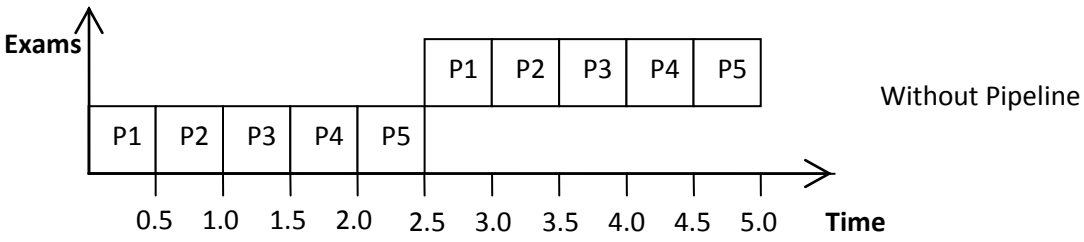
Ch6: Enhancing Performance with Pipelining

High-Performance Processors

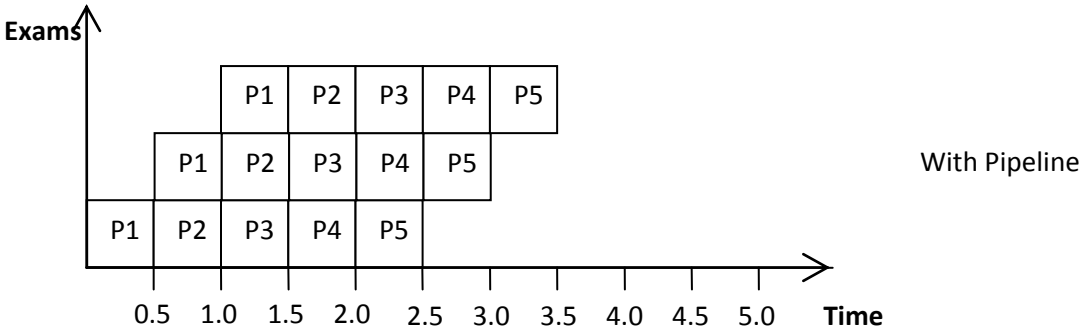
- Two techniques for designing high-performance processors:
  - Pipelining
  - Multiprocessing
- Both techniques exploit parallelism:
  - Pipelining: parallelism among multiple instructions. (multiple instructions are overlapped in execution)
  - Multiprocessing: parallelism among multiple processors
- Pipelining Principles
  - Pipelining does not help the latency of a single task, but it helps the throughput of the entire workload.
  - Multiple tasks are processed simultaneously.

Key Ideas behind Pipelining

- Grading the midterm exams:
  - Each exam has 5 problems, and there are five people grading the exam.
  - Each person ONLY grades one problem.
  - Pass the exam to the next person as soon as one finishes his part
  - Assume each problem takes 0.5 hour to grade
  - Each individual exam still takes 2.5 hours to grade
  - But with 5 people pipelined technique, all exams can be graded much quicker
    - The second exam can start as soon as the 1st finishes its part.
    - Each exam still takes five cycles to complete, but the throughput is much higher



Without Pipeline



With Pipeline

Prepared By: Eng. Randa Al_Dallah

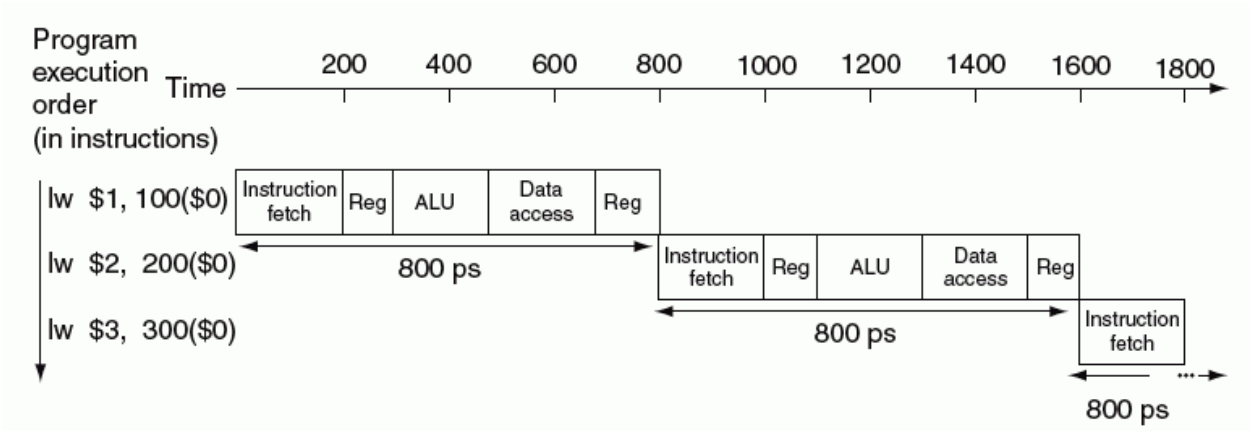
- The same pipelines apply to processors where we pipeline instruction execution.
- MIPS instructions take five steps:
  - Fetch instruction from memory.
  - Decoding the instruction and registers fetch.
  - Execute the operation, or calculate an address.
  - Access an operand in data memory.
  - Write the result into a register.
- The pipeline rate is limited by the slowest pipeline stage.

Single-Cycle versus Pipelined Performance

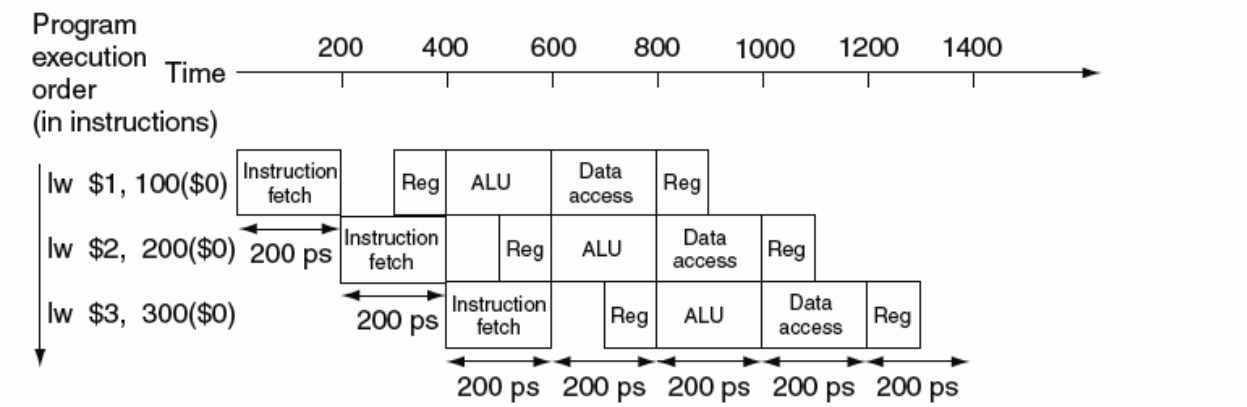
- In this chapter, we limit our attention to eight instructions: load word (lw), store word (sw), add (add), subtract (sub), and (and), or (or), set less than (slt), and branch on equal (beq).
- Example: Compare the average time between instructions of a single-cycle implementation, to a pipelined implementation.
  - The operation times (delay) for the major functional units are:
    - 200 ps for memory access
    - 200 ps for ALU operation
    - 100 ps for register file access

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 (`add`, `sub`, `and`, `or`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

  - In the following figure, we compare nonpipelined and pipelined execution of three load word instructions.



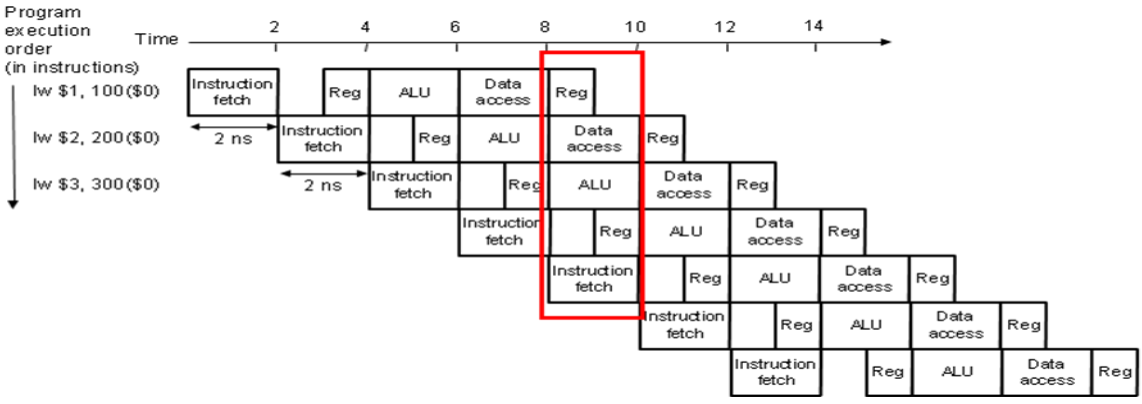
Prepared By: Eng. Randa Al_Dallah



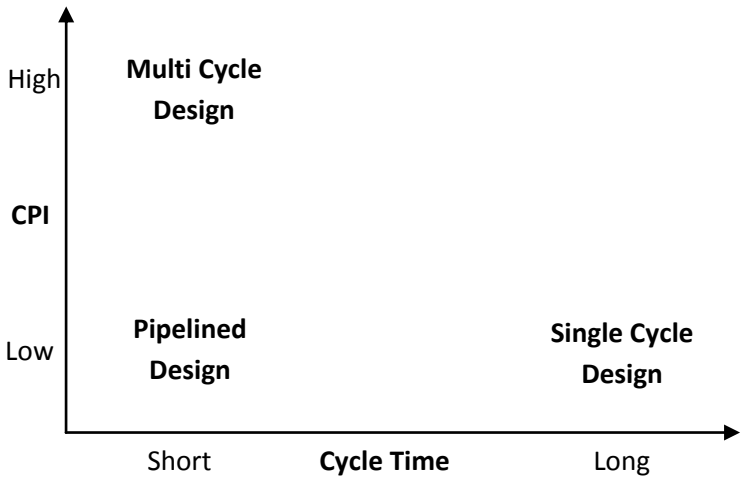
- The pipeline execution clock must have the worst-case clock cycle of 200 ps even though some stages take only 100ps.
- **Note:** we assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half.
- The execution time for the 3 instructions:
  - Without pipelining – the execution time = 3 x 800 = 2400ps
  - With pipelining – the execution time = 1400ps

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

- Potential speedup = number of pipeline stages
- Unbalanced lengths of pipeline stages can reduce speedup.
  - The time per instruction in the pipelined processor will exceed the minimum possible, and speedup will be less than the number of pipelined stages.
- Ideal conditions and large number of instructions
  - If there 1,000,003 instructions
  - Pipeline – 1000000 x 200 + 1400 = 200,001,400
  - Nonpipeline – 1,000,000 x 800 + 2400 = 800,002,400
  - Speedup Ratio ≈4, is closed to the ratio of times between instructions (800 ps / 200 ps)



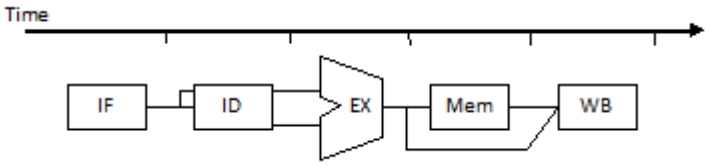
- Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction, but instruction throughput is the important metric because real programs execute billions of instructions.

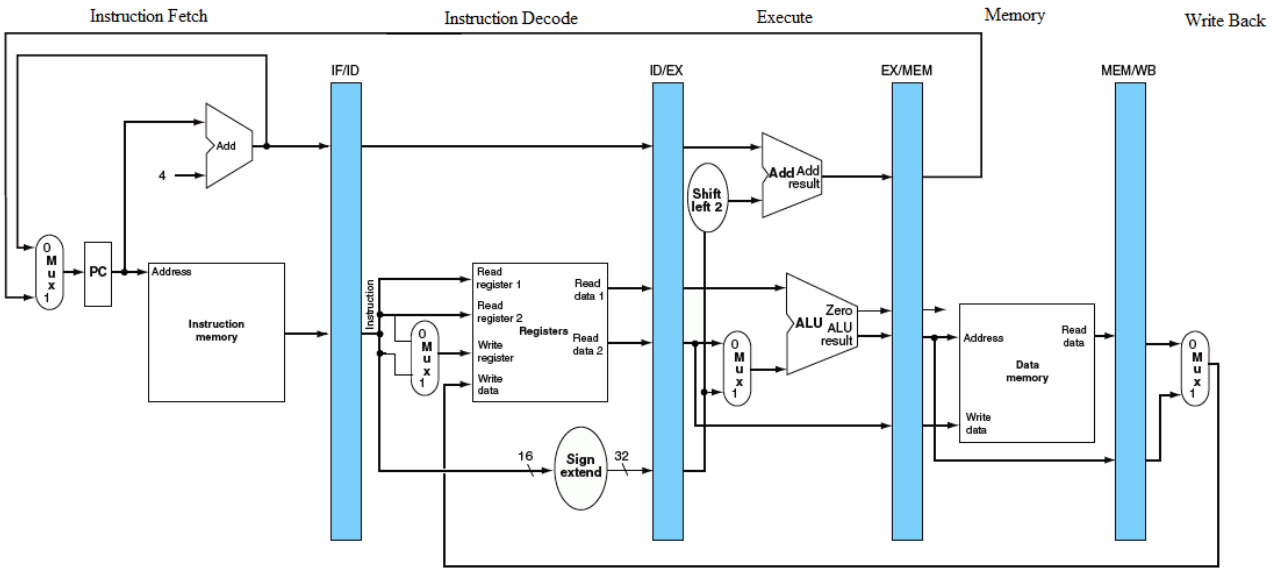


Designing (MIPS) Instruction Sets for Pipelining

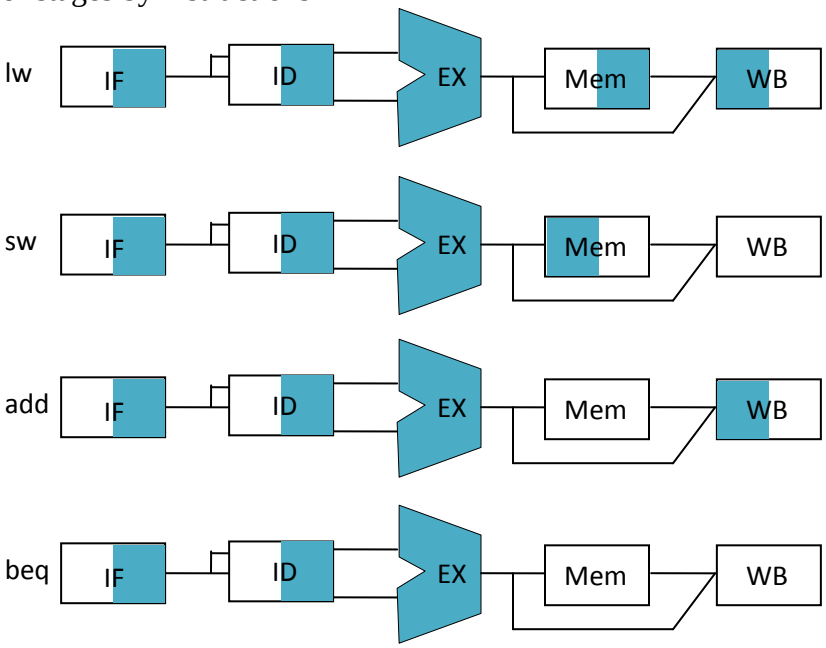
- What makes the design easy?
  - All MIPS instructions are of the same length.
    - It is easier to fetch instructions in the first pipeline stage and then decode them in the second stage.
    - In IA-32 instructions vary from 1-17 bytes, pipelining is considerably more challenging.
    - All recent IA-32 architectures translate instructions into microoperations, which are pipelined
  - MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction.
    - The second stage can begin reading the register file when the hardware is determining the type of instruction just fetched.
    - Without this symmetry we would need to split stage 2.
  - Memory operands only appear in load or store instructions in MIPS.
    - We can use the execution stage to calculate the memory address and then access memory in the following stage.
    - If we operate on operand in memory, like IS-32, stages 3 and 4 would expand to an address stage, memory stage and then execute stage

General Representation of the Instruction Pipeline





- Usage of stages by instructions

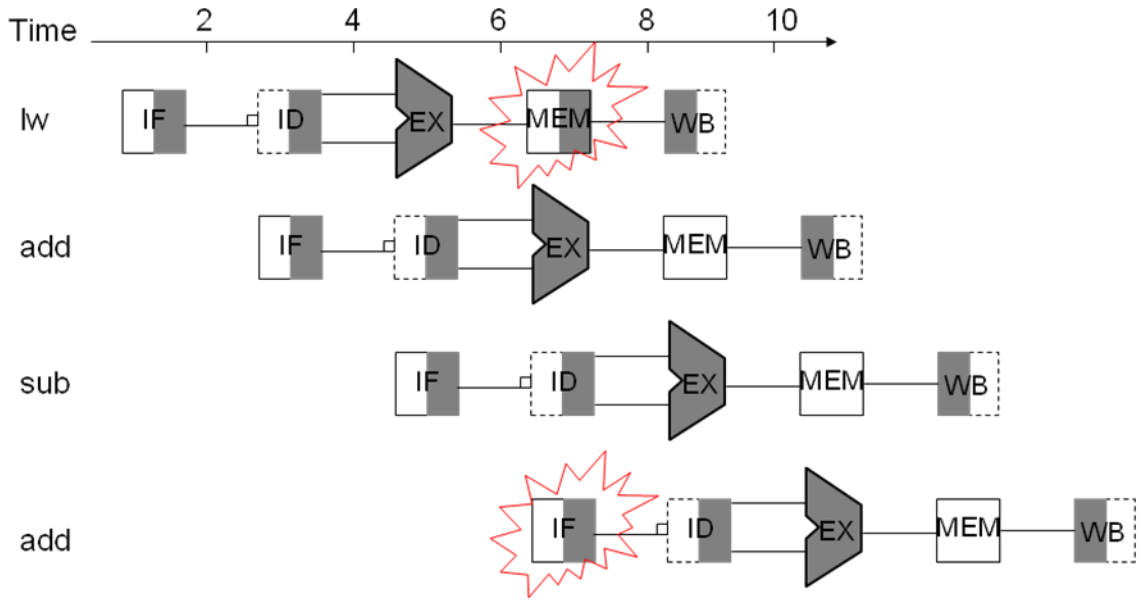


Pipeline Hazards

- Hazards are situations in pipelining when the next instruction cannot be executed in the following clock cycle.
- Hazards reduce the performance of pipelining.
- Types of pipeline hazards:
    - ***Structural hazards:*** Hardware cannot support the combination of instructions to execute in the same clock cycle.
    - ***Data hazards:*** An instruction depends on the results of a previous instruction still in the pipeline.
    - ***Control hazards:*** Which instruction to execute next depends on the results of a previous instruction still in the pipeline.

Structural Hazards

- MIPS instruction set was designed to be pipelined, making easy to avoid structural hazards
- No structural hazards in the present design
    - Separate instruction and data memories
        - Suppose we had a single memory instead of two memories
            - If the pipeline had four load word instructions, we should see that in the same cycle the first instruction is accessing data memory while the forth is fetching an instruction from the same memory.
            - Without two memories, our pipeline could have a structural hazard.



    - One instruction can read from RF while other can write into it in the same cycle.

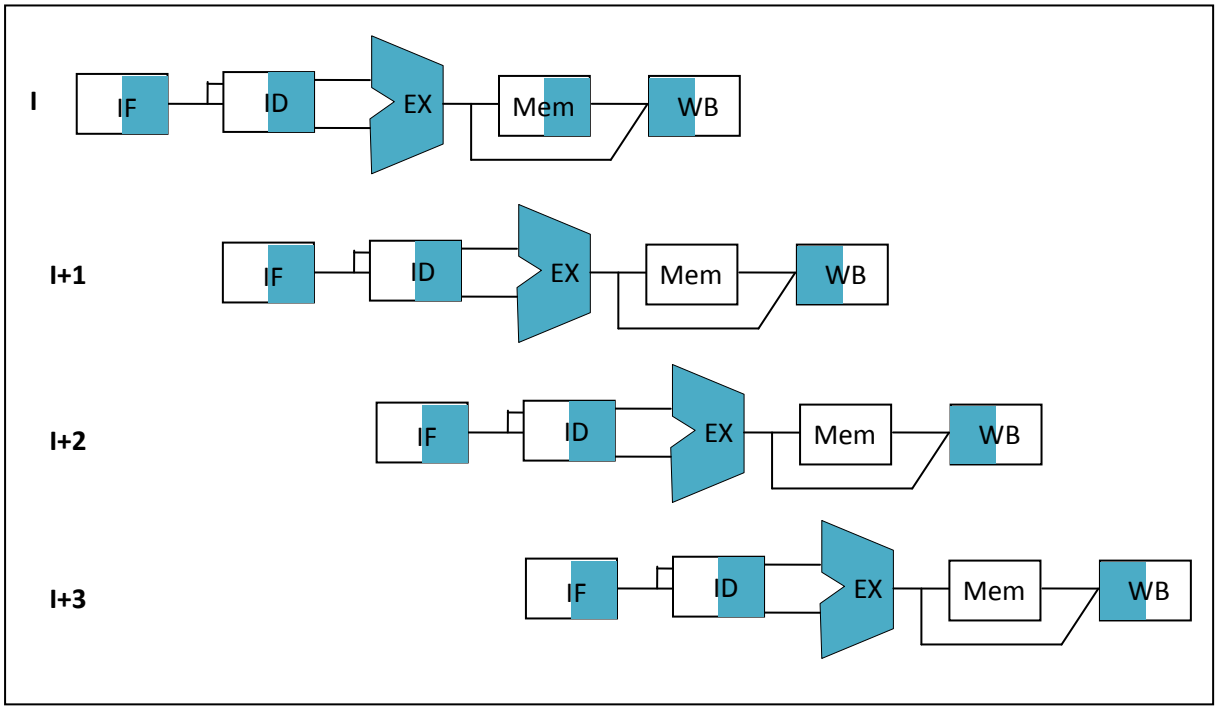
Data Hazards

- Data hazards occur when the pipeline must be stalled because one step must wait for another to complete (starting next instruction before first is finished).
- Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.
- Example:

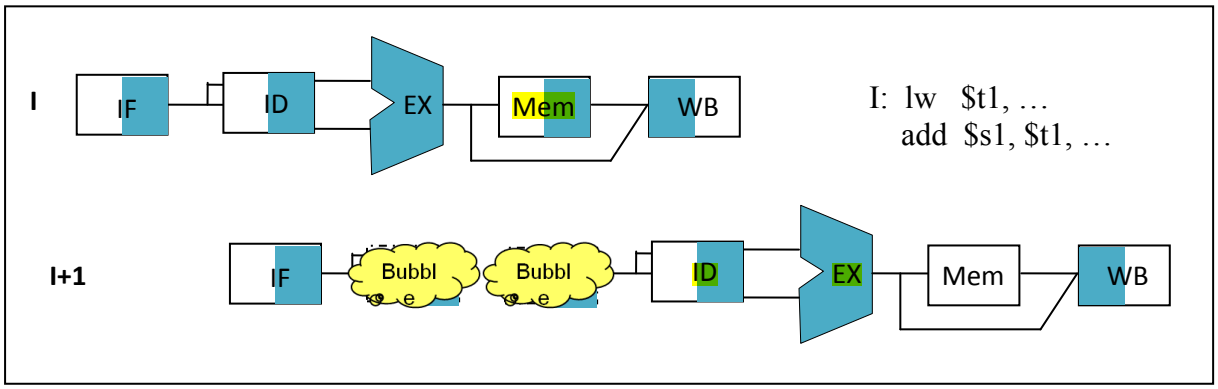
  add $s0, $t0, $t1
  sub $t2, $s0, $t3
- The add instruction doesn't write its result until the fifth stage

- There are many solutions:
  - Add bubbles (NOP) to the pipeline (by software or hardware).
  - Remove all such hazards by the compiler (Reorder code to avoid pipeline stalls).
  - The primary solution is based on forwarding or bypassing: adding extra hardware to retrieve the missing item earlier from the internal resources
- Example:

  I: lw $t1, …
  add $s1, $t1, …
  sub $s2, $t1, …
  slt $t2, $t1, …



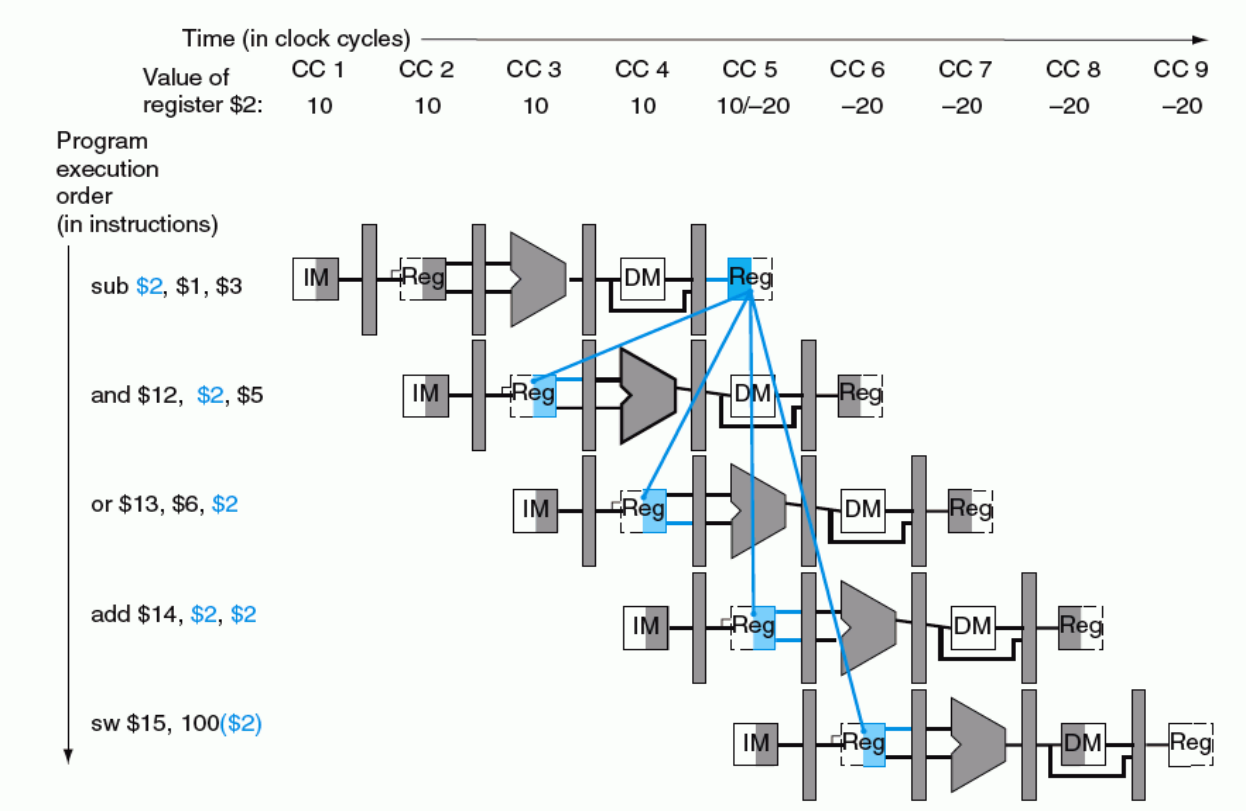
Stalls due to data hazards



- Example: Consider the following code, and determine the data hazards (data dependences) in the code

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

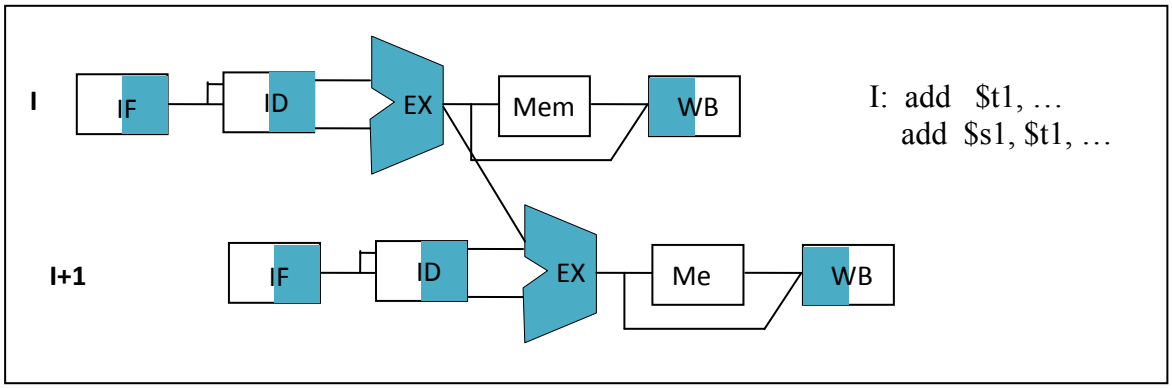
- The last four instructions are all dependent on the result in register $2 of the first instruction
- The following figure illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation.



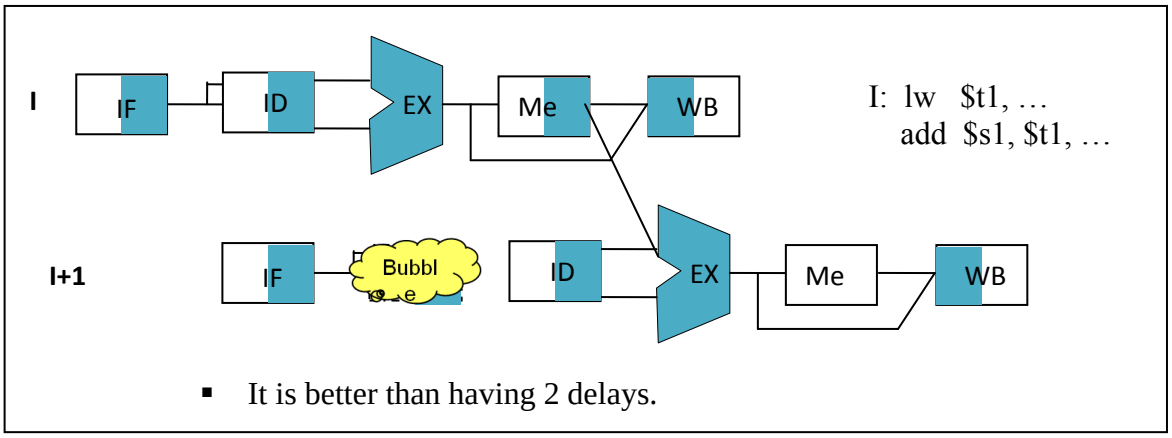
- In this situation, the instructions that would get the correct value are add *and sw*; the *and* and *or* instructions would get the incorrect value.
- The result of the sub instruction is available at the end of the EX stage or clock cycle 3.
  - When is the data actually needed by the *and* and *or* instructions?
  - At the beginning of the EX stage, or clock cycle 4 and 5, respectively.
- We can execute this segment without stalls if we simply forward the data as soon as it is available to any units that need it before it is available to read from the register file.

Data Forwarding

- **Data forwarding-1 (P1):** from EX/MEM (ALU out) to ALU in (operand 1 or operand 2)

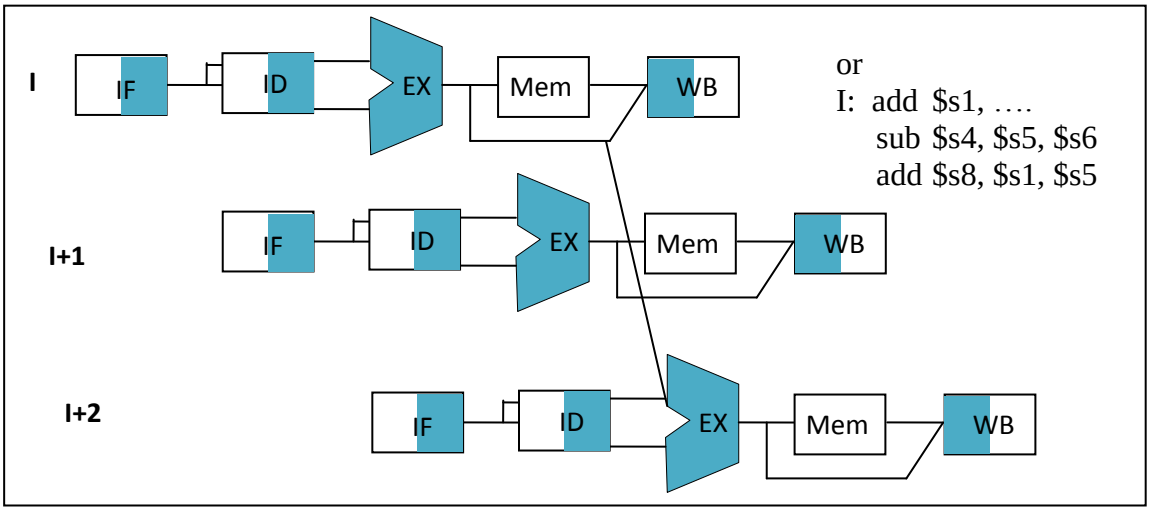


- **Data forwarding-2 (P2_A):** from MEM/WB (Memory) to ALU in (operand 1 or operand 2)

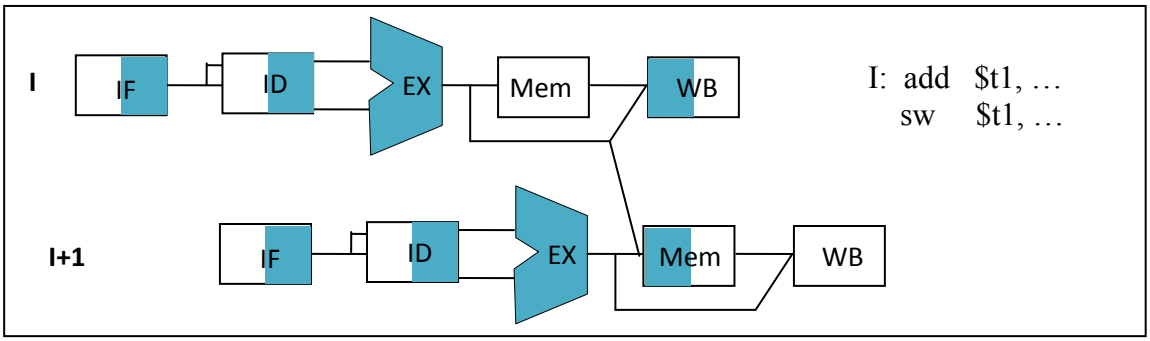


- It is better than having 2 delays.

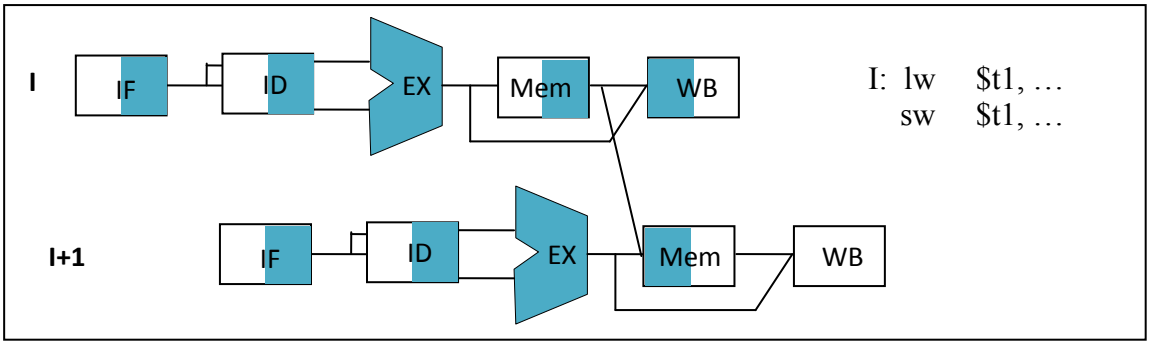
- **Data forwarding-2 (P2_B):** from MEM/WB (ALU out) to ALU in (operand 1 or operand 2)



- **Data forwarding-3 (P3_A):** from MEM/WB (ALU out) to Memory



- **Data forwarding-3 (P3_B):** from MEM/WB (Memory) to Memory



Examples: Data Dependences

- Consider the following code sequence:

  lw $s1, 0($s2)
  sub $s4, $s1, $s5
  and $s6, $s1, $s7
  or $s8, $s1, $s9

  - Observations:
    - Load instruction causes $s1 dependency
    - $s1 not available until 'MEM' stage (clock 4)
  - The following table represent the execution clock cycles for the previous code with forwarding

Instruction	Clock								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
I1	IF	ID	EX	MEM	WB				
I2		IF	**Stall**	ID	EX	MEM	WB		
I3				IF	ID	EX	MEM	WB	
I4					IF	ID	EX	MEM	WB

- For each code sequence below, find the dependences and state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding:

1. lw $s1, 45($s2)
   add $s5, $s6, $s7
   sub $s8, $s6, $s7
   or $s9, $s6, $s7

   - No dependence: No hazard possible because no dependence exists on $s1 in the immediately following three instructions.

2. lw $s1, 45($s2)
   add $s5, $s1, $s7
   sub $s8, $s6, $s7
   or $s9, $s6, $s7

   - Dependence requiring stall: stall the add (and sub and or) before the add begins EX

3. lw $s1, 45($s2)
   add $s5, $s6, $s7
   sub $s8, $s1, $s7
   or $s9, $s6, $s7
   - Dependence overcome by forwarding: forward result of load to ALU in time for sub to begin EX

4. lw $s1, 45($s2)
   add $s5, $s6, $s7
   sub $s8, $s6, $s7
   or $s9, $s1, $s7
   - Dependence with accesses in order: No action required because the read of $s1 by or occurs in the second half of the ID stage, while the write of the loaded data occurred in the first half

5. add $s1, $s2, $s3
   lw $s4, 0($s1)
   sw $s2, 12($s1)
   - There are dependences causes a hazard?

Reordering code to avoid pipeline stalls

- Consider the following MIPS code. Find the hazards in the code and solve the problem.

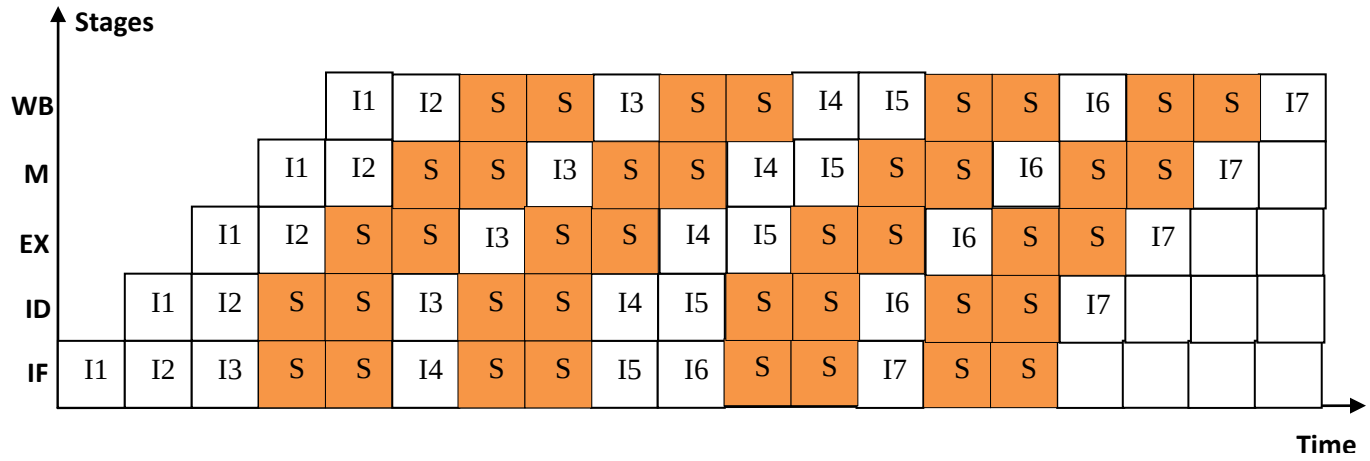
```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add   $t5, $t1, $t4
sw    $t5, 16($t0)
```


Prepared By: Eng. Randa Al_Dallah

- Solution1: Pipelined Processor without Forwarding (Stalls due to hazards) (diagram 1)

Inst	Clock																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
I1	IF	ID	EX	M	WB														
I2		IF	ID	EX	M	WB													
I3			IF	S	S	ID	EX	M	WB										
I4						IF	S	S	ID	EX	M	WB							
I5									IF	ID	EX	M	WB						
I6										IF	S	S	ID	EX	M	WB			
I7													IF	S	S	ID	EX	M	WB

- Or diagram2



Solution2: Add bubbles (NOP) to the code.

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
NOP
NOP
add  $t3, $t1, $t2
NOP
NOP
sw   $t3, 12($t0)
lw   $t4, 8($t0)
NOP
NOP
add  $t5, $t1, $t4
NOP
NOP
sw   $t5, 16($t0)
```

- Solution3: Pipelined Processor with Forwarding

Inst	Clock												
	1	2	3	4	5	6	7	8	9	10	11	12	13
I1	IF	ID	EX	Mem	WB								
I2		IF	ID	EX	Mem	WB							
I3			IF	**Stall**	ID	EX	Mem	WB					
I4					IF	ID	EX	Mem	WB				
I5						IF	ID	EX	Mem	WB			
I6							IF	**Stall**	ID	EX	Mem	WB	
I7									IF	ID	EX	Mem	WB

- Solution4: Reorder the instructions to avoid any pipeline stalls by moving up the third lw instruction to eliminates both hazards:

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
lw    $t4, 8($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
add   $t5, $t1, $t4
sw    $t5, 16($t0)
```

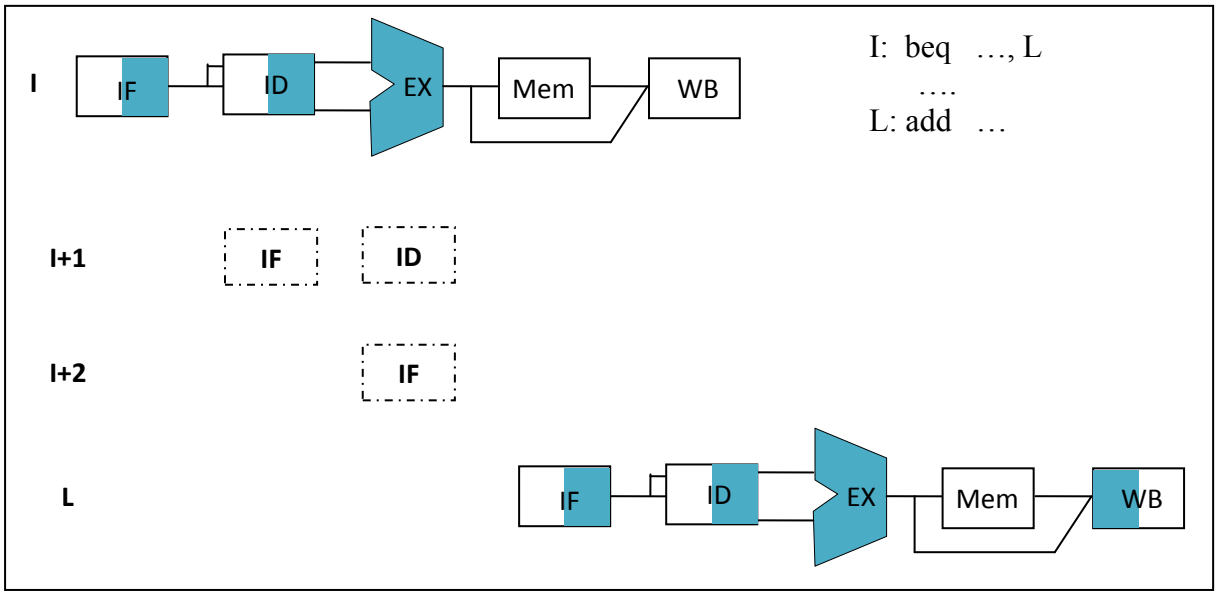
Pipelined Processor with Forwarding after the reorder

Instruction	Clock										
	1	2	3	4	5	6	7	8	9	10	11
I1	IF	ID	EX	Mem	WB						
I2		IF	ID	EX	Mem	WB					
I3			IF	ID	EX	Mem	WB				
I4				IF	ID	EX	Mem	WB			
I5					IF	ID	EX	Mem	WB		
I6						IF	ID	EX	Mem	WB	
I7							IF	ID	EX	Mem	WB

- On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

Control Hazards

- Control hazard also called branch hazard: 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; that is, the flow of instruction addresses is not what the pipeline expected.
- The first solution is stalling the pipeline.
    - Stall – wait until the pipeline determines the outcome of the branch



Actual address will be known after the third stage (depending on the test condition)

- There are two choices:
    - Freezing a pipeline until knowing the address of the next instruction.
        - Losing a few cycles
    - Continue with instruction sequence, then
        - Flushed out the instructions from the pipelined if the condition true (branching)

- The second solution is prediction.
    - One simple approach is to always predict that branches will be untaken. When you are right the pipeline proceeds at full speed. Only when branches are taken the pipeline stalls.
    - Branch prediction: In loops – branch backwards. It could be predicted taken for branches that jumps to an earlier address
    - Dynamic prediction: keep a history for branches – can reach 90% accuracy
    - Delayed Decision – used in MIPS: Continue with instructions not affected by the branch.

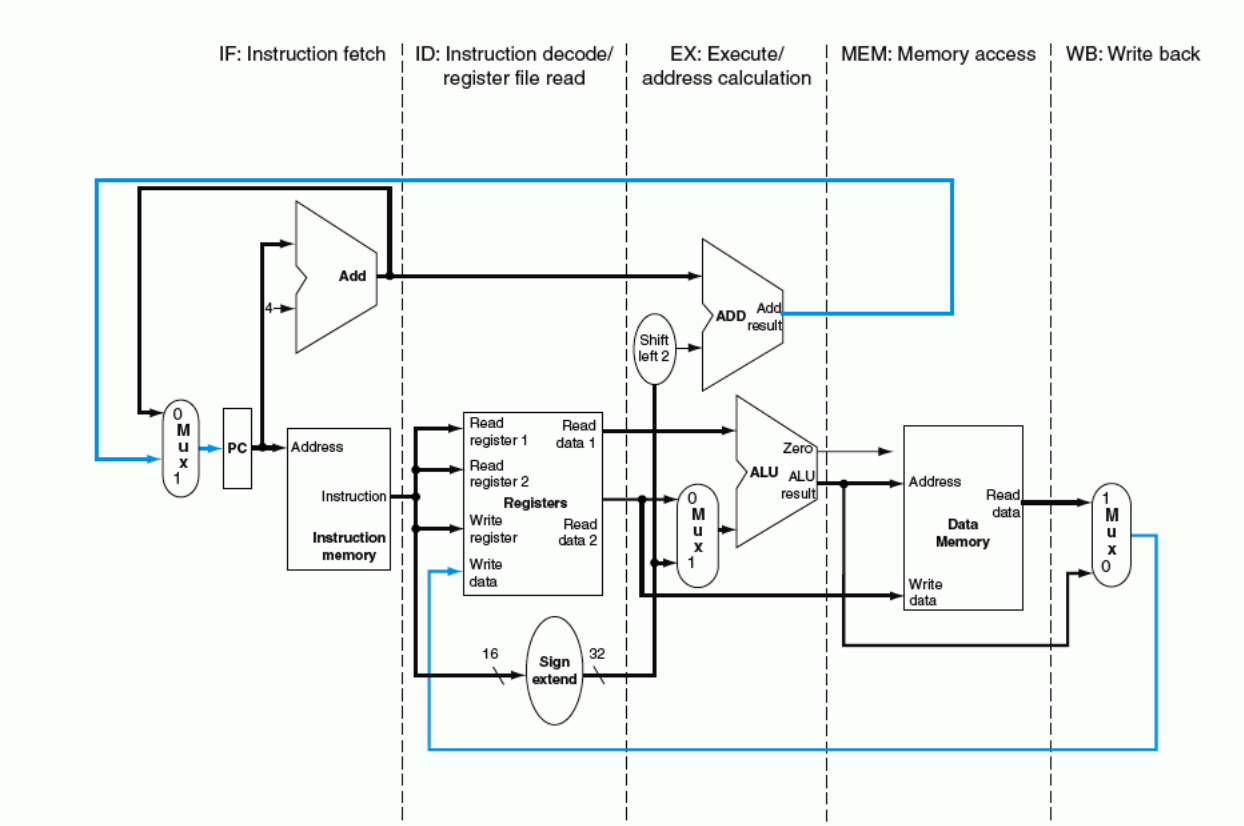
- Example:

```
        add     $s4, $s5, $s6
        beq     $s1, $s2, L1
        …..
L1:     or      $s7, $s8, $s9
```

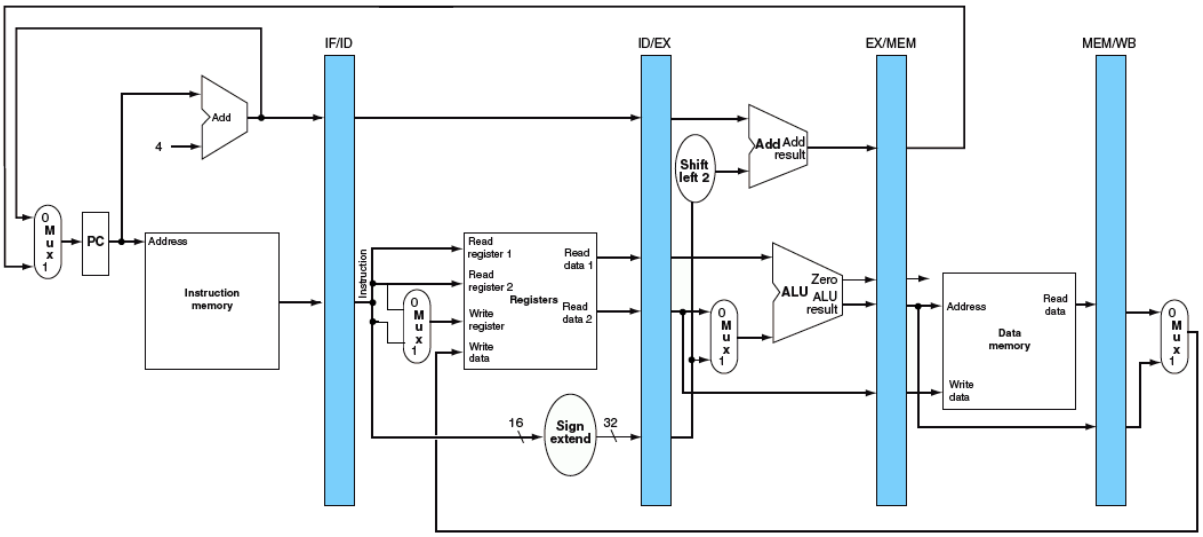
- The add instruction before the branch does not affect the branch and can be moved after the branch to fully hide the branch delay.

A pipelined Datapath

- Single-cycle data path



- Connect the PC + 4 back to the PC in the same stage, to get one instruction every cycle.
- Add registers to hold data to share multiple datapathes

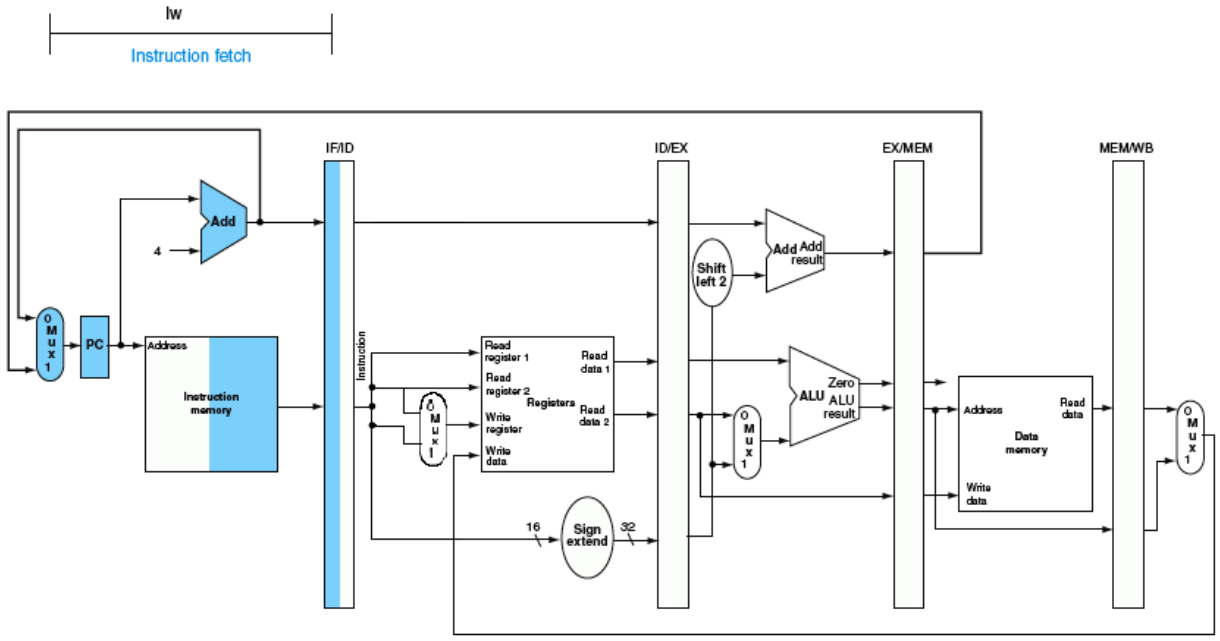


- No register after write-back stage
- The PC can be though as a pipeline register
    - It feeds the IF stage to the pipeline
    - Its content must be saved when an exception occurs

- Instructions and data move generally from left to right through the five stages
- There are two exceptions
    - The write-back stage, which does the results back into the register file
        - Could lead to data hazards
    - The selection of the next value of PC, choosing between the incremented PC and the branch address from the MEM state
        - Could lead to control hazards

Instruction Fetch

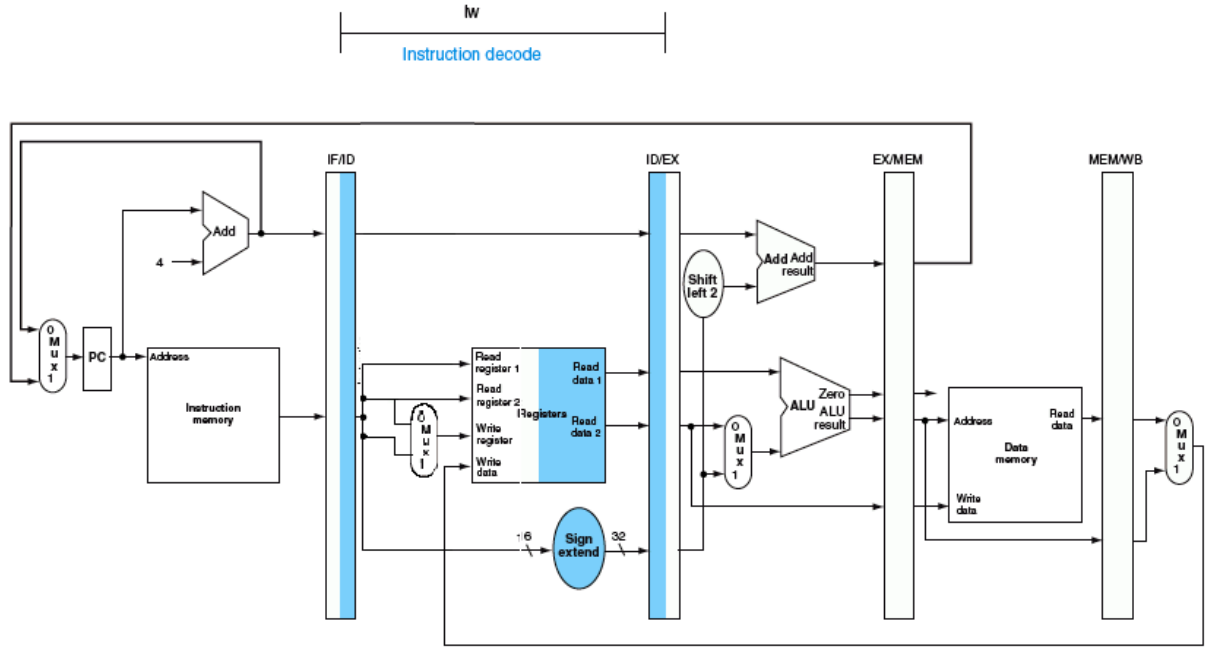
- The instruction is read from memory using the address in the PC and then placed to IF/ID register
    - IF/ID register is similar to IR in the multicycle databath.
- The PC is increased by 4 and written back to PC. Also saved in the IF/ID in case needed by a branch instruction (beq)



Instruction Fetch Stage for lw instruction

Instruction Decode and Register File Read

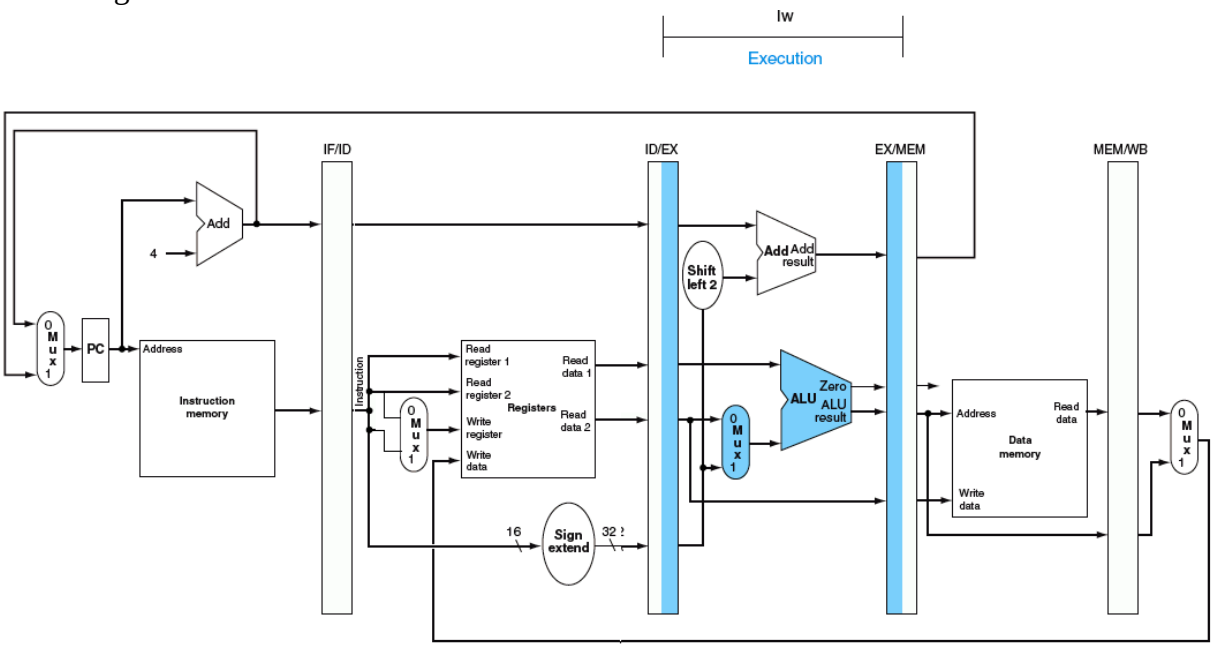
- IF/ID pipeline register supplies
  - The 16-bit immediate field, which is sign-extended to 32 bits
  - Register numbers to read the two registers
- All three values and the incremented PC address are stored in the ID/EX pipeline register



Instruction Decode Stage for lw instruction

Execute or address calculation

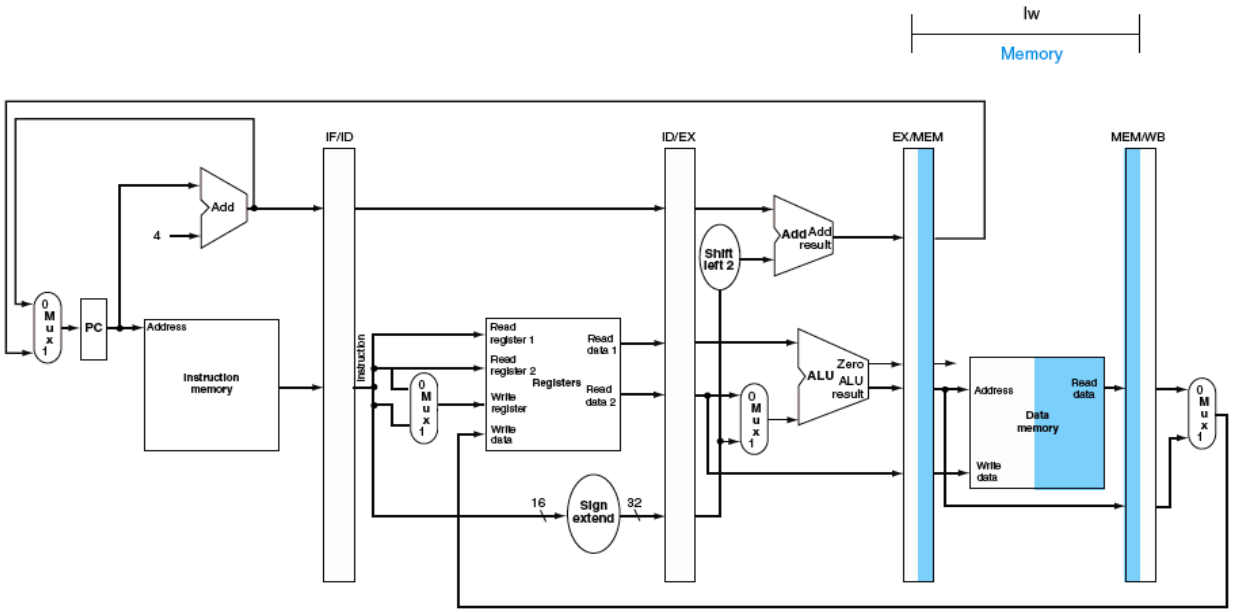
- The load instruction reads the contents of register 1 and the sign-extended immediate from ID/EX pipeline registers and adds them using ALU. The sum is placed in EX/MEM pipeline register.



Execute Stage for lw instruction

Memory Access

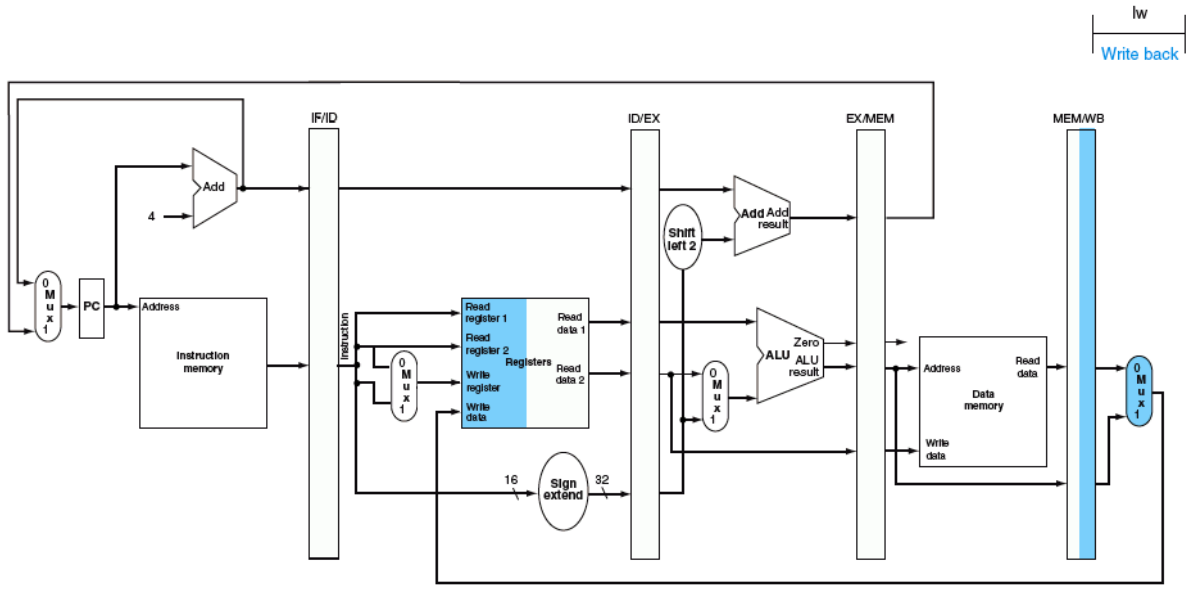
- The load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.



Memory Stage for lw instruction

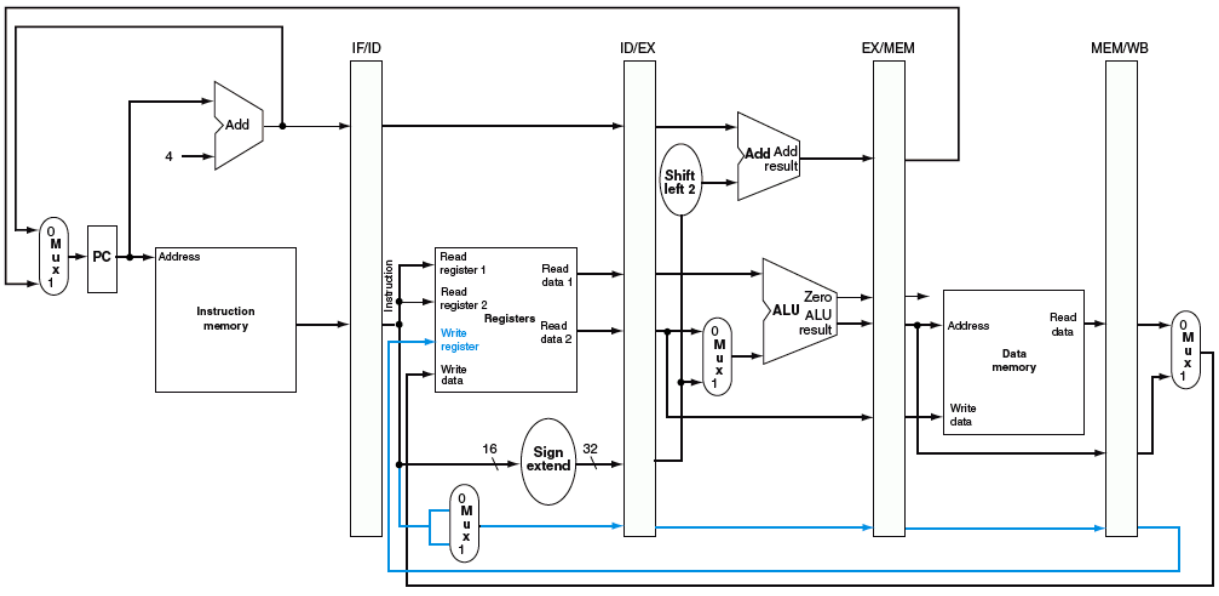
Write back

- Reading the data from MEM/WB pipeline register and writing it into the register file in the middle of the figure



Write back Stage for lw instruction

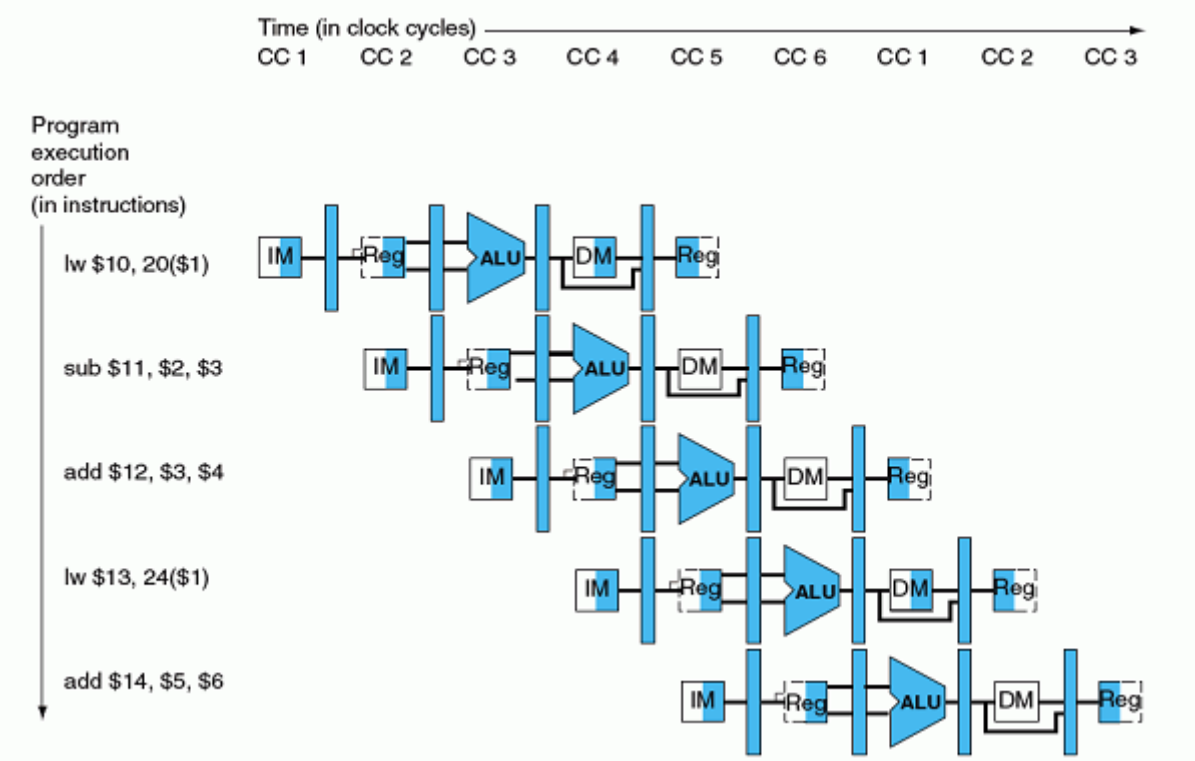
- There is a bug in this design (for load instruction):
  - The write back address
  - We need to preserve the destination register number in the load instruction.
- Sol: load must pass the register number from ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage.



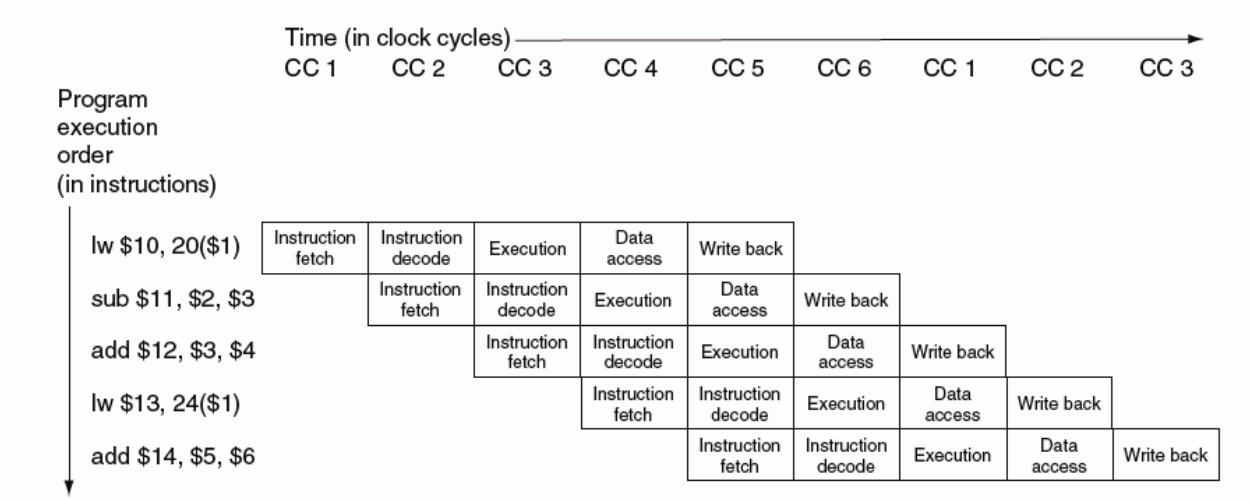
Graphically Representing Pipelines

- Example: Consider the following five instruction sequence:

```
lw    $10, 20($1)
sub   $11, $2, $3
add   $12, $3, $4
lw    $13, 24($1)
add   $14, $5, $6
```

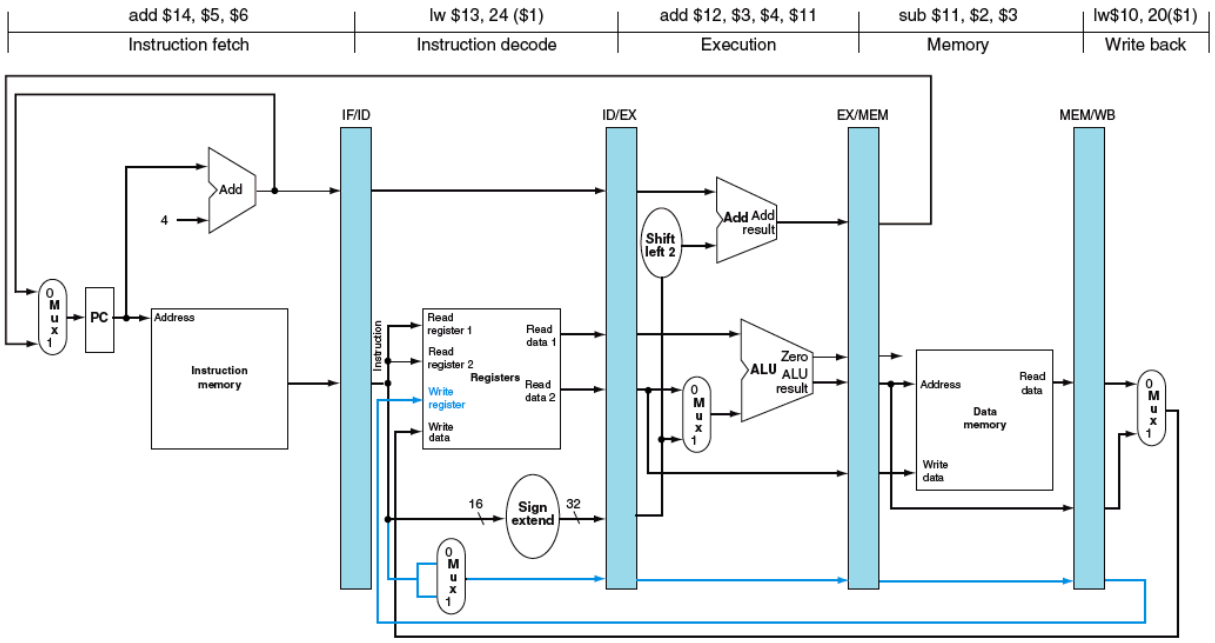


Multiple-clock-cycle pipeline diagram of five instructions



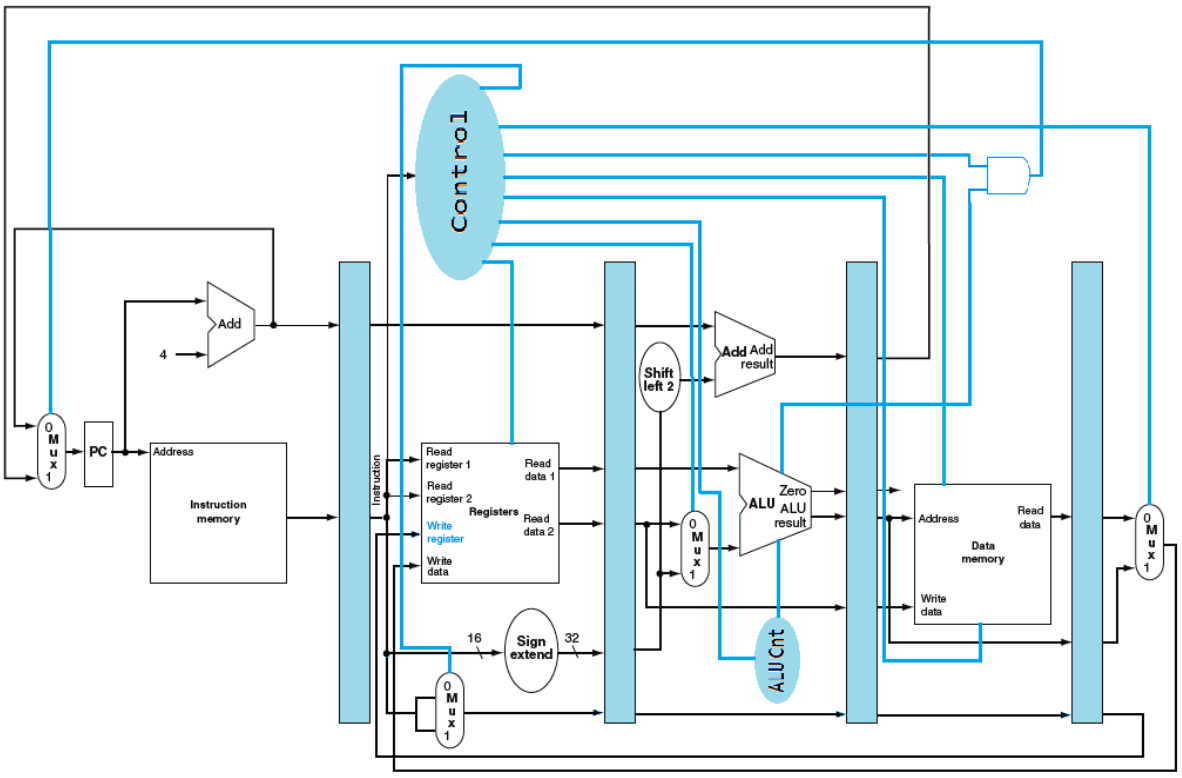
Traditional Multiple-clock-cycle pipeline diagram of five instructions

- The following figure shows the single-clock-cycle diagram corresponding to clock cycle 5 of the previous diagram
    - Single-clock cycle pipeline diagrams show the state of the entire datapath during a single clock cycle.



Pipeline Control

- The control design of the pipelined datapath is as simple as the control design of the single-cycle datapath.
    - We use the same ALU control logic, branch logic, destination-register-number multiplixer, and control lines.
- As for the single cycle implementation, we assume that the PC is written on each clock cycle, so there is no separate write signal for the PC
- There are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

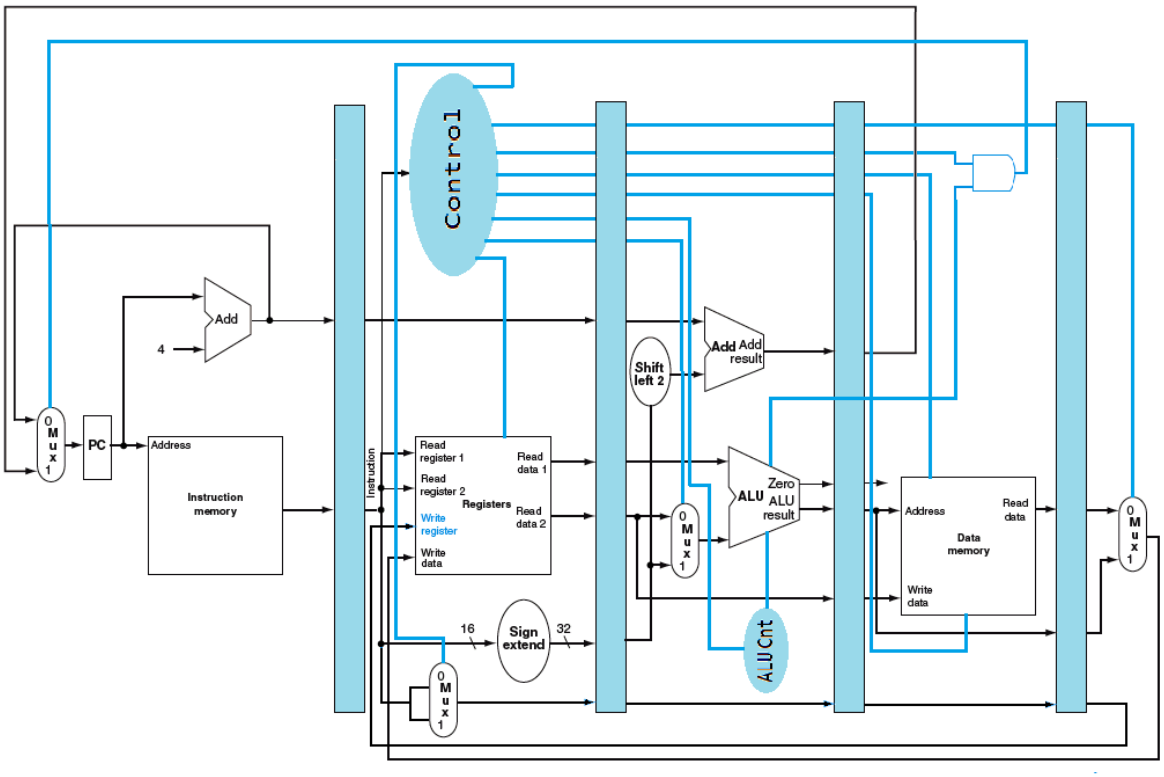


- Divide the control lines into five groups according to the pipeline stage

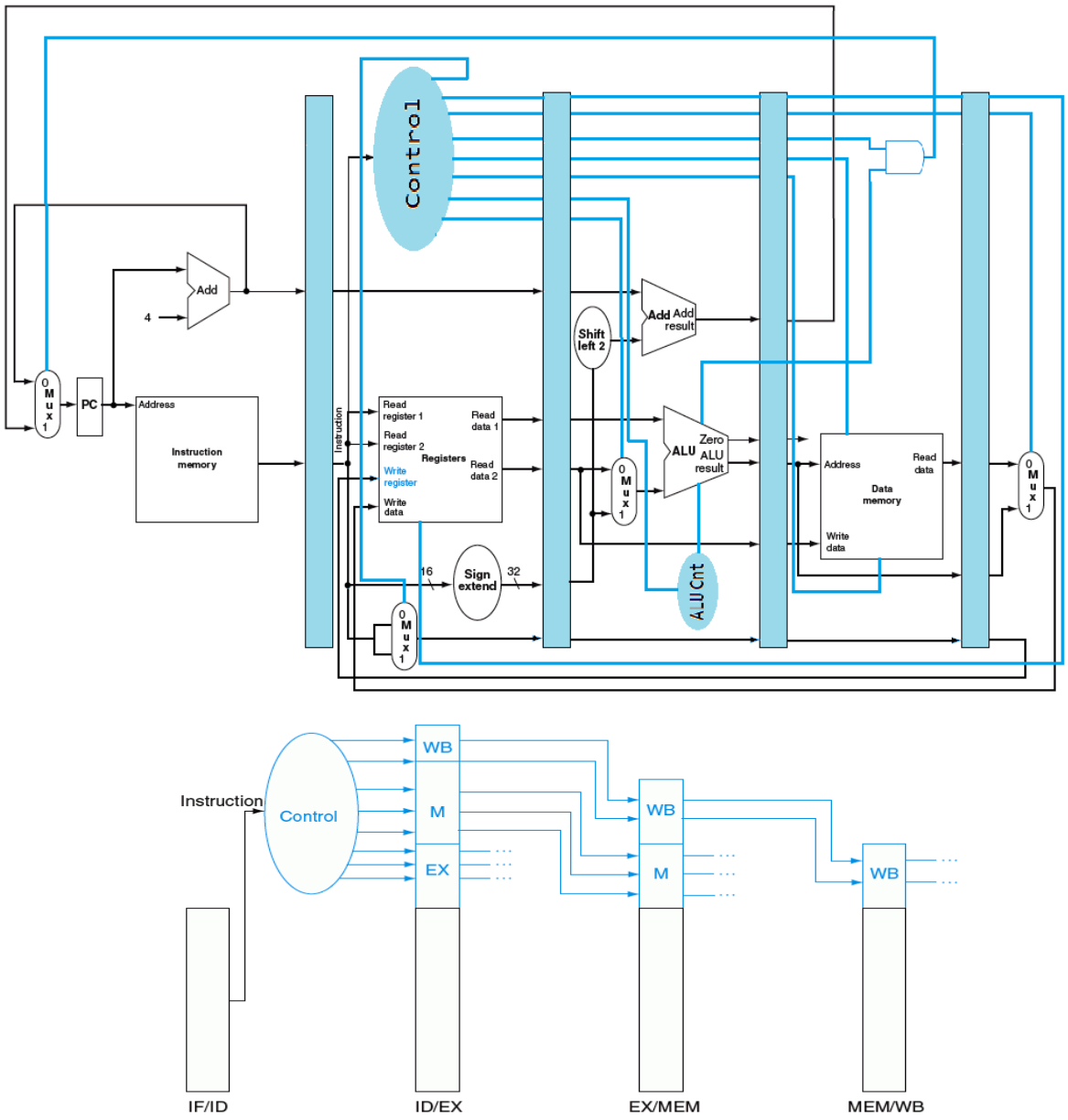
- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back

- Instruction fetch
  - The control signals to read instruction memory, and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
- Instruction decode/register file read
  - As in the previous stage, there are no optional control lines to set.
- Execution / address calculation
  - The signals to be asserted are ALUOp, and ALUSrc. The signals select the Result register, the ALU operation, and Read data or a sign-extended immediate for ALU
- Memory access
  - Control lines to be set by branch equal, load, and store: Branch, MemRead, and MemWrite. PCSrc selects the next sequential address unless control asserts Branch and the ALU result was zero

- Write back
    - The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register, and RegWrite, which writes the chosen value.



- The control design for single-cycle datapath is combinational circuit, while the control design for pipelined datapath is sequential circuit (using registers to hold and passing the signals).
- There are some correction in this control design (WB stage)
    - Correction for RF write signal



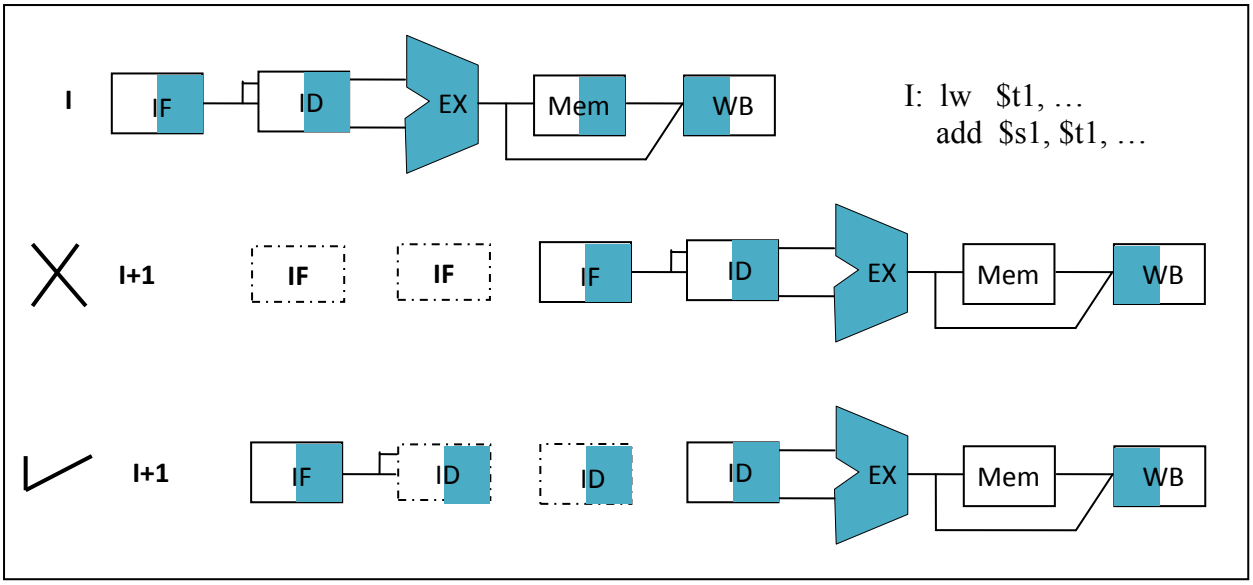
Control lines for the final 3 stages

Instruction	Execution/address calculation stage control lines			Memory access stage control lines			Write-back stage control lines	
	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	0	0	0	0	0	1	0
lw	0	0	1	0	1	0	1	1
sw	0	0	1	0	0	1	0	X
beq	0	1	0	1	0	0	0	X

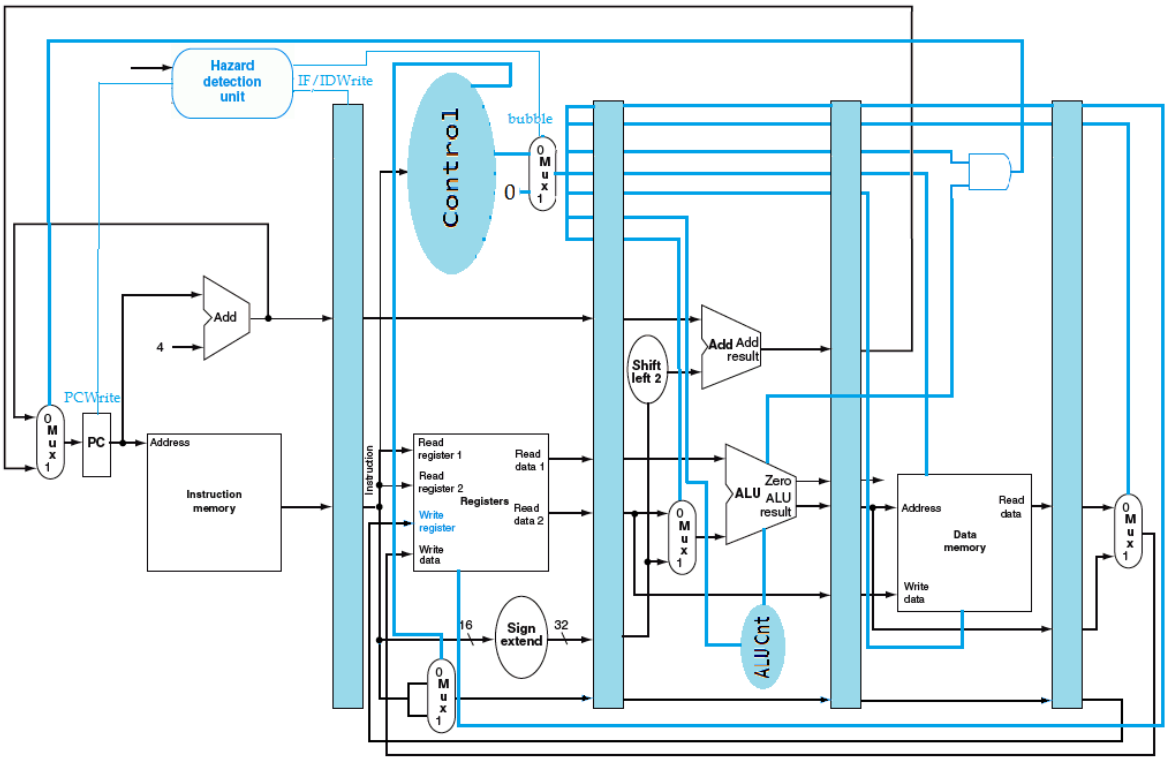
The values of the control lines, but they have been shuffled into 3 groups corresponding to the last 3 pipeline stages

Control for data hazard stalls

- Example for stalls due to data hazards



- The following figure shows the control for data hazard stalls



- To stall the pipeline datapath, set the following values of the hazard control signals:
  - Bubble = 1
  - PCWrite = 0

- IF/IDWrite = 0

Hazard Detection Unit

- It there are data dependences, stall the pipelined datapath.
  - Pipelined data path without forwarding;
- Condition to be checked:
  - Instruction in ID stage reads from a register in which instruction in EX stage or MEM stage is going to write
  - "ID/EX.RegisterRs" refers to the number of one register whose value is found in the pipeline register ID/EX.
  - The conditions are:

    Codition1:

    ID/EX.RegWrite and
    (IF/ID.RegisterRs = ID/EX.RegisterRd
    or IF/ID.RegisterRt = ID/EX.RegisterRd)

    Codition2:

    EX/MEM.RegWrite and
    (IF/ID.RegisterRs = EX/MEM.RegisterRd
    or IF/ID.RegisterRt = EX/MEM.RegisterRd)
  - Note: Rd is the RF destination address after multiplexing

How does Forwarding work?

- Types of Data Forwarding Paths (show the figures in page 8):
  - P1: from EX/MEM (ALU out) to ALU in (operand 1 or operand 2)

    add **$t1**, …..
    add $t2, **$t1**, ….
  - P2: from MEM/WB (Memory) to ALU in (operand 1 or operand 2)

    lw **$t1**, ….
    add $s0, **$t1**, …

    or from MEM/WB (ALU out) to ALU in (operand 1 or operand 2)

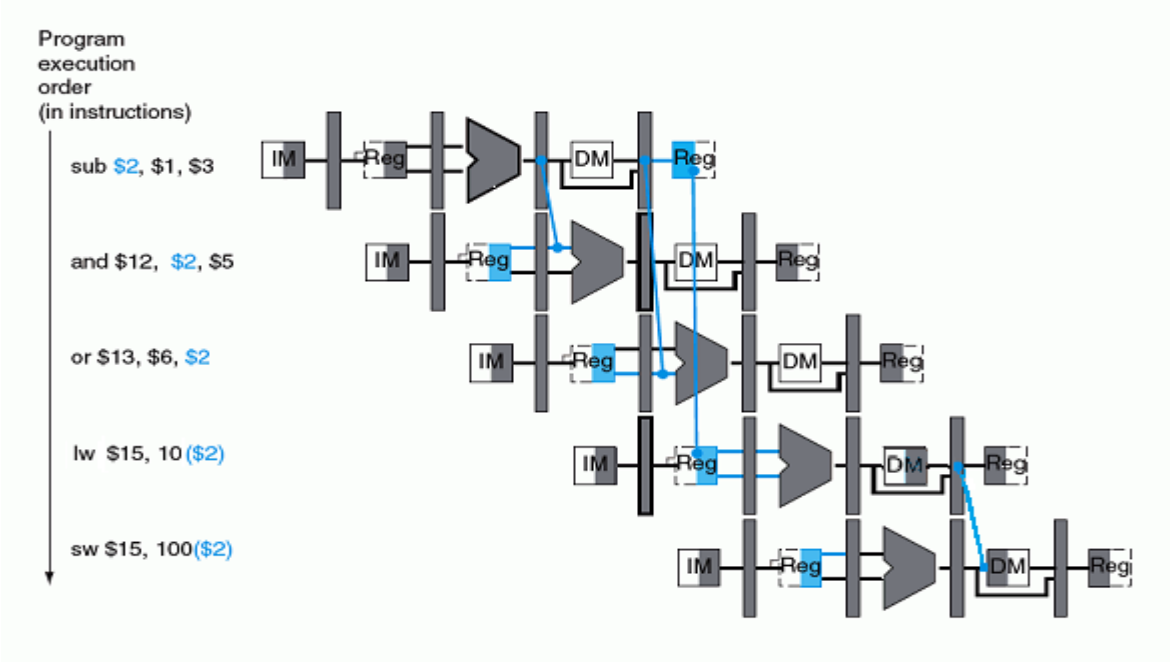
    add **$s1**, $s2, $s3
    sub $s4, $s5, $s6
    add $s8, **$s1**, $s5
  - P3: from MEM/WB (ALU out) to Memory

    add **$t1**, …..
    sw **$t1**, ….

    or from MEM/WB (Memory) to Memory

    lw **$t1**, ….
    sw **$t1**, ….

- Example: the following figure shows all possible forward paths



Hazard Conditions

- The hazard conditions are:

  P1:

  1a. EX/MEM.RegWrite and EX/MEM.RegisterRd = ID/EX.RegisterRs

  1b. EX/MEM.RegWrite and EX/MEM.RegisterRd = ID/EX.RegisterRt

  P2:

  2a. MEM/WB.RegWrite and MEM/WB.RegisterRd = ID/EX.RegisterRs

  2b. MEM/WB.RegWrite and MEM/WB.RegisterRd = ID/EX.RegisterRt

  P3:

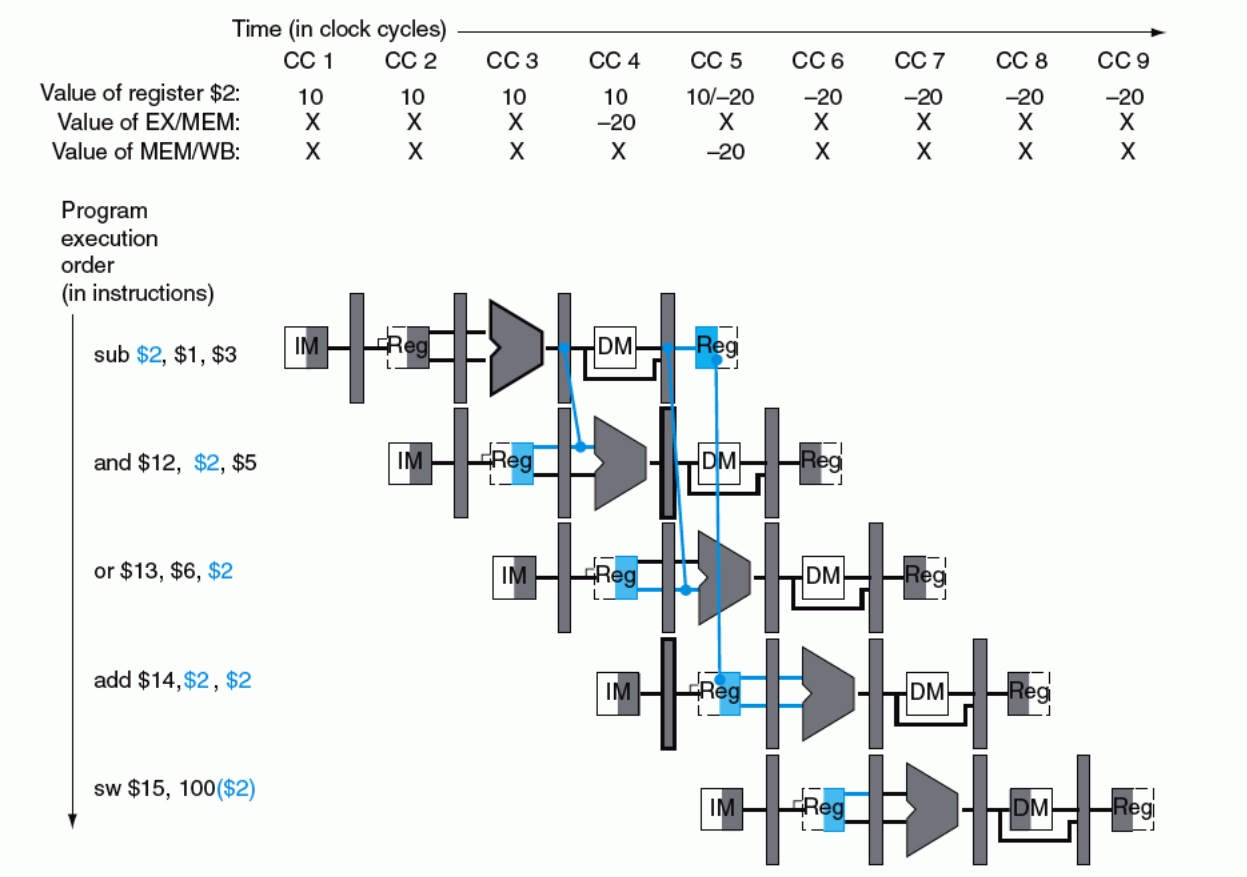
  1c. MEM/WB.RegWrite and MEM/WB.RegisterRd = EX/MEM.RegisterRt

- The hazards in this example are:

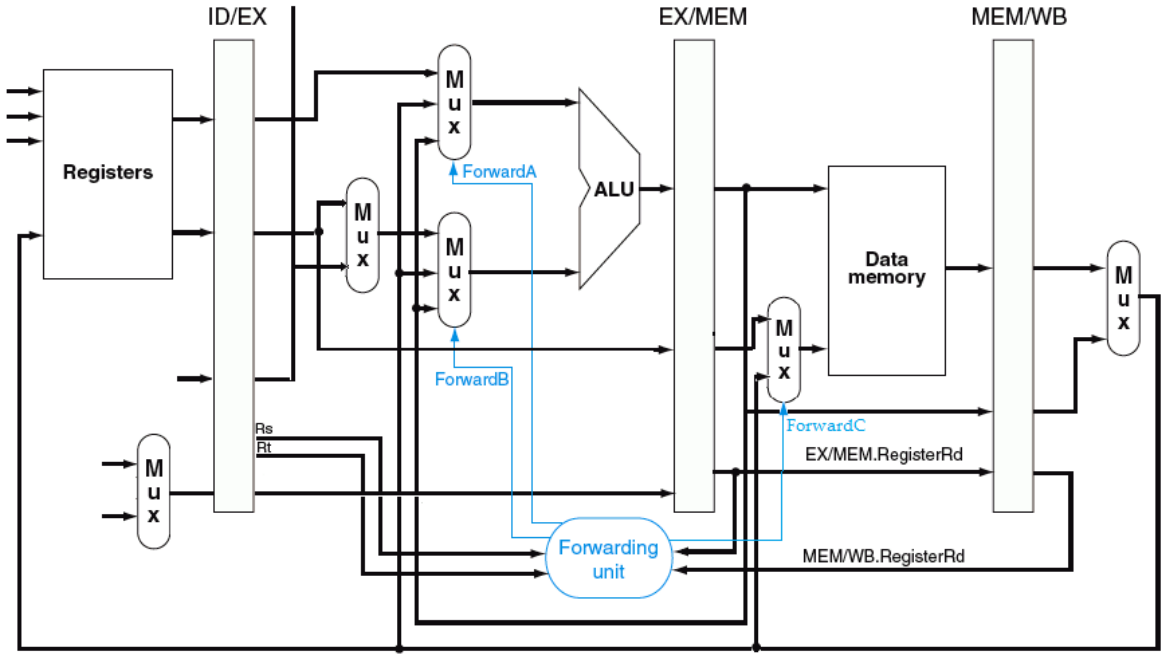
  sub **$2**, $1, $3
  and $12, **$2**, $5
  or $13, $6, **$2**
  add $14, **$2**, **$2**
  sw $15, 100(**$2**)

  - The first hazard between the result of sub $2, $1, $3 and the first read operand of and $12, $2, $5.
    - This hazard can be detected when the and instruction is in the EX stage and the prior instruction is in the MEM stage, so this hazard 1a: EX/MEM.RegWrite and EX/MEM.RegisterRd = ID/EX.RegisterRs
  - The second is sub-or is a type 2b hazard: MEM/WB.RegWrite and MEM/WB.RegisterRd = ID/EX.RegisterRt

- The following figure shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in previous example.



- The MIPS requires that every use of $0 as an operand must yield an operand value zero.
    - If the instruction in the pipeline has $0 as its destination (ex: sll $0, $1, 2), avoid forwarding.
    - Frees the assembly programmer and compiler of any requirement to avoid using $0 as a destination.
    - Add EX/MEM.RegisterRd ≠ 0 to the first hazard condition and MEM/WB.RegisterRd ≠ 0 to the second.
- Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.
    - Add more multiplexers



- The conditions for detecting hazards and the control signals to resolve them are:
  1. EX hazard:

     P1:

     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

  2. MEM hazard:

     P2:

     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

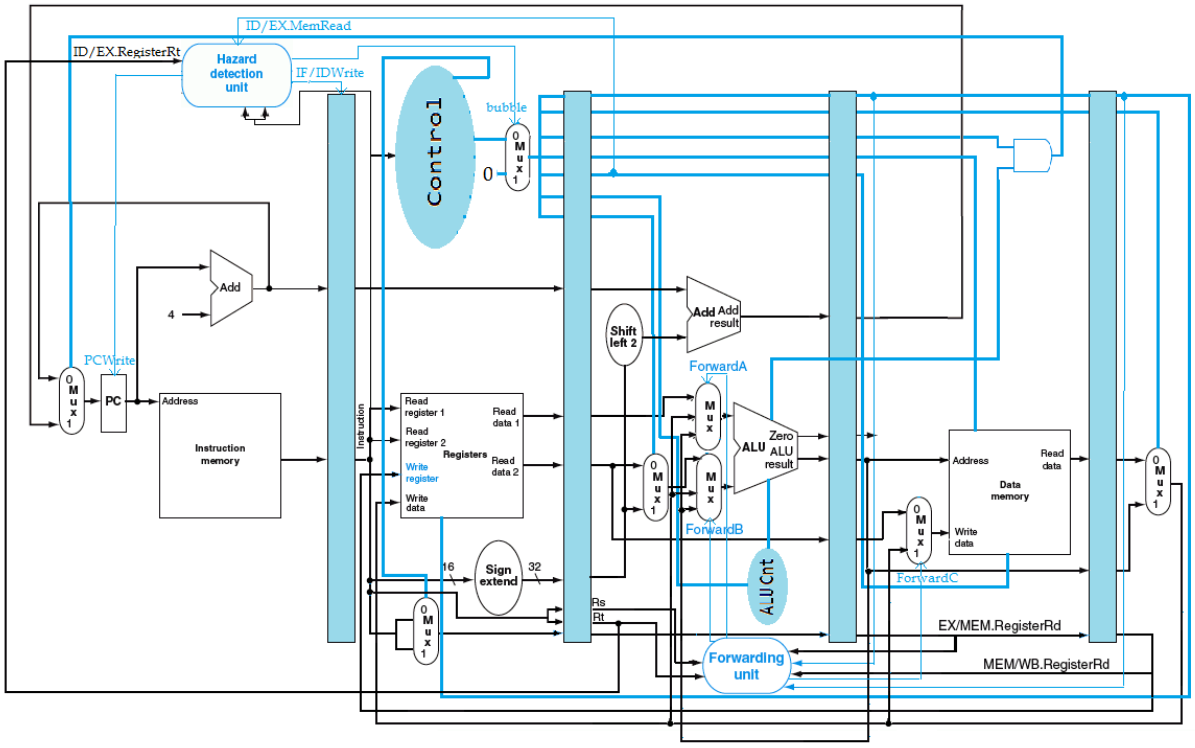
     P3:

     if (MEM/WB.RegWrite
     and (MEM/WB.RegisterRd ≠ 0)
     and (MEM/WB.RegisterRd = EX/MEM.RegisterRt)) ForwardC = 1

Stalling with Data Forwarding

- One case of data forwarding is still cause a hazard
  - An instruction tries to read a register following a load instruction that writes the same register (P2).
  - Thus, we need a hazard detection unit to "stall" the pipeline
- The hazard detection unit operates during ID stage so that it can insert the stall between the load and its use
- Checking for load instructions, the control for the hazard detection unit is this single condition:

If (ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt)))
Stall the Pipeline



Pipelined Datapath with Forwarding and Hazard Detection (for P2)