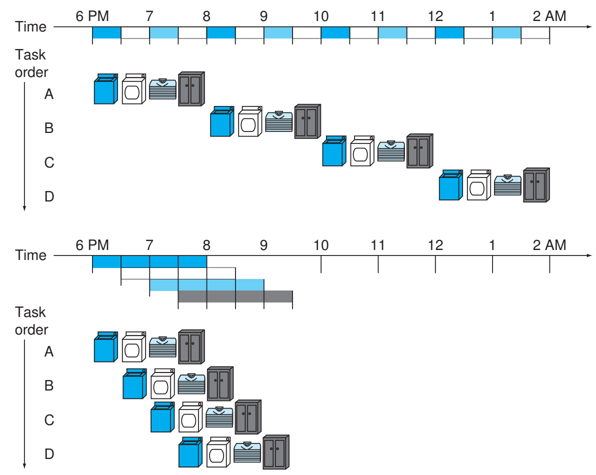
Concepts Introduced

- pipeline overview
- hazards
  - structural hazards
  - data hazards
  - control hazards
- pipeline datapath and control

The Laundry Analogy for Pipelining

- Multiple loads can be accomplished more quickly by pipelining the steps (washing, drying, folding, putting away).



Instruction Pipelining

- Pipelining is like an assembly line.
- Each step is called a pipe step (or stage) and is a machine cycle.
- Different steps from different instructions are processed in parallel.
- Pipelining is similar to the multicycle implementation, but instead of starting the next instruction after the last step of the current instruction, we overlap the steps.
- Pipelining improves throughput.

Pipeline Stages

- The stages described in the text are:
  - IF - Instruction Fetch
  - ID - Instruction Decode and register file read
  - EX - EXecution or address calculation
  - MEM - data MEMory access
  - WB - Write Back

Speedup from Pipelining

- Pipelining supports greater instruction throughput by allowing different parts of multiple instructions to be overlapped in execution.
- The ideal speedup would be the number of stages in the pipeline.

$$time\ between\ instructions_{pipelined} = \frac{time\ between\ instructions_{nonpipelined}}{number\ of\ pipe\ stages}$$

- There are several factors that prevent ideal speedup.
  - Stages may be imperfectly balanced.
  - Storing and retrieving information between pipeline stages requires overhead.
  - Hazards can prevent instructions from correctly completing a pipeline stage.

Pipeline Stages in More Detail

- IF (Instruction Fetch): fetches the instruction from the instruction cache and increments the PC.
- ID (Instruction Decode):
  - Decode the instruction.
  - Reads two values from the register file.
  - Sign extends the immediate value.
  - Calculates the PC-relative target address of a branch and checks if the branch should be taken.

Pipeline Stages in More Detail (cont.)

- EX (Execution/Effective Address):
  - Calculates an effective address for accessing memory.
  - Performs an arithmetic/logical operation on the two register values.
  - Performs an arithmetic/logical operation on a register value and the sign extended immediate value.
- MEM (Memory Access): loads a value from or stores a value into the data cache.
- WB (Write Back): updates the register file with the result of an operation or a load.

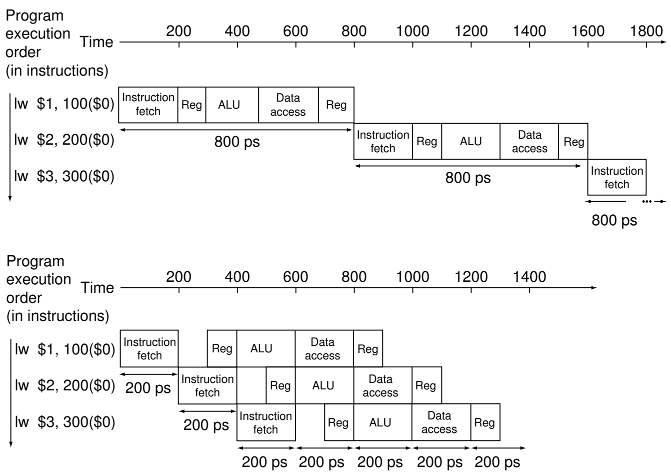
Total Time for Instructions Calculated for Each Component

- Some instruction stages require less time than others.
- Some instructions require more stages than other instructions.

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

Single-Cycle Execution versus Pipelined Execution

- There is a fourfold speedup on average between instructions (800ps single cycle on top to 200ps pipelined on bottom).



MIPS Is Designed for Pipelining

- All MIPS instructions are the same length (4 bytes).
- There are very few MIPS instruction formats (3 general formats).
- Memory access only occurs in load and store instructions.
- Accesses to memory must be aligned.

Pipeline Terms

- dependencies - relationships between instructions that prevent one instruction from being moved past another
- pipeline hazards - a situation when the current instruction cannot execute correctly in the next cycle without some type of resolution
  - structural
  - data
  - control
- pipeline stalls - a technique to resolve pipeline hazards by preventing some instructions from moving forward in the pipeline until the hazard no longer exists

Pipeline Diagram

- A pipeline diagram shows for a sequence of instructions when each instruction enters each stage of the pipeline.

cycle	1	2	3	4	5	6	7	8
inst 1	IF	ID	EX	MEM	WB			
inst 2		IF	ID	EX	MEM	WB		
inst 3			IF	ID	EX	MEM	WB	
inst 4				IF	ID	EX	MEM	WB

Structural Hazards

- A structural hazard occurs when the hardware cannot support a particular combination of instructions to be executed in the same cycle.
- One example is having a single memory for both instructions and data.

cycle	1	2	3	4	5	6	7	8
inst 1	IF	ID	EX	**MEM**	WB			
inst 2		IF	ID	EX	MEM	WB		
inst 3			IF	ID	EX	MEM	WB	
inst 4				**IF**	ID	EX	MEM	WB

Structural Hazards (cont.)

- Why not design the hardware to always avoid structural hazards?
  - Some hazards don't occur that often, so the cost may outweigh the benefit.
  - More complicated hardware that isn't used very often may impact performance.

Data Hazards

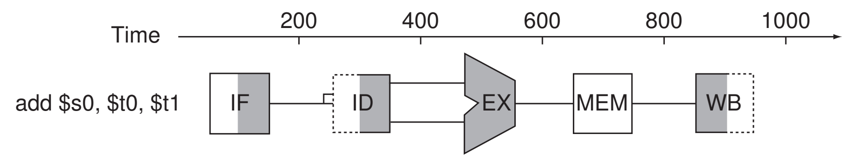
- A data hazard occurs because one instruction depends on the result of a previous instruction in the pipeline.

cycle	1	2	3	4	5	6	7	8	9
add **\$s0**,\$t0,\$t1	IF	ID	EX	MEM	WB				
sub \$t2,**\$s0**,\$t3		IF	ID	stall	stall	ID	EX	MEM	WB

- Can sometimes resolve (or decrease) stalls for data hazards.
  - forwarding
  - instruction scheduling

Graphical Representation of an Instruction Pipeline

- This figure conveys similar information as a conventional pipeline diagram, but with a graphical representation of each pipeline stage.



Dependences

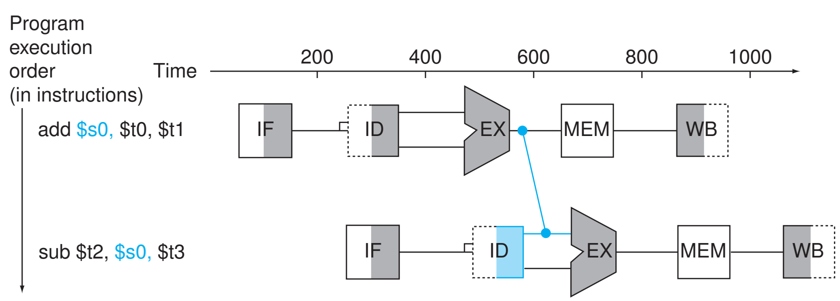
- dependences
  - Constrain the order in which results must be calculated.
  - Indicate the possibility of hazards.
  - Set a limit on the amount of parallelism that can be exploited.
- types of dependences
  - data (true) dependences
  - name (false) dependences
  - control dependences

Data Hazards

- types
  - RAW (read after write) - most common type of hazard
  - WAW (write after write) - Cannot occur in the MIPS integer pipeline since all instructions require the same number of stages and writes to memory occur in the MEM stage and writes to registers occur in the WB stage.
  - WAR (write after read) - Cannot occur in the MIPS integer pipeline because memory reads and writes both occur in the MEM stage and register reads occur early in the ID stage and register writes occur later in the WB stage.
- In the integer pipeline that is presented in the text, only loads can cause RAW stalls.

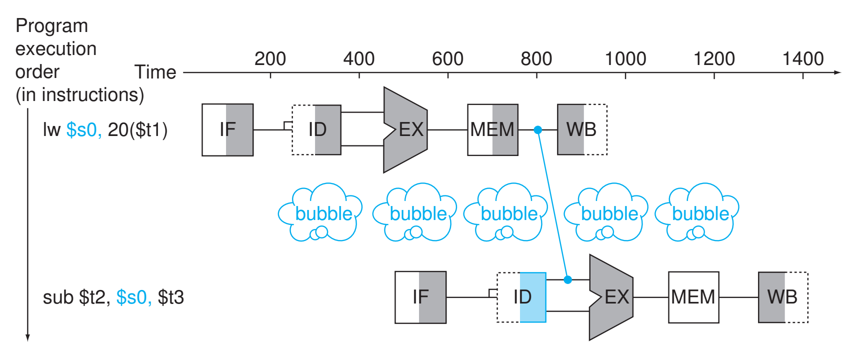
Resolving Data Hazards with Forwarding

- Data values can be forwarded from internal pipeline state registers (instead of the register file) when they are available.



Resolving Data Hazards with Stalls

- Sometimes forwarding cannot resolve a data hazards, such as a load followed by an R-format instruction that references the loaded register.
- A pipeline stall or bubble can be inserted into the pipeline.



Stall Shown in a Traditional Pipeline Diagram

- If one instruction is stalled, then all instructions that have entered the pipeline later are also stalled.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t4)	IF	ID	EX	MEM	WB								
lw $t2,4($t4)		IF	ID	EX	MEM	WB							
add $t3,$t1,$t2			IF	ID	stall	EX	MEM	WB					
sw $t3,12($t0)				IF	stall	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$t1,$t4							IF	ID	stall	EX	MEM	WB	
sw $t5,16($t0)								IF	stall	ID	EX	MEM	WB

Instruction Scheduling

- Reordering instructions can sometimes avoid stalls due to data hazards.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t4)	IF	ID	EX	MEM	WB								
lw $t2,4($t4)		IF	ID	EX	MEM	WB							
add $t3,$t1,$t2			IF	ID	stall	EX	MEM	WB					
sw $t3,12($t0)				IF	stall	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$t1,$t4							IF	ID	stall	EX	MEM	WB	
sw $t5,16($t0)								IF	stall	ID	EX	MEM	WB

=>

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t4)	IF	ID	EX	MEM	WB								
lw $t2,4($t4)		IF	ID	EX	MEM	WB							
lw $t4,8($t0)			IF	ID	EX	MEM	WB						
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
add $t5,$t1,$t4						IF	ID	EX	MEM	WB			
sw $t5,16($t0)							IF	ID	EX	MEM	WB		

An Example Pipeline Diagram

- For the following example, fill in when each instruction goes through each stage of the pipeline.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $3,0($5)													
add $7,$7,$3													
lw $4,4($5)													
sw $7,8($4)													
lw $5,0($4)													
add $10,$7,$8													
sub $10,$10,$5													

Control Dependences

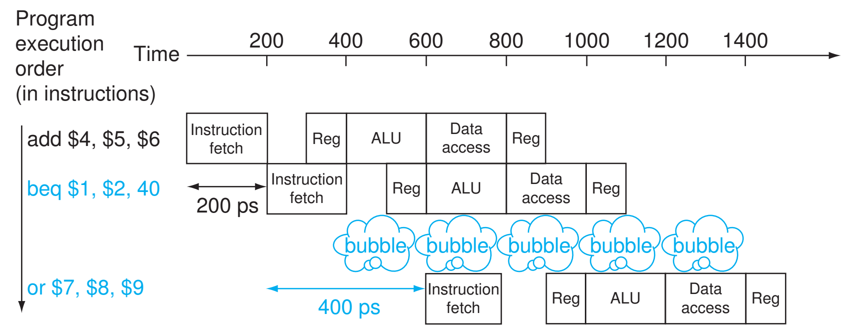
- An instruction is control dependent on a branch instruction if the instruction will only be executed when the branch has a specific result.
- An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
- An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

Control Hazards

- A control hazard occurs because the CPU does not know soon enough
  - whether or not the conditional branch will be taken
  - the target address of the transfer of control
- solutions
  - Stall until the needed information is available.
  - Predict whether or not the branch will be taken.
  - Delay the branch execution until the branch decision and branch target address are available.

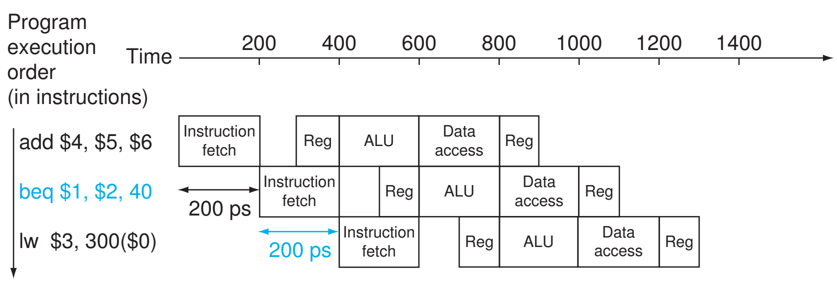
Resolving Control Hazards by Stalling on Every Branch

- One solution for control hazards is to stall on every conditional branch, which can affect performance.



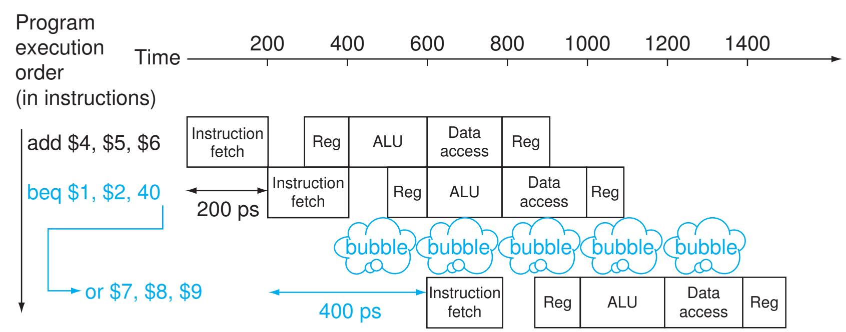
Resolving Control Hazards by Predicting Not Taken

- Another solution for control hazards is to predict every conditional branch to be not taken.
- The figure below shows there is no delay when the branch is not taken.



Resolving Control Hazards by Predicting Not Taken (cont.)

- The figure below shows there is a one cycle delay when the conditional branch is taken.



Branch Prediction by the Compiler

- Sometimes the compiler can perform analysis to exploit hardware support for branch prediction, which may be in the form of a likely bit that is part of the branch instruction.
- What is the likely behavior of the three branches in the code segment below?

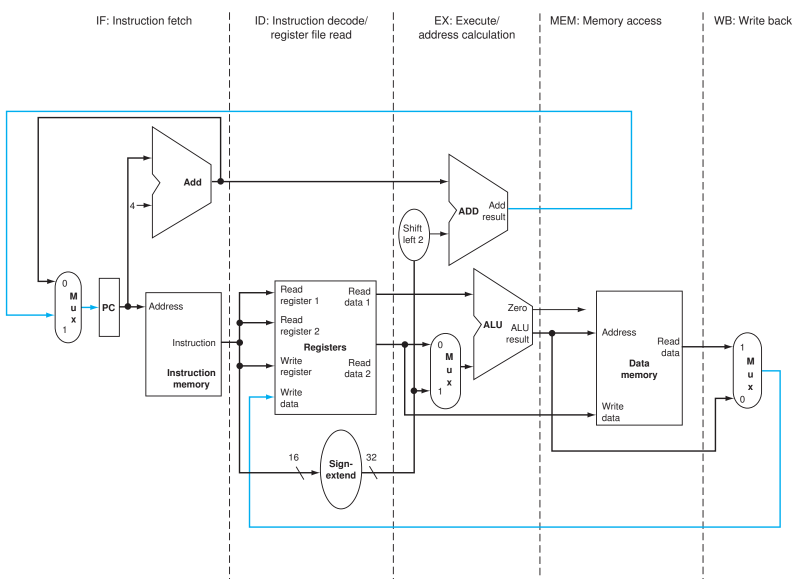
```
L1: ...
    beq $3,$2,L3    # fall thru or branch?
    ...
    bne $4,$5,L2    # fall thru or branch?
    ...
L2: ...
    beq $6,$0,L1    # fall thru or branch?
L3: ...
```

- Branch prediction by the compiler is difficult to exploit since the prediction needs to be decoded before it is used.

Effects from Pipeline Hazards

- Structural hazards are most often affected by multicycle operations (multiplies, divides, FP operations), which are sometimes not fully pipelined.
- Data hazards can cause performance problems in both integer and floating-point applications.
- Control hazards more often cause stalls in integer applications where branch frequencies are typically higher and less predictable.

Single Cycle Datapath Separated into Five Parts

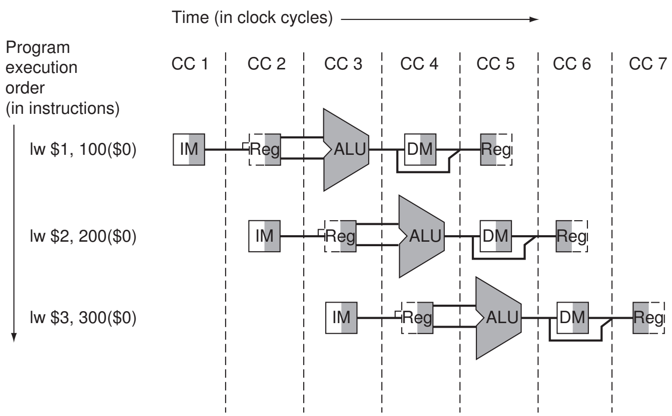


Flow of Data in the Datapath

- All flow of data in the single cycle datapath goes from left to right with two exceptions.
  - Placing the result back into the register file.
  - Updating the program counter (PC) with the branch target address.

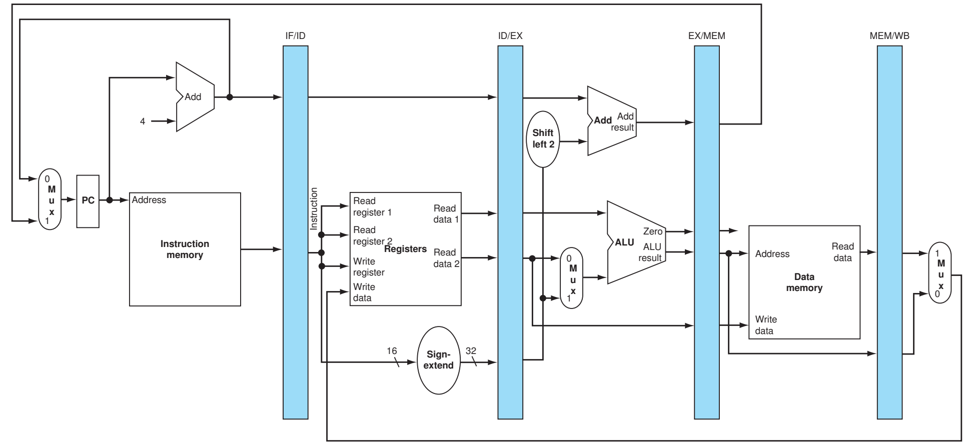
Instructions Being Executed Assuming Pipelined Execution

- The figure below suggests that each instruction has its own separate datapath, which is not the case.
- Note that each portion of the datapath is being used at most in one cycle.

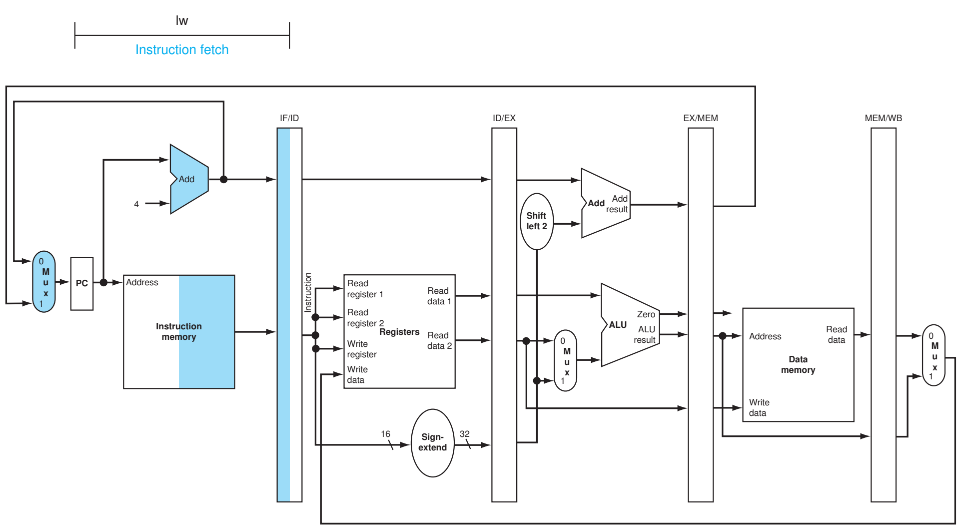


High-Level View of the Pipelined Datapath

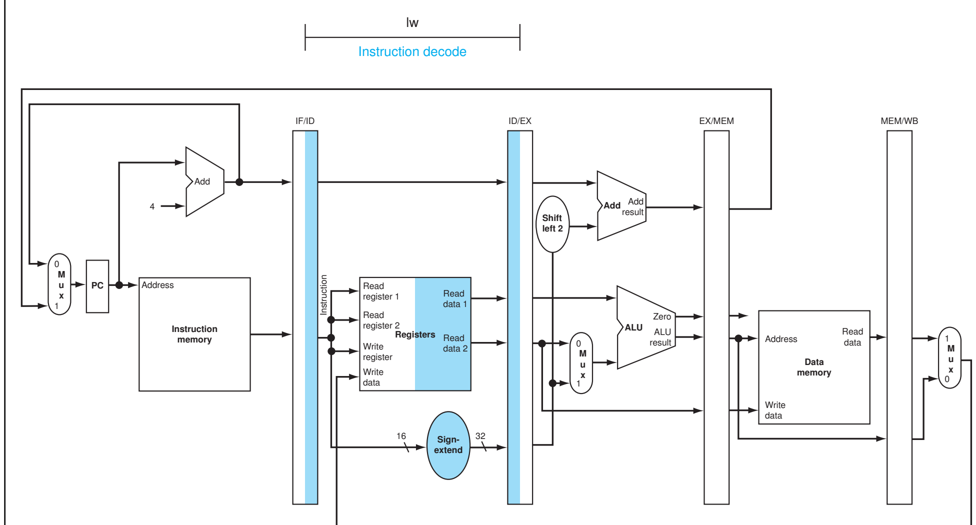
- Pipeline registers separate each stage, are labelled by the two stages they separate, and contain data and control information that may be needed for a later stage.



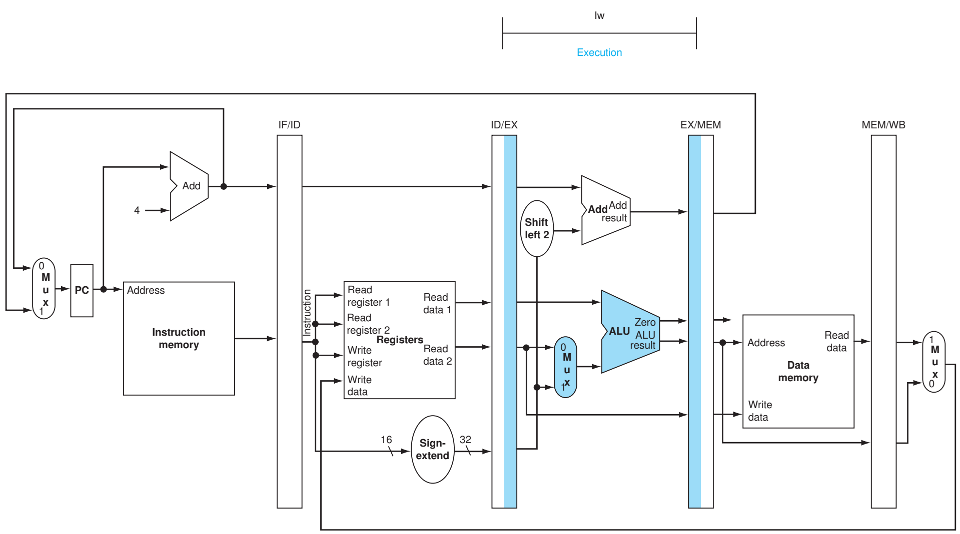
Load Instruction in the IF Stage



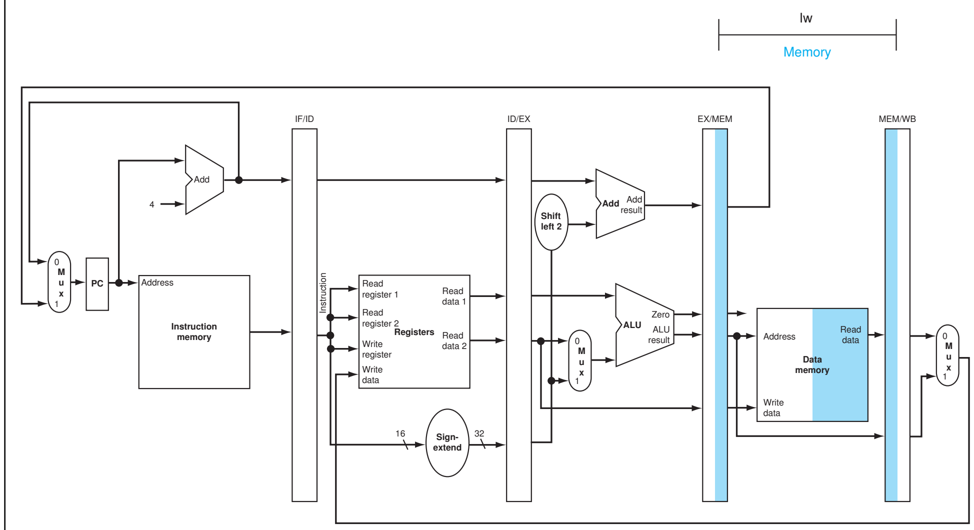
Load Instruction in the ID Stage



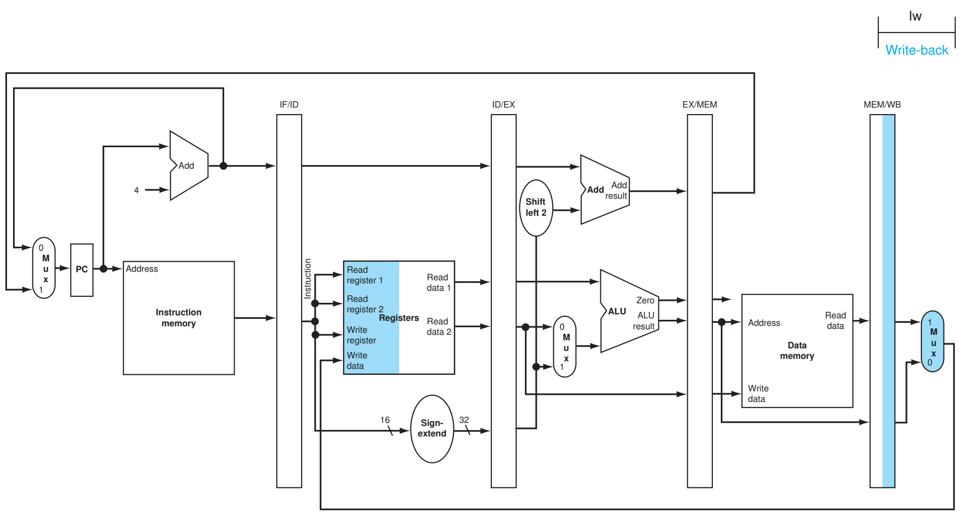
Load Instruction in the EX Stage



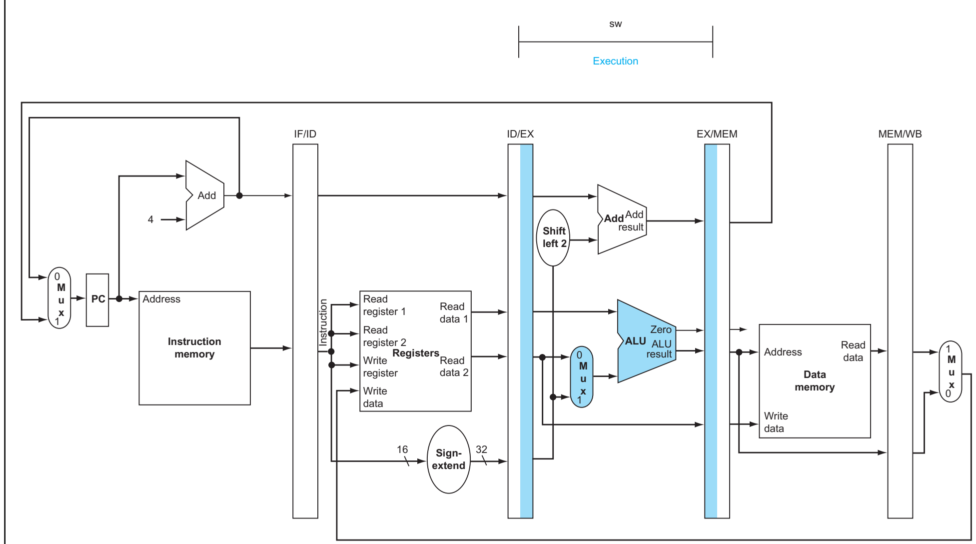
Load Instruction in the MEM Stage



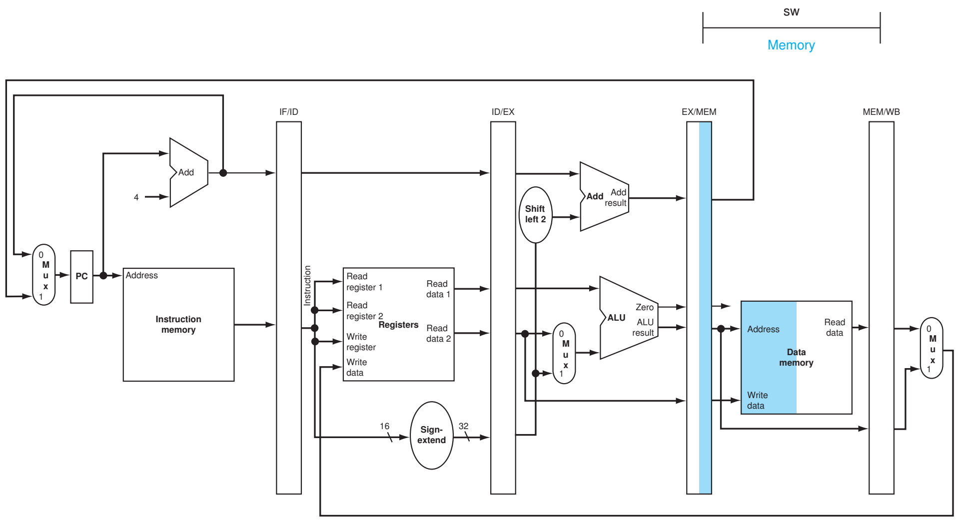
Load Instruction in the WB Stage



Store Instruction in the EX Stage

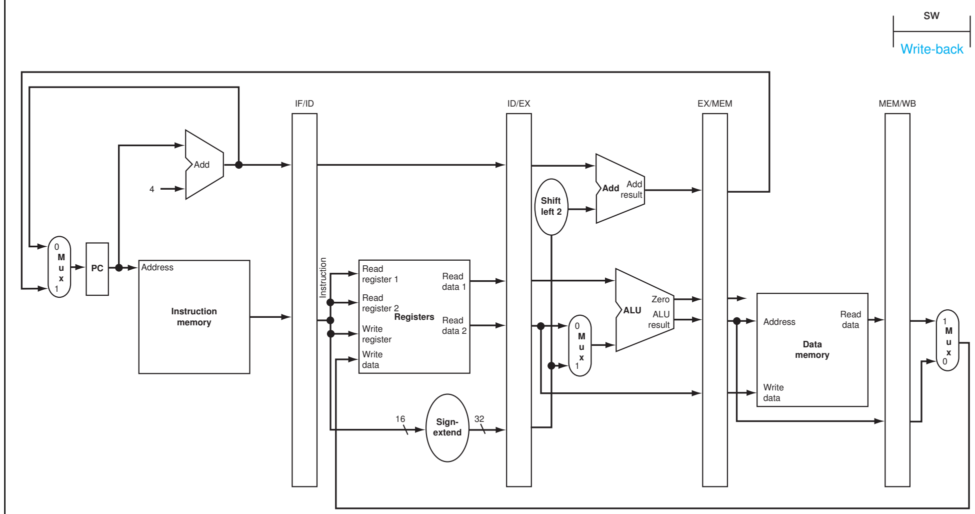


Store Instruction in the MEM Stage



Intro Hazards Pipeline Datapath Pipeline Control

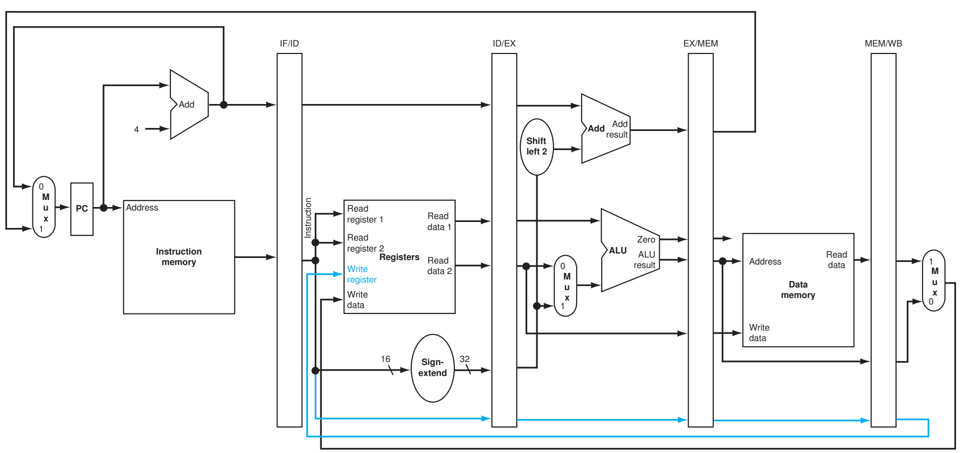
Store Instruction in the WB Stage



Intro Hazards Pipeline Datapath Pipeline Control

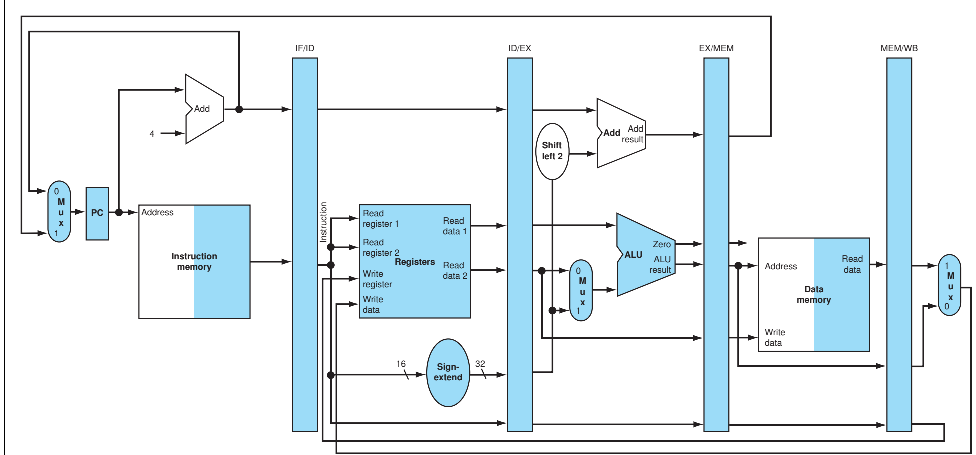
Corrected Datapath for a Load Instruction

- Now the correct write register number is used for a load instruction.

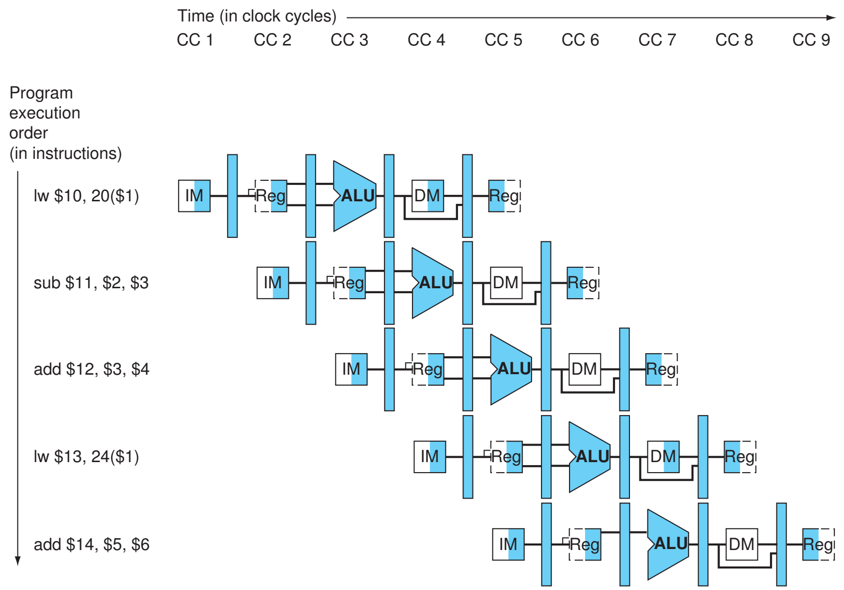


Intro Hazards Pipeline Datapath Pipeline Control

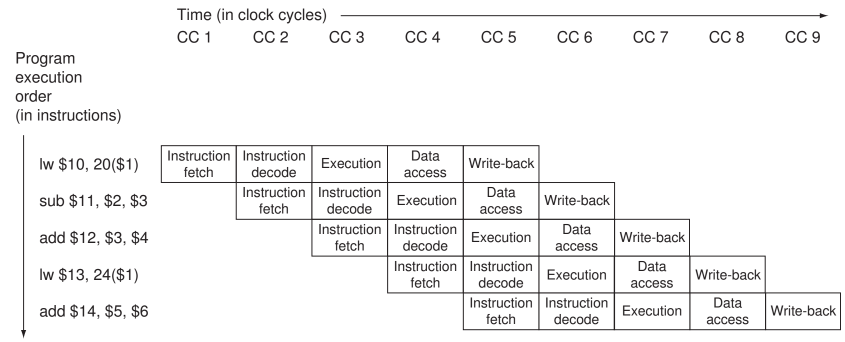
Datapath Showing All Portions Used for a Load Instruction



Multiple-Clock-Cycle Pipeline Diagram of Five Instructions

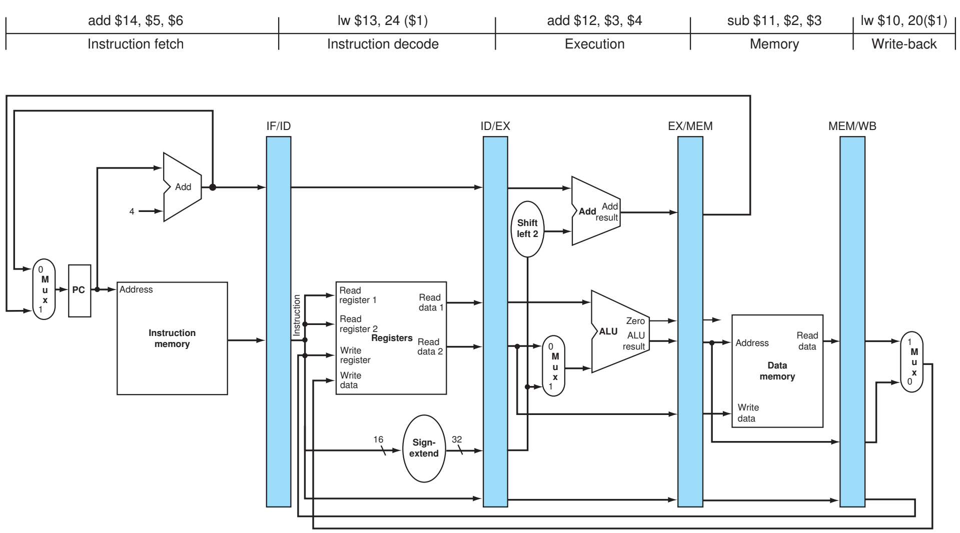


Traditional Pipeline Diagram of Five Instruction



Pipeline Datapath with Five Instructions Active

- Each instruction is in a different pipeline stage.



ALU Control Bits Depend on ALUOp and Function Code

- *ALUOp* is set depending on the *instruction opcode*.
- The *ALU control input* for R-type instructions is affected by the *Function code*.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Control Signal Effects

- The table below shows the effects for each signal that controls the pipelined datapath.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

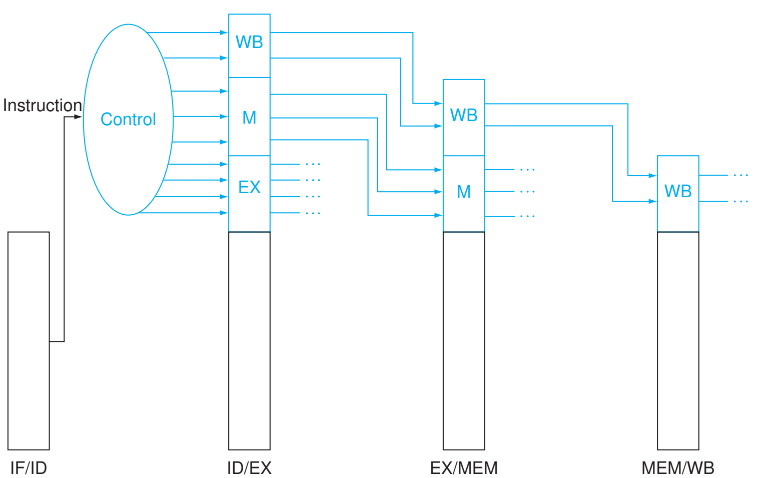
Control Signals Organized by Pipeline Stage

- The table below shows how the signals are used to control each pipeline stage after instruction decode (ID).

	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

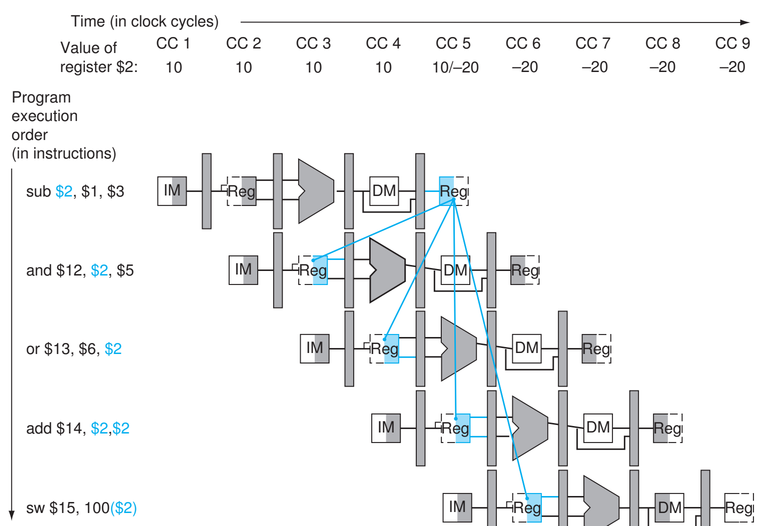
Control Signals Passed through Pipeline Registers

- Control signals are passed through pipeline registers until they are used.



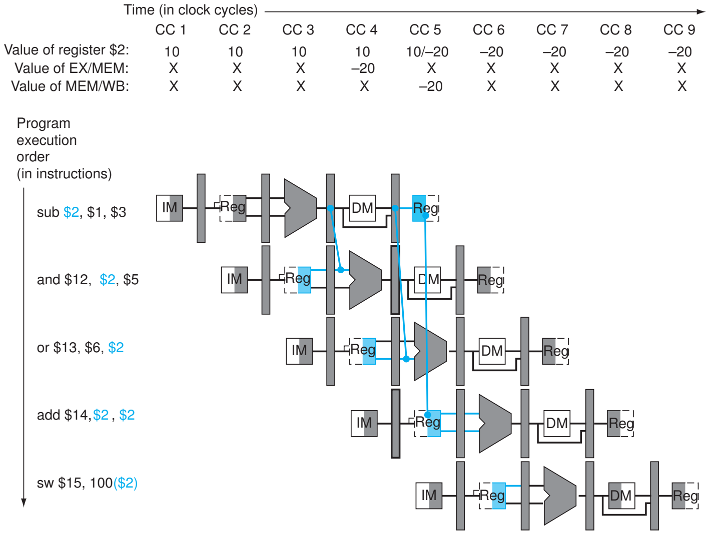
Example of Pipeline Dependences

- $2 is written in cycle 5 and read in cycles 3, 4, 5, and 6.



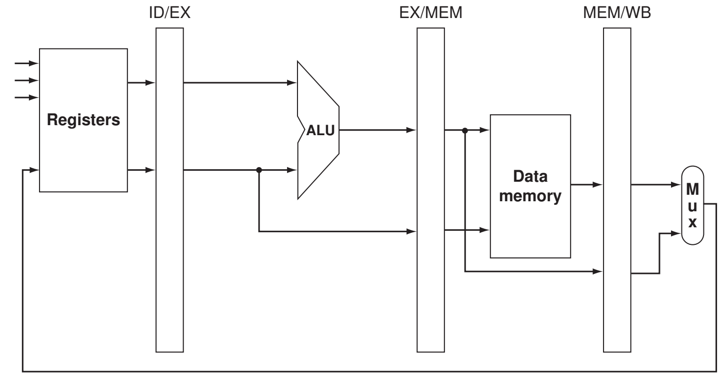
Example of Resolving Pipeline Hazards with Forwarding

- Value of $2 can be obtained from pipeline registers.



Pipelined Datapath without Forwarding

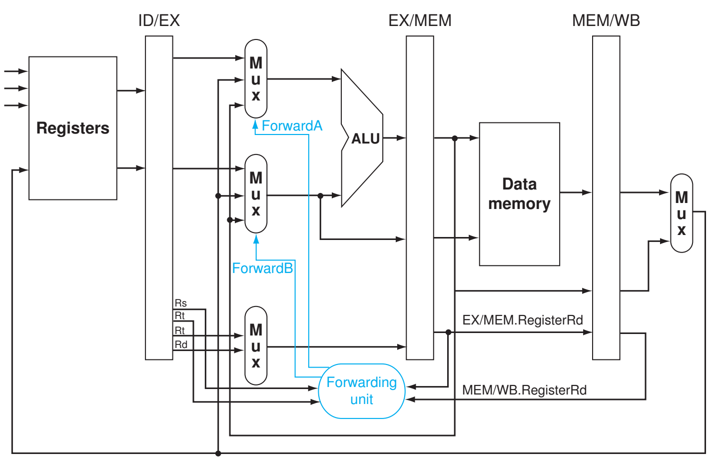
- The figure below shows a simplified pipelined datapath without forwarding.



a. No forwarding

Pipelined Datapath with Forwarding

- Forwarding compares a destination register number of an instruction to the source registers of later instructions.



b. With forwarding

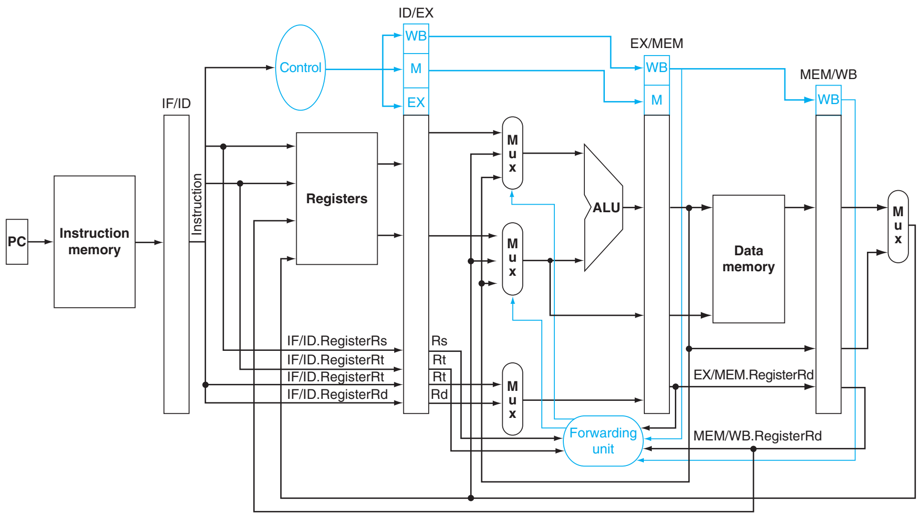
Control Values for the Forwarding Multiplexors

- The table below shows that each input to the ALU can come from three different sources.

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.

Datapath Modified to Resolve Data Hazards via Forwarding

- The forwarding unit takes register numbers as input and produces control signals as outputs.



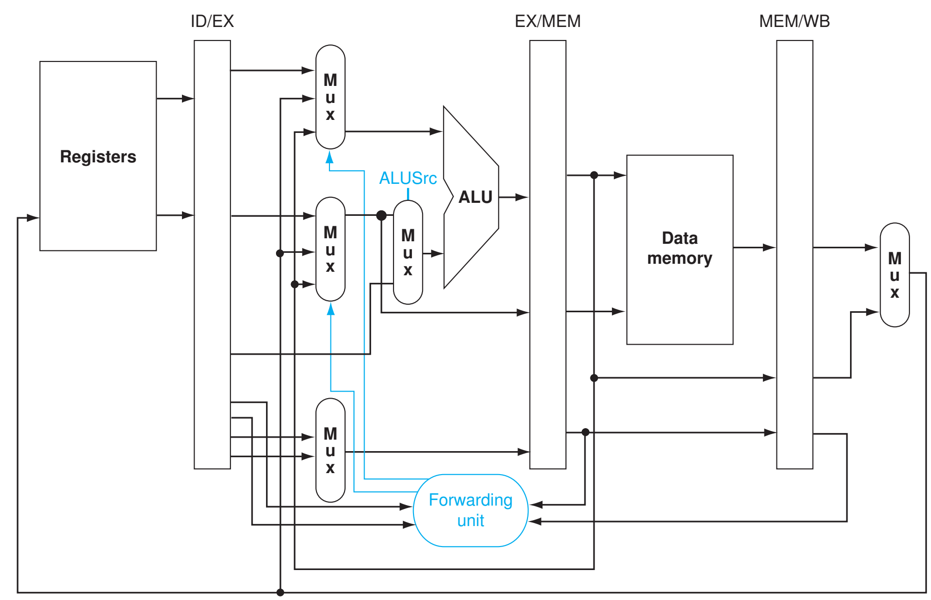
Forwarding Control Logic for EX Hazard

```
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
```


Forwarding Control Logic for MEM Hazard

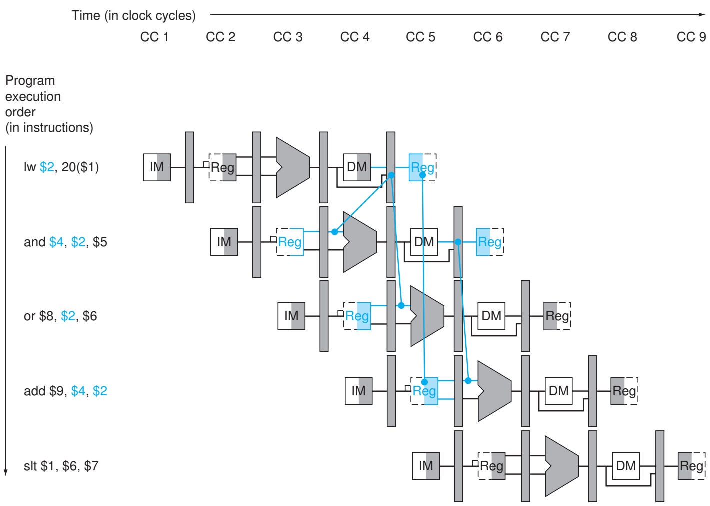
```
if (MEM/WB.RegWrite and
    MEM/WB.RegisterRd != 0 and
    not (EX/MEM.RegisterWrite and
         EX/MEM.RegisterRd != 0 and
         EX/MEM.RegisterRd != ID/EX.RegisterRs)
    MEM/WB.RegisterRd == ID/EX.RegisterRs)
   ForwardA = 10

if (MEM/WB.RegWrite and
    MEM/WB.RegisterRd != 0 and
    not (EX/MEM.RegisterWrite and
         EX/MEM.RegisterRd != 0 and
         EX/MEM.RegisterRd != ID/EX.RegisterRt)
    MEM/WB.RegisterRd == ID/EX.RegisterRt)
   ForwardB = 10
```


Datapath Allowing Immediate as Second ALU Input



Example Sequence of Instructions with a Load Hazard

- A hazard from a load followed by an immediate use of the loaded register cannot be resolved by forwarding.



Control Logic for Load Hazard Detection

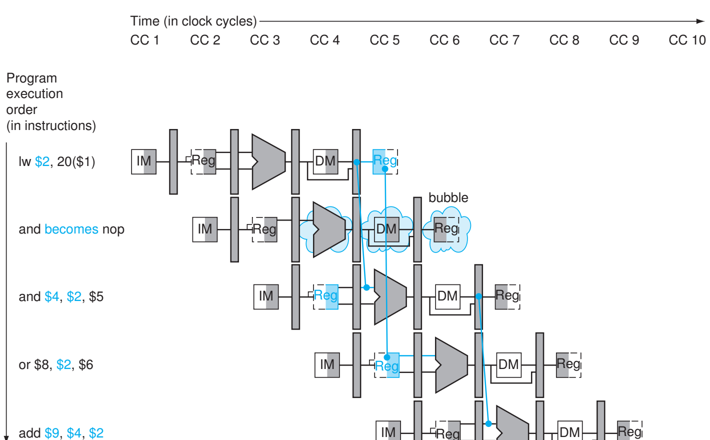
- Checking for hazards to stall the pipeline is performed in the instruction decode (ID) stage.

```
if (ID/EX.MemRead and
    (ID/EX.RegisterRt == IF/ID.RegisterRs or
     ID/EX.RegisterRt == IF/ID.RegisterRt))
  stall the pipeline
```

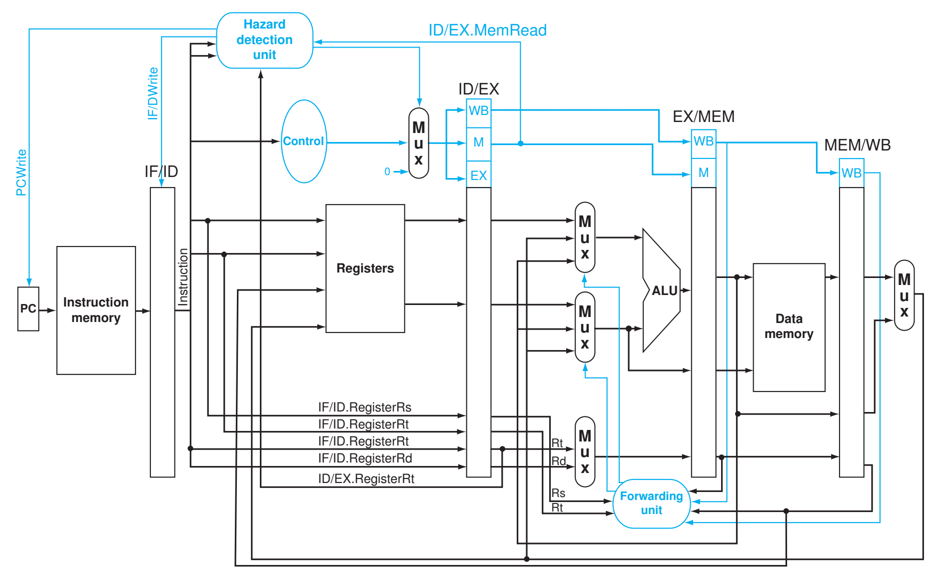
- Stalls can be inserted in the pipeline by deasserting all the control signals coming out of the ID stage.

How Stalls Are Inserted into the Pipeline

- If it is found that an instruction needs to stall, then a noop is inserted into the pipeline after the ID stage and the instruction stays in the ID stage.

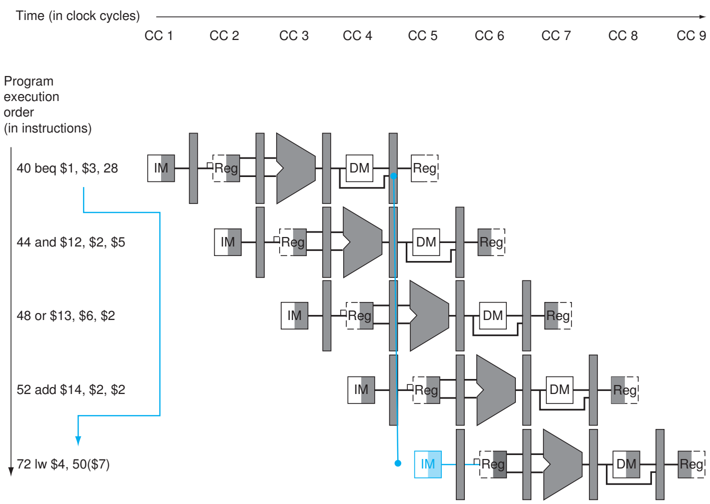


Pipeline Datapath with Hazard Detection Unit Added



The Impact of a Taken Branch on the Pipeline

- If the decision to take a conditional occurs in the MEM stage, then a taken branch will cause a delay of three cycles.

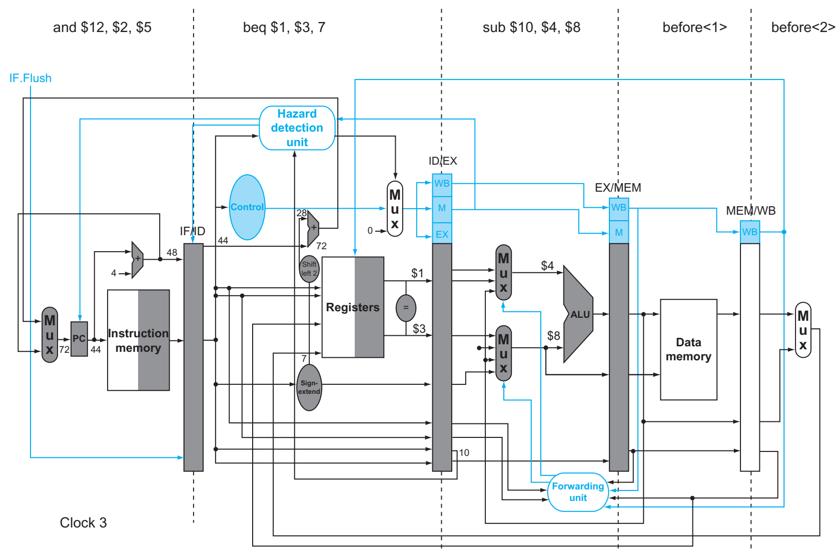


Reducing the Delay of Branches

- If the branch execution can be moved earlier in the pipeline, then fewer instructions will need to be flushed.
  - Compute the branch target address in the ID stage.
  - Compare the values of two registers for equality in the ID stage.

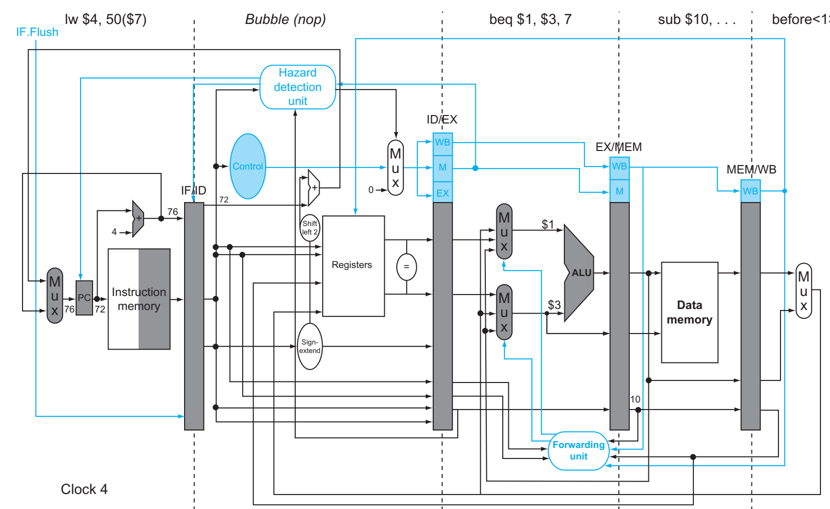
Pipeline Datapath When Branch Is Decoded

- Branch is now resolved in the ID stage.



Pipeline Datapath After Branch Is Taken

- A taken branch now causes a one cycle stall.

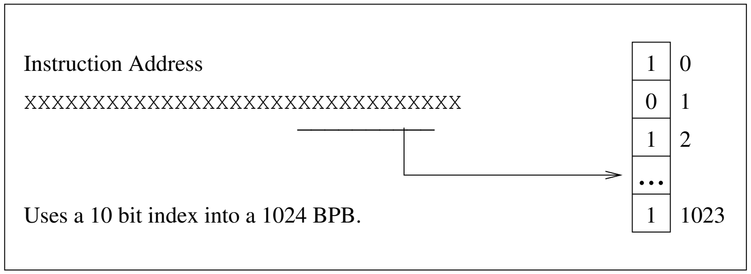


Problems with Resolving a Branch in the ID Stage

- There are multiple problems with the approach of trying to resolve a branch in the ID stage.
  - Will require new forwarding logic for the equality test.
  - May introduce new data hazards if one or both register values are not yet available.
  - If the pipeline is deeper (has more stages), which is common, then it is just infeasible to resolve the branch in the second stage.
- Solutions
  - Predict the branch result.
  - Delay the execution of the branch.

1-Bit Branch Prediction Buffer

- Small memory indexed by the lower portion of the word address of the branch instruction.
- Each element of this memory contains a bit indicating if the branch was last taken or not.
- If the prediction is found to be incorrect, then the bit is inverted.
- BPB with 1024 entries:



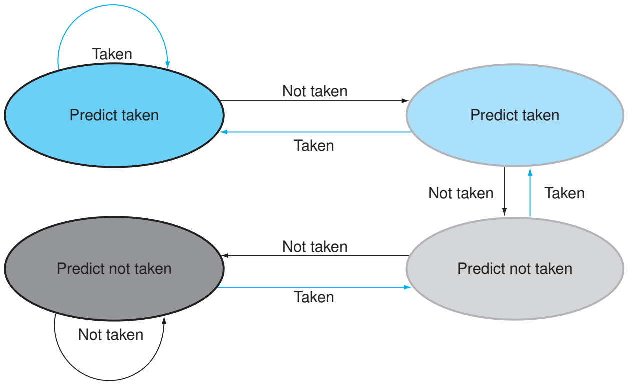
1-Bit Branch Prediction Buffer Example

- How often will each of the two branches associated with the following source code miss with a 1-bit predictor?

```
for (i = 0; i < 100; i++)
   if (i & 1)
      A;
   else
      B;
```

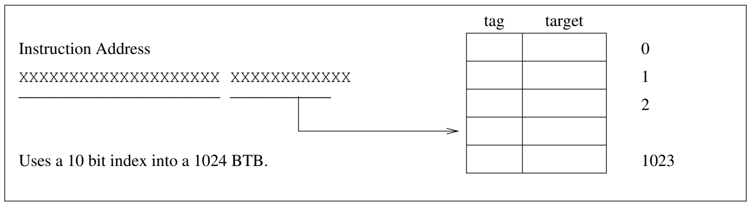

2-Bit Branch Prediction Buffer

- How often will the branches from the previous slide miss using the following 2-bit predictor?



Branch Target Buffer

- We not only need to predict the branch result, but we also need the branch target when the branch is taken.
- A branch target buffer contains a tag and a target address.
- The tag is the high-order bits of the branch address and is used to verify that the instruction is really a branch in the table.
- The index is again used to select an entry in the table.
- The target address is used to update the PC only if the tag matches and the branch is predicted taken by the BPB.

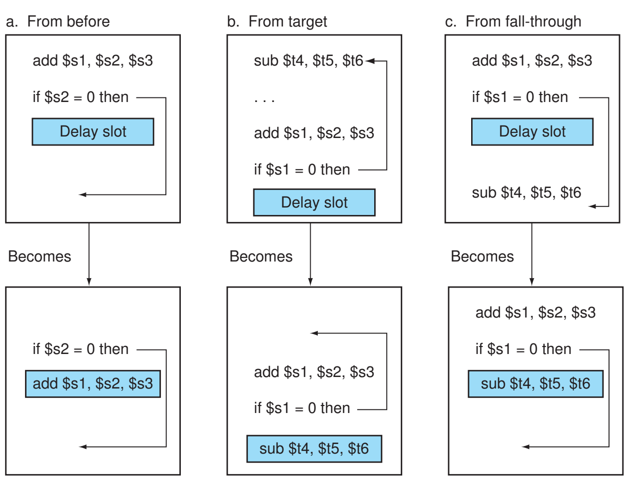


Delayed Branches

- Another approach is to delay the execution of the branch until the branch target address and the branch result is known.
- This means that a specified number of instructions after the branch will always execute.

Scheduling the Branch Delay Slot

- Typically delayed branches have a single delay slot.
- The figure below shows the three options for filling this slot.



Problems with Delayed Branches

- The delay slot cannot always be filled with a useful instruction due to dependences and the difficulty of predicting at compile time which is the most likely successor of the branch.
- Delayed branch slots are very hard to fill with multiple issue processors.

Final Datapath and Control