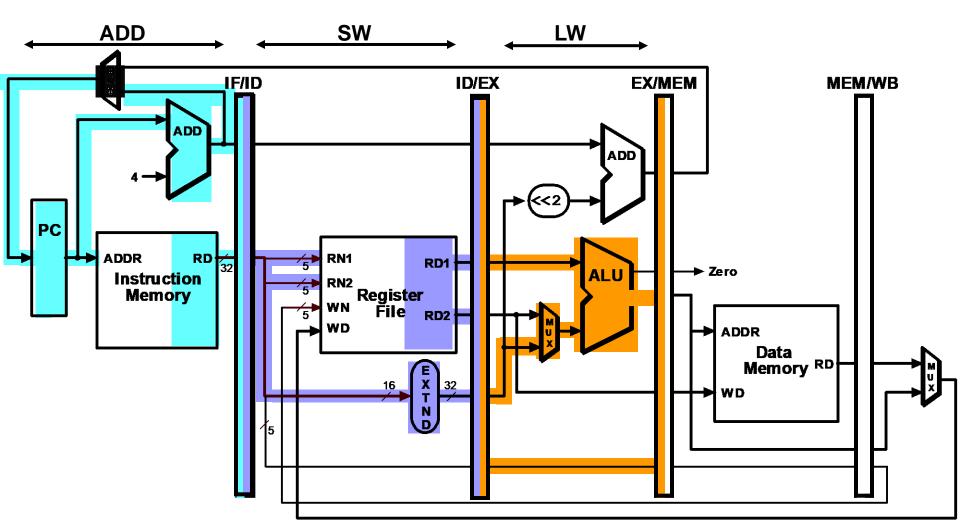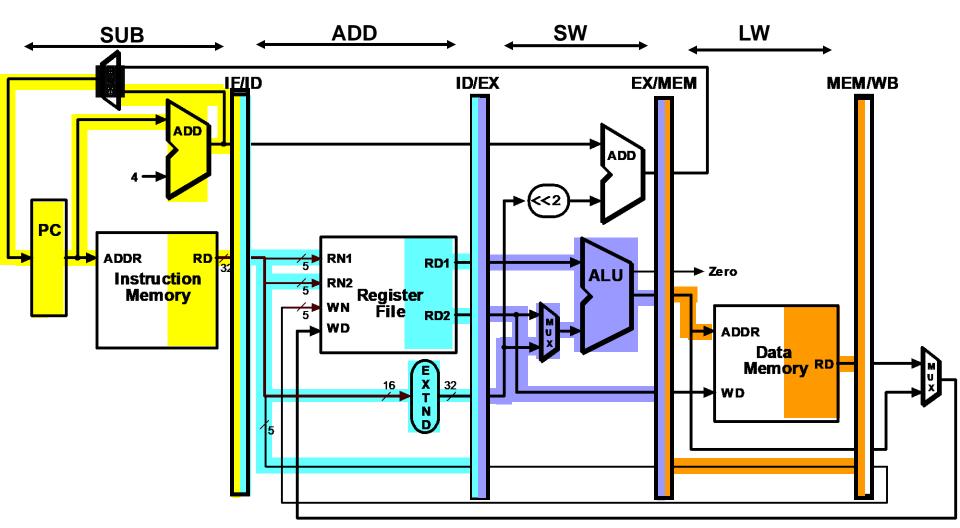
Pipeline Flow (lw example)

Pipeline Flow (lw example)

Pipeline Flow (lw example)

Pipeline Flow (lw example)

Pipeline Flow (lw example)
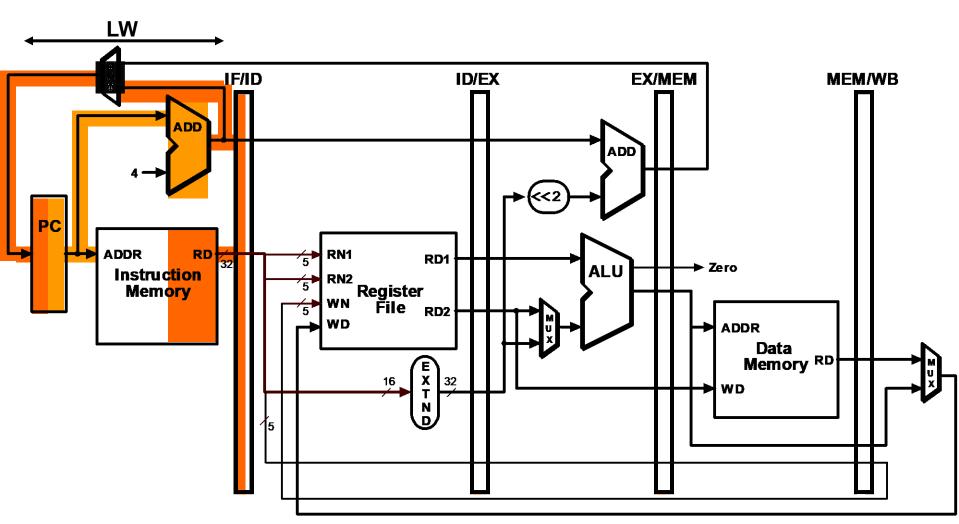
Pipelined Example

- Consider the following instruction sequence:
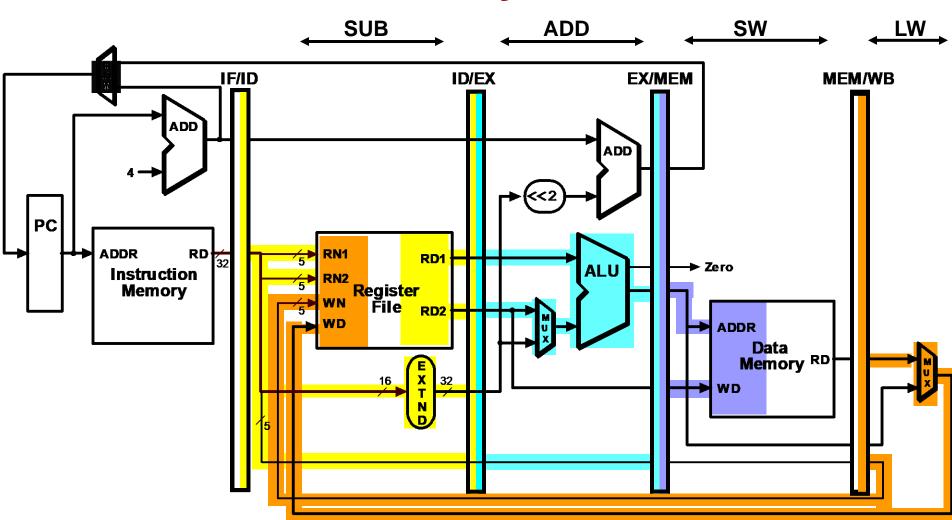
    lw $t0, 10($t1)

    sw $t3, 20($t4)

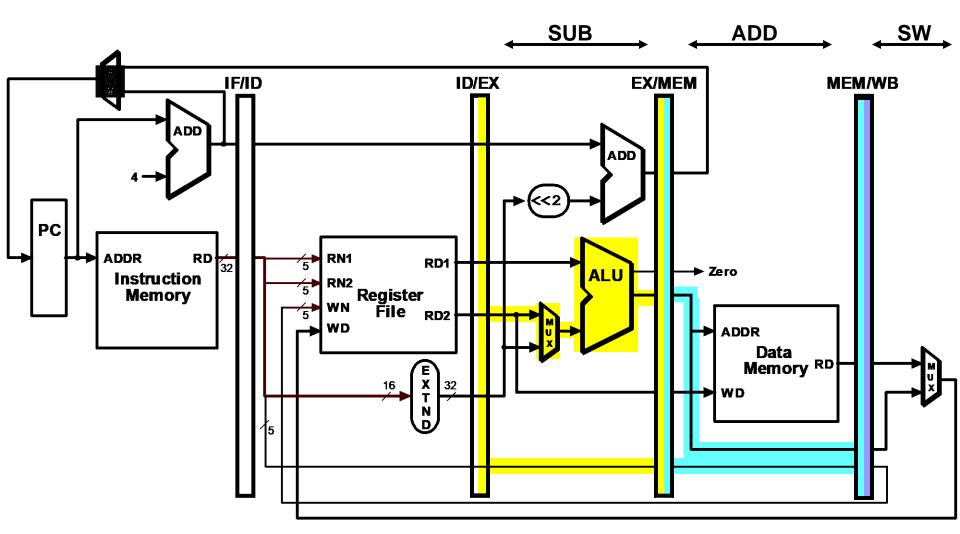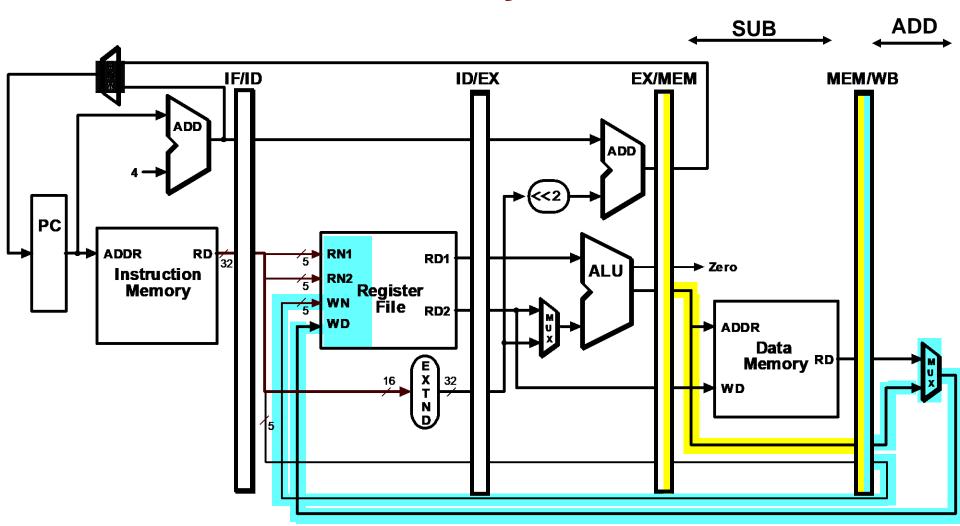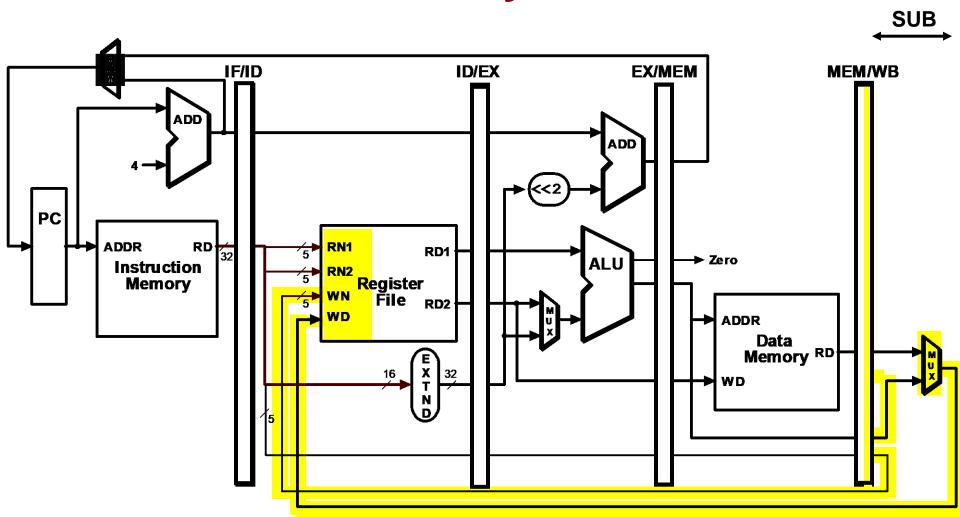   **add $t5, $t6, $t7**

    sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 1

Single-Clock-Cycle Diagram: Clock Cycle 2

Single-Clock-Cycle Diagram: Clock Cycle 3

Single-Clock-Cycle Diagram: Clock Cycle 4

Single-Clock-Cycle Diagram: Clock Cycle 5

Single-Clock-Cycle Diagram: Clock Cycle 6

Single-Clock-Cycle Diagram: Clock Cycle 7

Single-Clock-Cycle Diagram: Clock Cycle 8
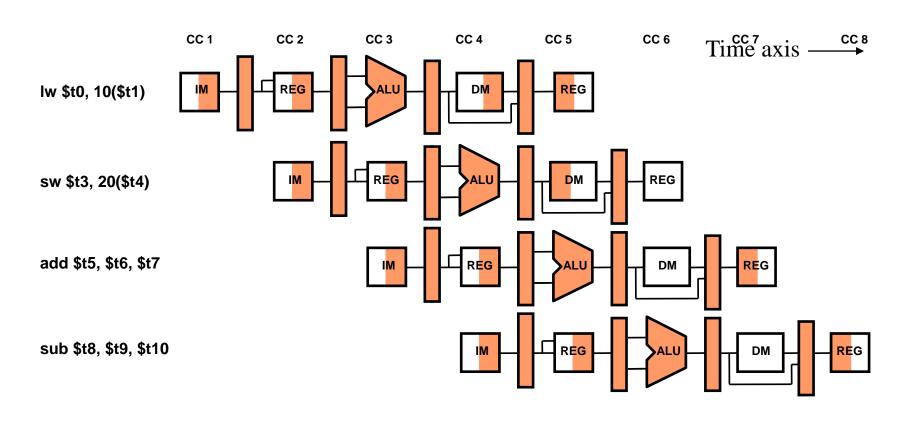
Alternative View – Multiple-Clock-Cycle Diagram

Another View – Multiple-Clock-Cycle Diagram