JEPPIAAR INSTITUTE OF TECHNOLOGY

“Self-Belief | Self Discipline | Self Respect”

DEPARTMENT

OF

ELECTRONICS AND COMMUNICATION ENGINEERING

LECTURE NOTES

EC8552 – COMPUTER ARCHITECTURE AND ORGANIZATION

(Regulation 2017)

Year/Semester: III / V ECE

2021 – 2022

Prepared by

Mr. N. Prabhakaran

Associate Professor / ECE

UNIT-III

PROCESSOR AND CONTROL UNIT

Basic MIPS implementation – Building datapath – Control Implementation scheme – Pipelining – Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions.

A BASIC MIPS IMPLEMENTATION:

The implementation that includes a subset of the core MIPS instruction set:

- The memory-reference instructions **load word (lw)** and **store word (sw)**
- The arithmetic-logical instructions **add, sub, AND, OR, and slt**
- The instructions **branch equal (beq)** and **jump (j)**

An Overview of the Implementation:

For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.

2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

- After these two steps, the actions required to complete the instruction depend on the instruction class.
- For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.
- The following diagram shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection.

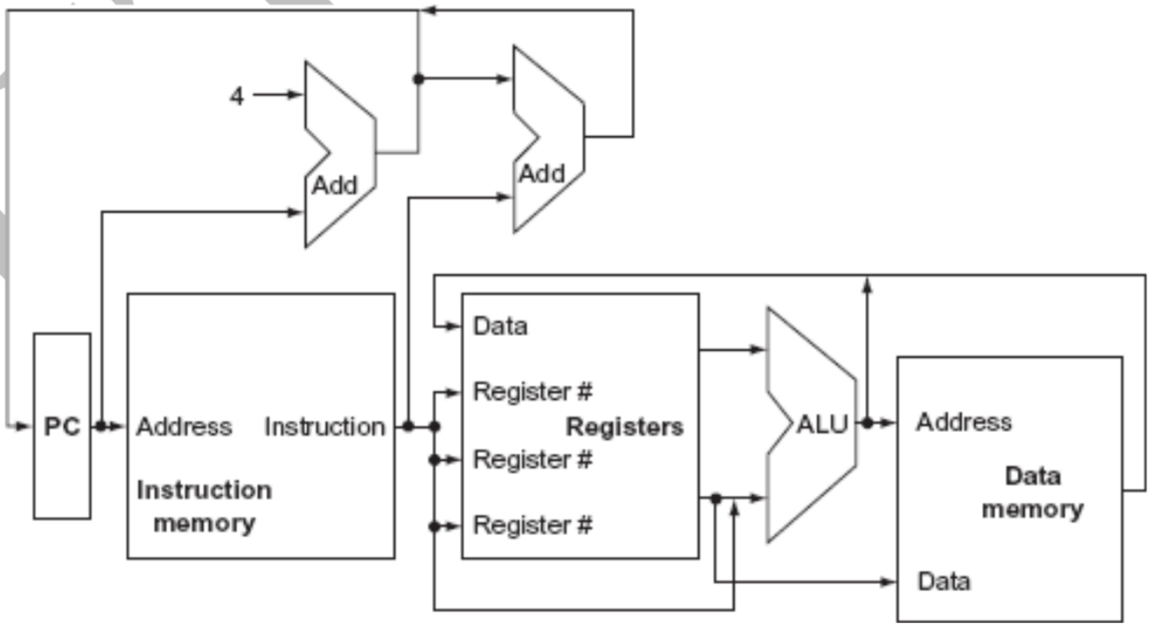


FIGURE 3.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

Operation:

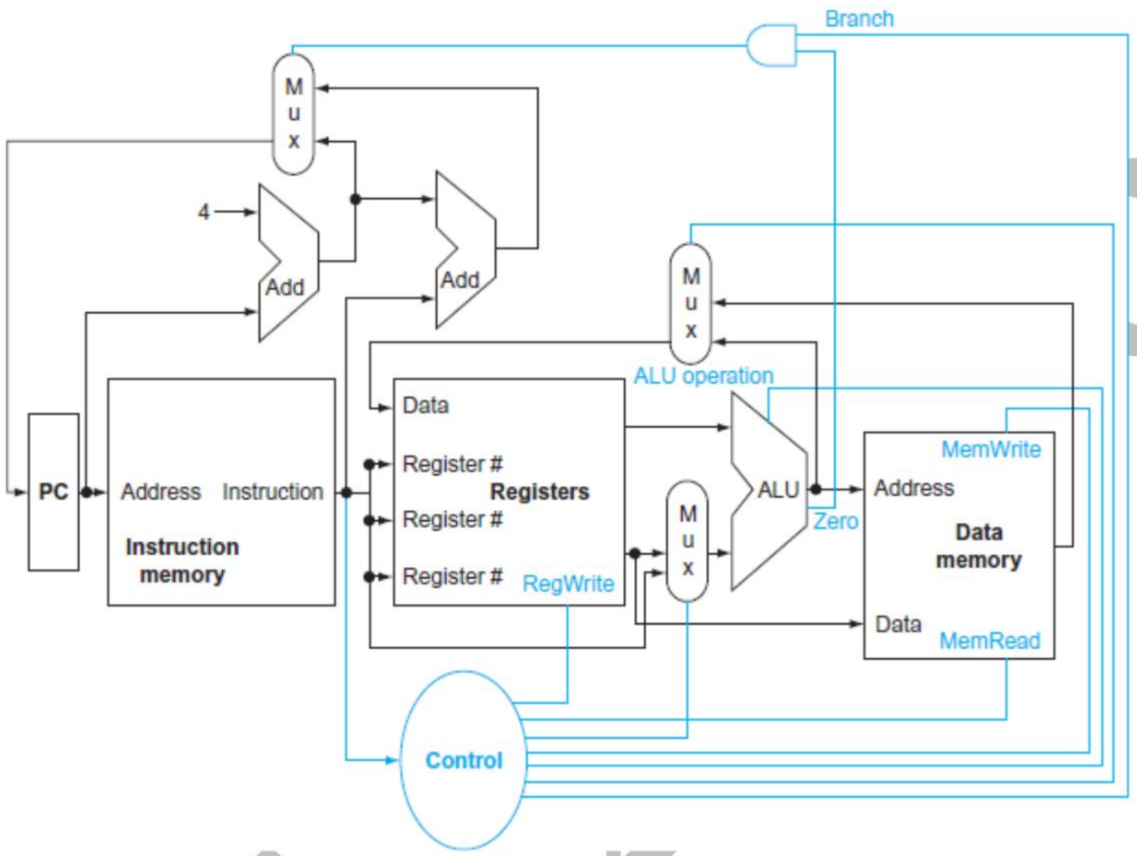
- All instructions start by using the program counter to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction.
- Once the register operands have been fetched, all the instruction classes, except jump, use the ALU after reading the registers.
    - Memory reference instructions (load or store) use the ALU for an address calculation.
    - Arithmetic Logical instructions use the ALU for the operation execution.
    - Branches use the ALU for comparison.
- The second input to the ALU can come from a register or the immediate field of the instruction.
- After using the ALU, the actions required to complete various instruction classes are not same.
    - If the operation is a memory reference instruction a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file.
    - If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
    - Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch off set are summed) or from an adder that increments the current PC by 4.

Basic implementation of MIPS with multiplexer:

- We must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a **multiplexor**, although this device might better be called a **data selector** which selects from among several inputs based on the setting of its control lines.
- The control lines are set based primarily on information taken from the instruction being executed.
- The following figure shows the datapath with the three multiplexors added, as well as control lines for the major functional units.
- A control unit is used to determine how to set the control lines for the functional units and two of the multiplexors.
- The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.
- Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset

field of the instruction (for a load or store).

The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.



Logic Design Conventions:

The datapath elements in the MIPS implementation consist of two different types of logic elements:

1. **Combinational Elements**:

- The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs.
- Given the same input, a combinational element always produces the same output.
- The ALU is an example of a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

2. State Elements:

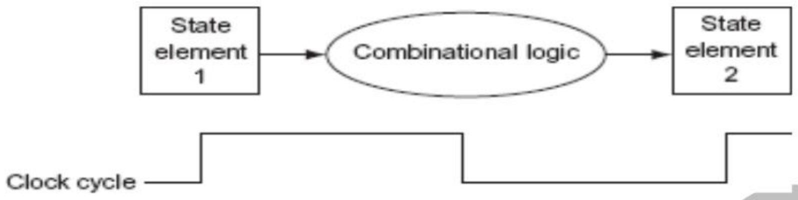
- It holds information about the state of the processor during the current clock cycle.
- An element contains state if it has some internal storage.
- All registers are state elements.
- A state element has at least two inputs and one output.
- The required inputs are the data value to be written into the element and the clock, which determines when the data value is written.
- The output from a state element provides the value that was written in an earlier clock cycle.

Clocking Methodology

- A **clocking methodology** defines when signals can be read and when they can be written.

The approach used to determine when data is valid and stable relative to the clock.

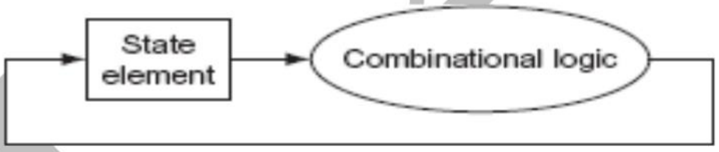
- All state elements including memory, are assumed to be positive edge-triggered; that is, they change on the rising clock edge.



- Figure shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle.
- All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle.
- The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

Edge-triggered clocking methodology:

- An **edge-triggered clocking methodology** means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa.
- An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle.



Control signal

- A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that is operated on by a functional unit.

Asserted: The signal is logically high or true.
Deasserted: The signal is logically low or false.

BUILDING A DATAPATH

Datapath

- It is a collection of function units organized in a manner to execute each class of instruction.

Datapath elements

- A unit used to operate on or hold data within a processor is called **datapath element.**
- In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

How to build a datapath:

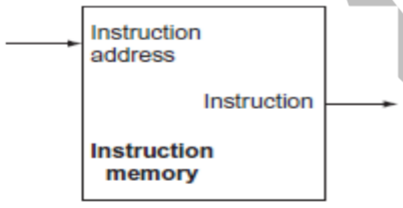
- Datapath design begins in examining the major components required to execute each class of MIPS instructions.

- First we have to know what the data path elements each instruction needs are, and also their control signals.

Stage: 1 [Datapath to fetch instruction and increment PC]

- The following diagram shows the datapath elements needed to fetch an instruction.
- The state elements are the **instruction memory, the program counter and adder.**

- **Instruction memory** - a memory unit to store the instructions of a program and supply instructions given an address.
- The instruction memory need only provide read access because the datapath does not write instructions.

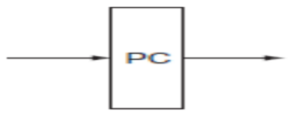


a. Instruction memory

- The output at any time reflects the contents of the location specified by the address input, and no read control signal is needed.

Program counter

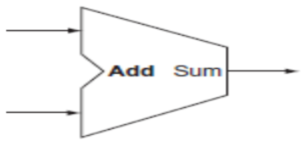
- The register containing the address of the instruction in the program being executed is called **program counter.**
- The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal.



b. Program counter

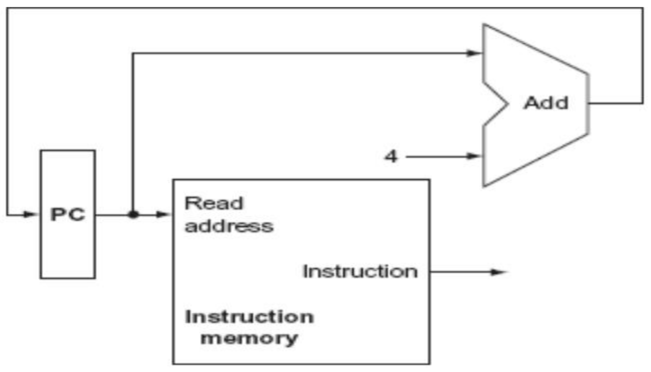
Adder

- **Adder** is used to increment the PC to the address of the next instruction.
- The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.



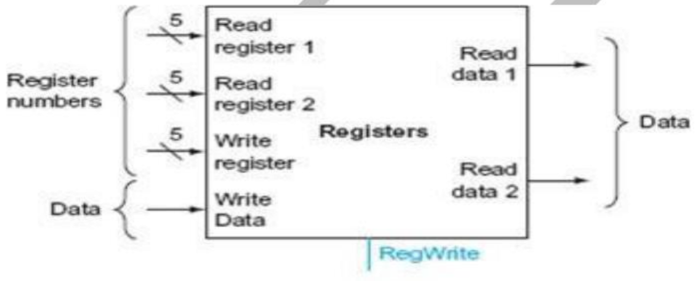
c. Adder

Combined all three elements into single stage

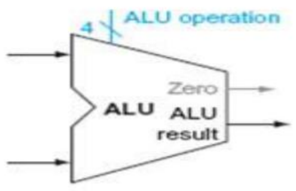


Stage: 2 [Datapath segment for multiport register file and the ALU]
Register File:

- A **register file** is a collection of registers in which any register can be read or written by specifying the number of the register in the file.
- R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction.
- For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers.
- To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register.
- The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.



a. Registers

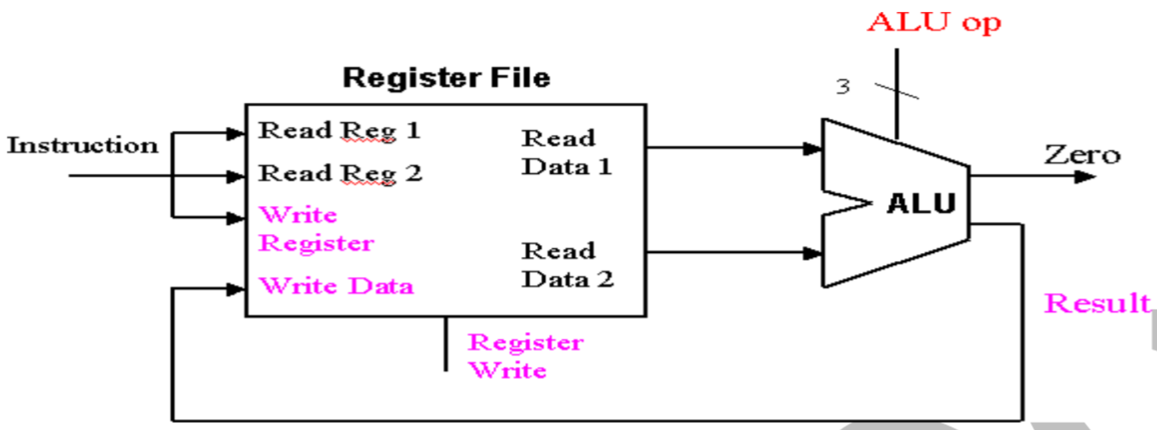


b. ALU

ALU:

- ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide.
- The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits control signal.
- ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract.
- If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. We will be using it only to implement the equal test of branches.

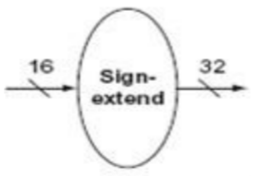
Combined two elements into single stage



Stage: 3 [Datapath segment for Branch Instruction]

Sign-extend

- To increase the size of a data item by replicating the high-order sign bit of the original data item in the high order bits of the larger, destination data item.



b. Sign extension unit

Branch

- A type of branch where the instruction immediately following the branch is always executed independent of whether the branch condition is true or false.

Branch taken

- A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional jumps are taken branches.

Branch not taken or (untaken branch)

- A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

Branch target address

- The address specified in a branch, which becomes the new program counter (PC) if the branch is taken.
- In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

Example:

- The beq instruction has three operands, two registers that are compared for equality, and a 16-bit off set used to compute the branch target address relative to the branch instruction address. **Ex: beq $t1,$t2,offset.**
- To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC.

There are two details in the definition of branch instructions.

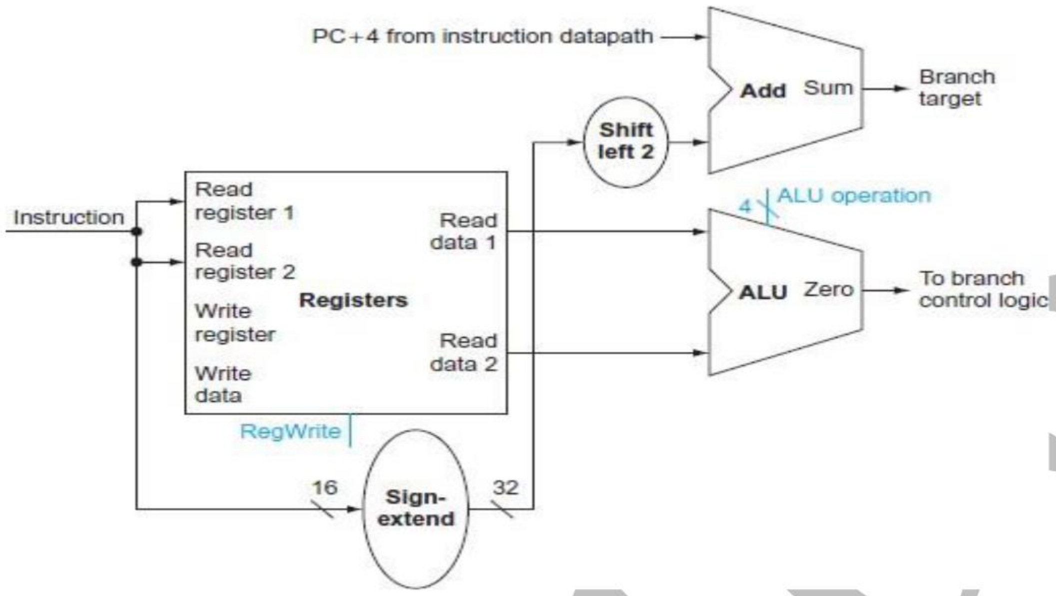
- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch i.e., PC+4 the address of the next instruction.

- The architecture also states that the offset field is shifted left 2 bits so that it is a word off set; this shift increases the effective range of the offset field by a factor of 4.

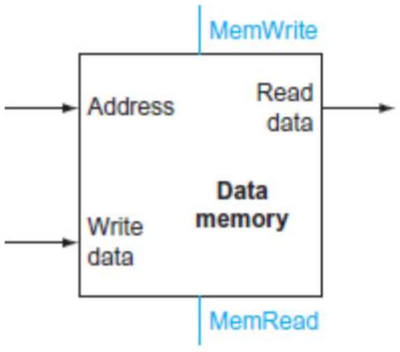
Branch Target Address = PC+4+offset (Shifted left 2 bits)

- The branch datapath must perform **two operations:** Compute the branch target address and compare the register contents.
- To compute the branch target address, the branch datapath includes a sign extension unit, shifter and an adder.
- Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Combined Diagram:



Stage: 4 [Datapath Segment for Load Word and Store Word Instructions] Data Memory

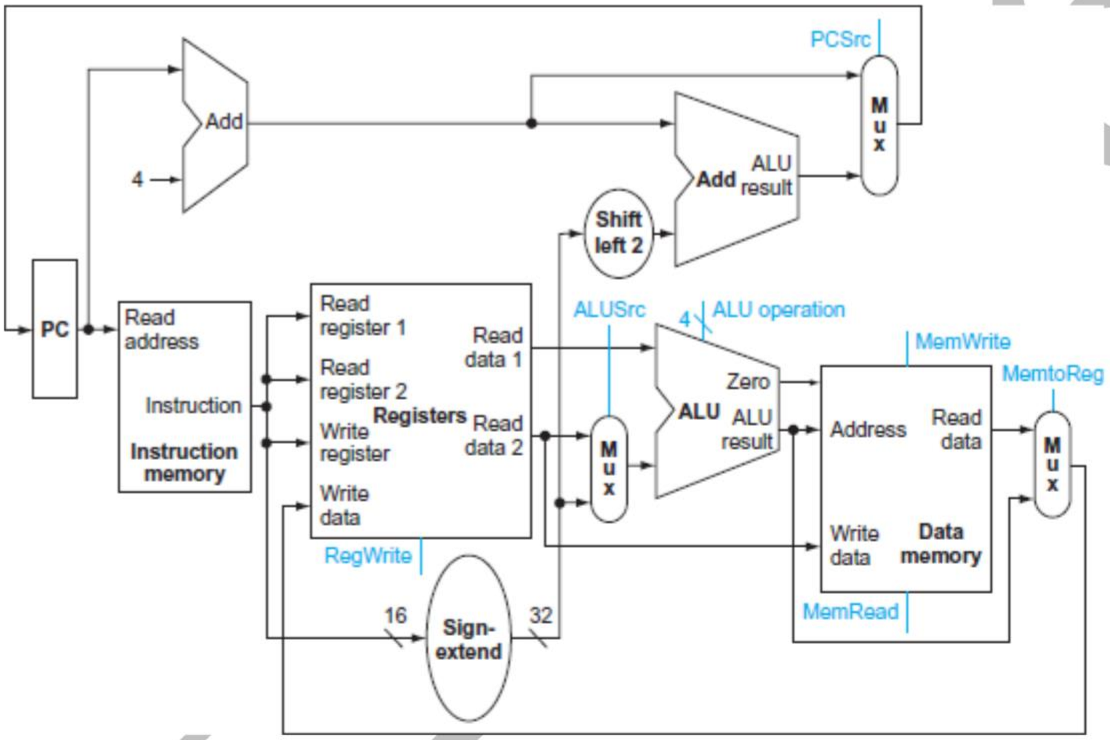


- The data memory unit is a state element with inputs for the address and the write data, and a single output for the read result. It has separate read and write controls to control the read and write operations.
- Although only one of these may be asserted on any given clock. The memory unit needs a read and write control signal.
- Consider the MIPS load word and store word instructions, which have the general form **Ex: lw $t1, offset_value ($t2)** or **sw $t1, offset_value ($t2).**
- These instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed off set field contained in the instruction.
- If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1.
- If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is $t1.

Building a Datapath with all the stages:

- Now we can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch, the datapath from R-type and memory instructions, and the datapath for branches.

- The following figure shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder for computing the branch target address.
- An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.
- The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control.



A CONTROL IMPLEMENTATION SCHEME

- This simple implementation covers load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than.

The ALU Control

- The MIPS ALU in defines the 6 following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

- Depending on the instruction class, the ALU will need to perform one of these first five functions.
  - For load word and store word instructions, we use the ALU to compute the memory address by addition.
  - For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-

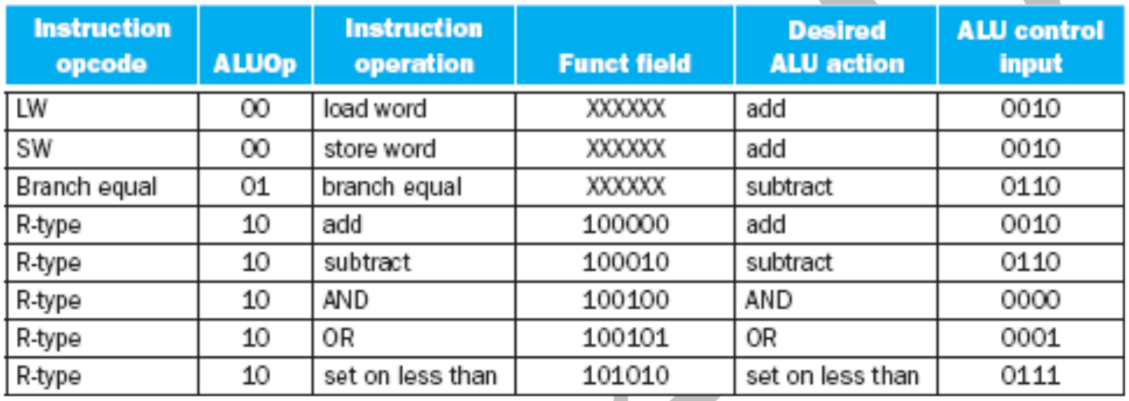
bit funct (or function) field in the low-order bits of the instruction

- For branch equal, the ALU must perform a subtraction.

- We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp.
- The 2 bits ALUOp is interpreted as shown in Table.

ALUOp	**Action**
00	loads and stores
01	subtract for beq
10	determined by the operation encoded in the funct field
11	--

- The following table shows how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code.



Instruction opcode	ALUOp	Instruction operation	Funct field	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

- Here multiple levels of decoding technique is used.

  Adv of using multiple levels of decoding:

  1. It reduces the size of the main control unit.
  2. Use of several smaller units may also increase the speed of the control unit.

<u>Truth table</u>

- From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.

<u>Don't-care term</u>

- An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

<u>Opcode</u>

- The field that denotes the operation and format of an instruction.
- The op field, is called the opcode, is always contained in bits 31:26. We will refer to this field as op[5:0].

- Designing other controls than ALU controls begins with identifying the fields of an instruction and the control lines that are needed for the datapath.
- There are three instruction classes: the R-type, branch, and load-store instructions. The following diagram shows these formats.
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The 16-bit off set for branch equal, load, and store is always in positions 15:0.

- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd).

- Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

- Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. Ex: add, sub, AND, OR, and slt.
- The ALU function is in the funct field and is decoded by the ALU control design.
- Instruction format for load (opcode = 35ten) and store (opcode = 43ten) instructions.
- The register rs is the base register(25:21) that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory.
- Instruction format for branch equal (opcode =4). The registers rs and rt are the source registers that are compared for equality.
- The 16-bit address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address. The following table describes seven other control lines.
- These nine control signals (seven from above table and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits 31 to 26.
- When the 1-bit control to a two way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input.

Signal name	Effect when deasserted	Effect when asserted
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.

- The following diagram shows the datapath with the control unit and the control signals.

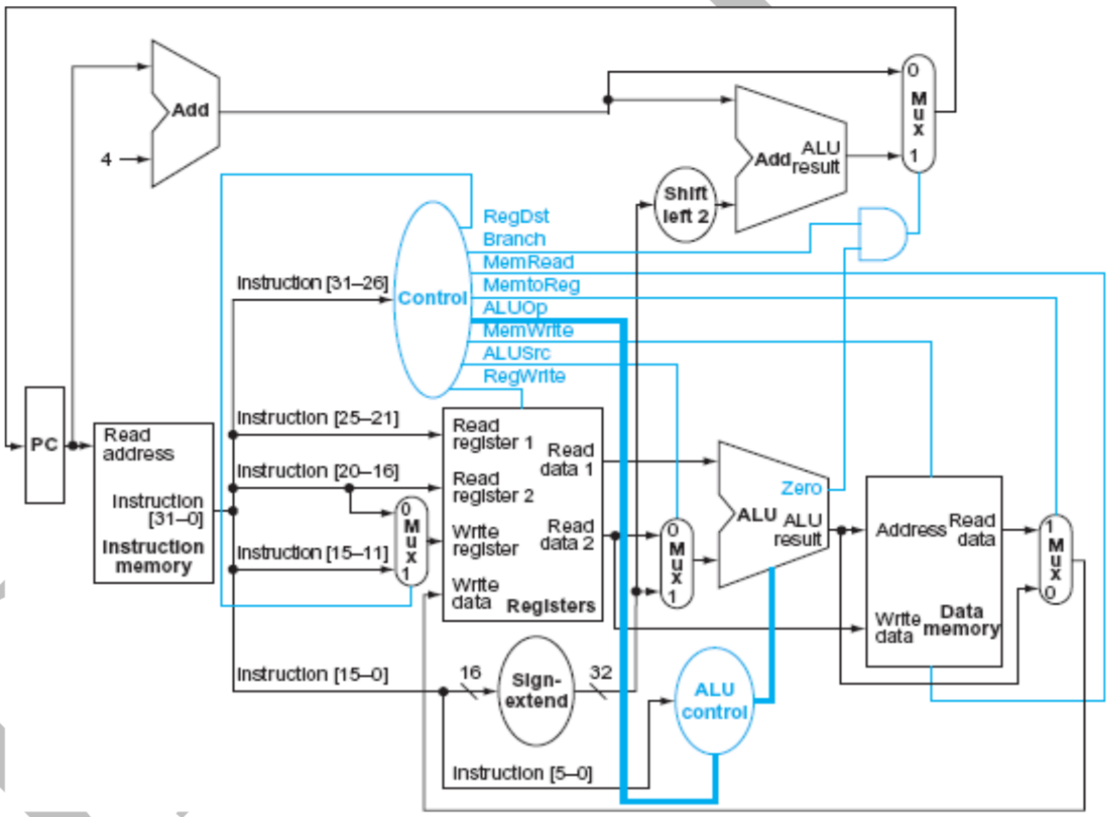
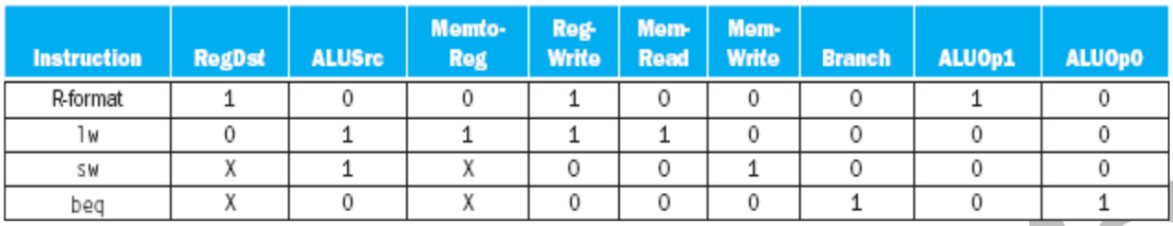


Figure: The simple datapath with the control unit

- The input to the control unit is the 6-bit opcode field from the instruction.
- The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg).
- Three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp).
- An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC.

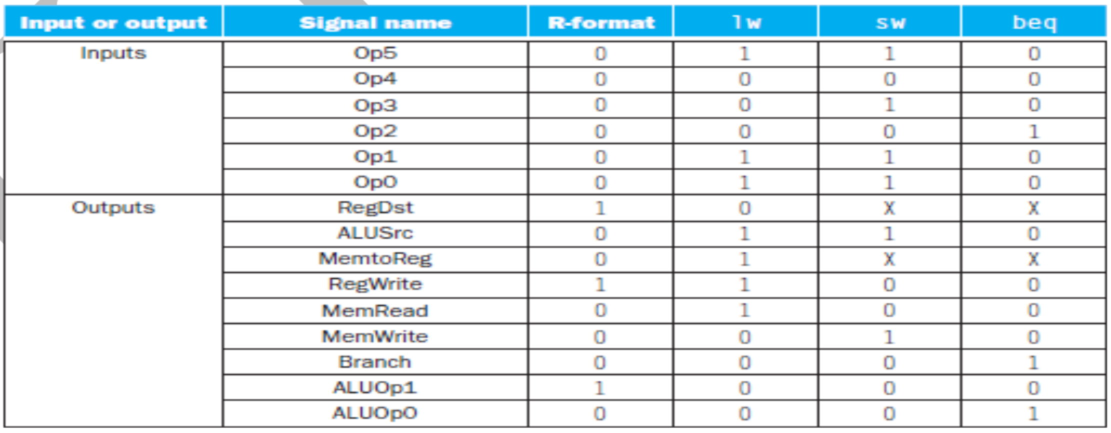
- Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit.The control lines is completely determined by the opcode fields of the instruction as shown below



Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- The first row of the table corresponds to the R-format instructions (add, sub, AND, OR, and slt). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the ALUSrc and RegDst are set.
- An R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory.
- When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.
- The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field.
- The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation.
- The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register.
- The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp fi eld for branch is set for a subtract (ALU control = 01), which is used to test for equality.
- Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0

<u>Finalizing Control</u>



Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

- The top half of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings.
- The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs.

<u>Single-cycle implementation:</u> An implementation in which an instruction is executed in one clock cycle called single clock cycle implementation.

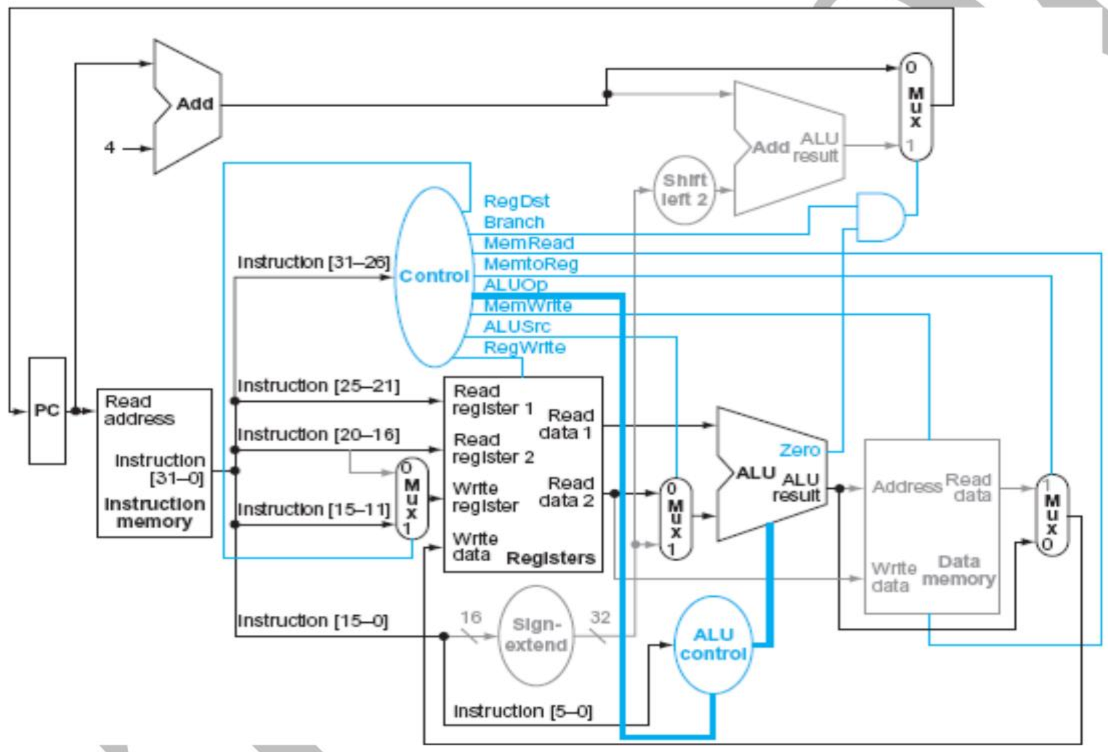
<u>Operation of the Datapath Example:</u> add $t1,$t2,$t3

Step:1 The instruction is fetched, and the PC is incremented.

Step:2 Two registers, $t2 and $t3, are read from the register fi le; also, the main control unit computes the setting of the control lines during this step.

Step:3 The ALU operates on the data read from the register fi le, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

Step:4 The result from the ALU is written into the register fi le using bits 15:11 of the instruction to select the destination register ($t1).



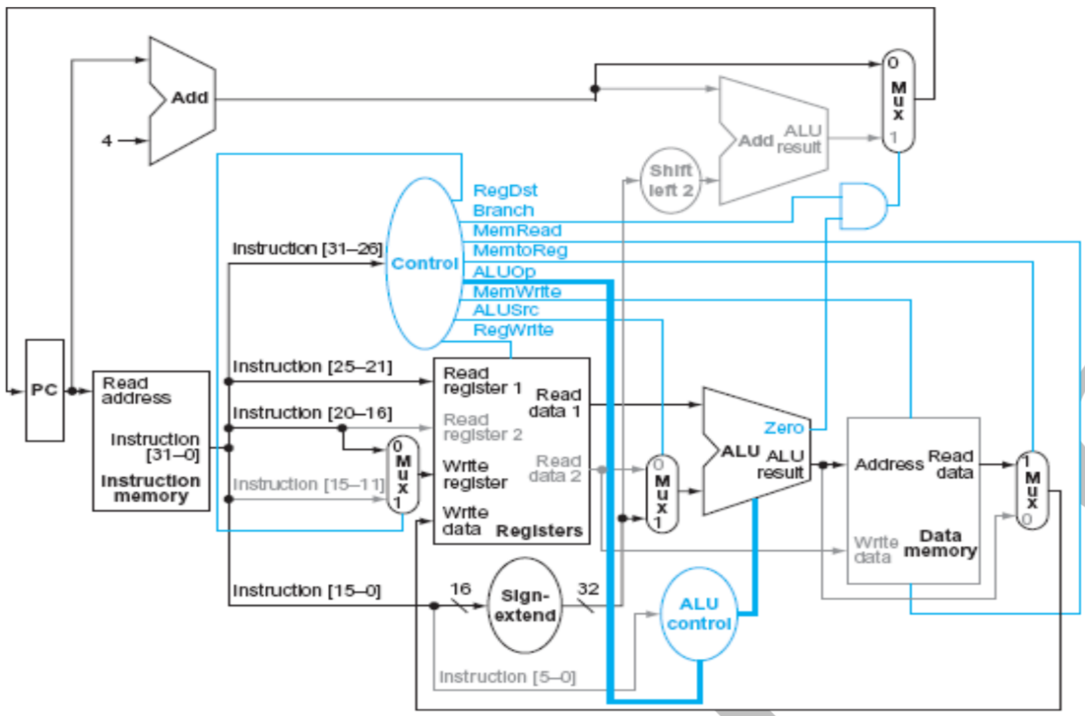
. **<u>Example:</u> lw $t1, offset ($t2)**

Step:1 An instruction is fetched from the instruction memory, and the PC is incremented.

Step:2 A register ($t2) value is read from the register file.

Step:3 The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).

Step:4 The sum from the ALU is used as the address for the data memory. **Step:5** The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).



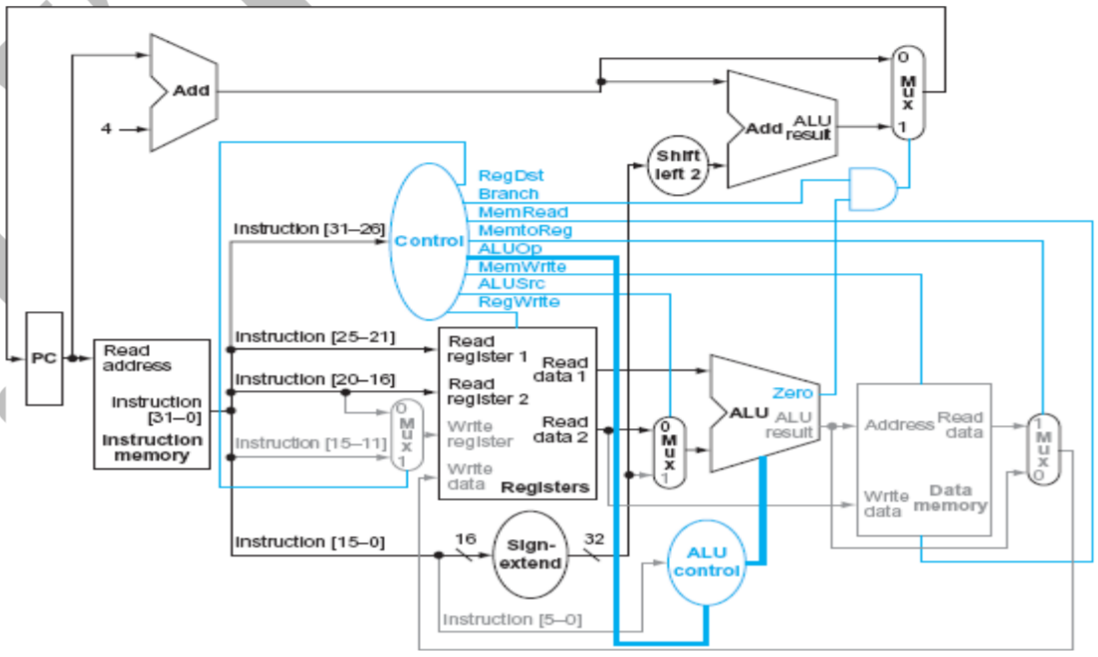
Example: **beq $t1, $t2, offset**

Step:1 An instruction is fetched from the instruction memory, and the PC is incremented.

Step:2 Two registers, $t1 and $t2, are read from the register fi le.

Step:3 The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is addedto the sign-extended, lower 16 bits of the instruction (offset) shift ed left by two; the result is the branch target address.

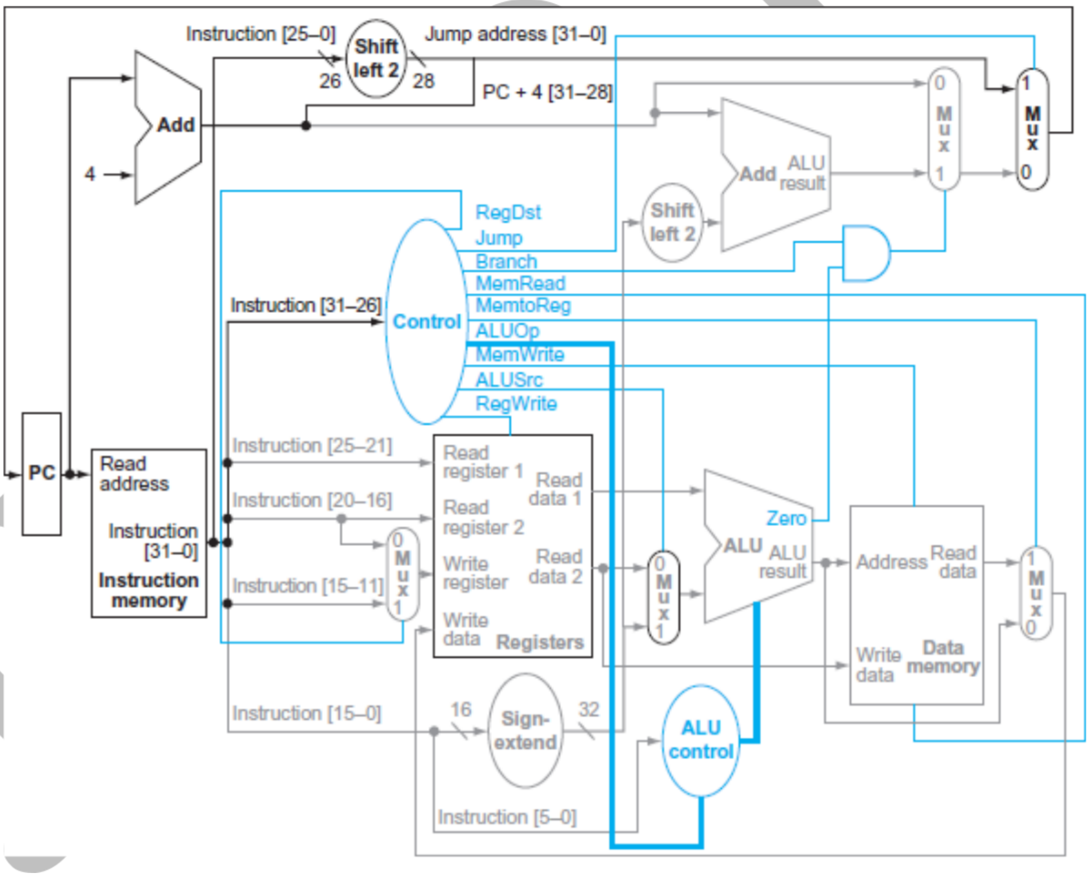
Step:4 The Zero result from the ALU is used to decide which adder result to store into the PC.



IMPLEMENTING JUMPS

Field	000010	address
Bit positions	31:26	25:0

- The jump instruction, looks somewhat like a branch instruction but computes the target PC differently and is not conditional.
- The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4.
- Thus, we can implement a jump by storing into the PC the concatenation of the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)
- An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one.
- This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.



AN OVERVIEW OF PIPELINING

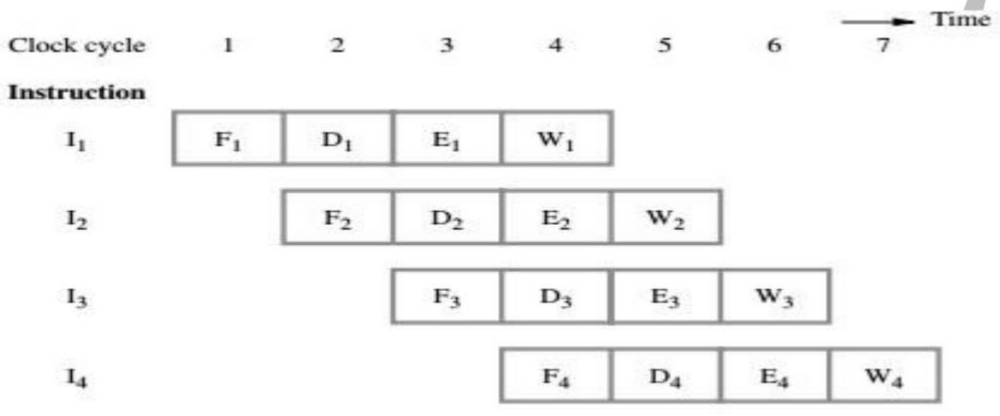
Pipelining:

An implementation technique in which multiple instructions are overlapped in execution is called pipeline. The different pipelining stages are,

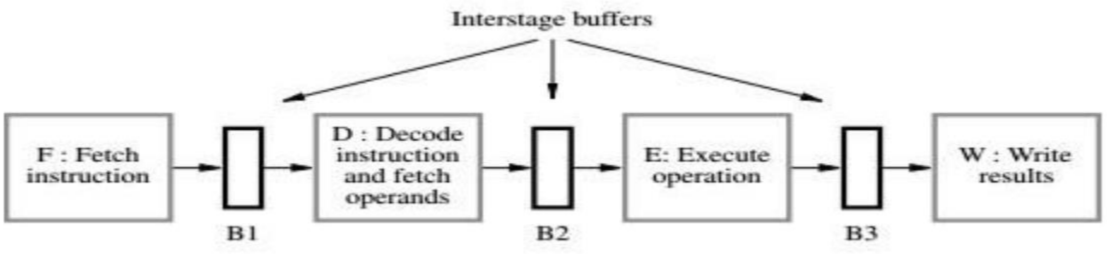
1. **Fetch** - Fetch instruction from memory.

2. **Decode**- Read registers while decoding the instruction. The regular format of MIPS instructions allow reading and decoding to occur simultaneously.
3. **Execute**- Execute the operation or calculate an address.
4. **Access**- Access an operand in data memory.
5. **Write**- Write the result into a register.

Four stage Instruction Pipelining



(a) Instruction execution divided into four steps



(b) Hardware organization

Hardware units are organized into stages:

- Execution in each stage takes exactly 1 clock period. Stages are separated by pipeline registers that preserve and pass partial results to the next stage.

Performance = complexity + cost.

- The pipeline approach brings additional expense plus its own set of problems and complications, called hazards.
- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline.
- If the stages are perfectly balanced, then the time between instructions on the pipelined processor – assuming ideal conditions – is equal to

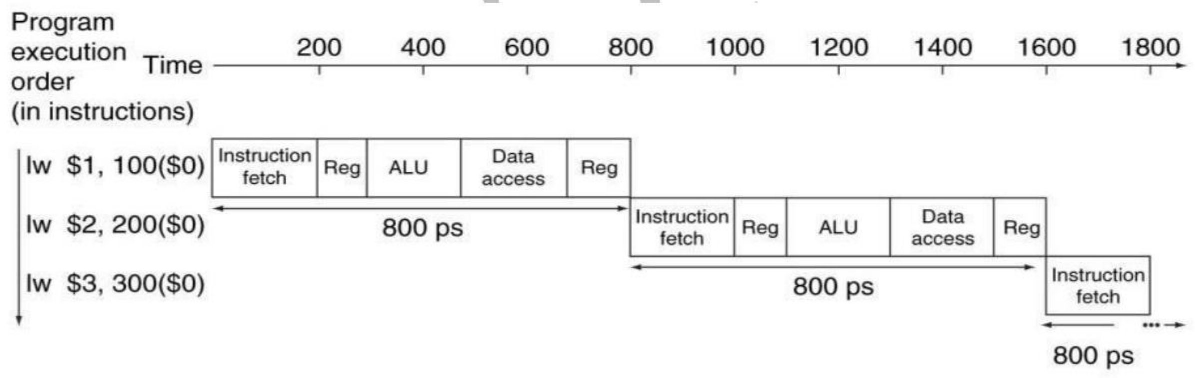
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath. This increases throughput, so programs can run faster. One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.

Example – Single-Cycle versus Pipelined Performance

- Consider a simple program segment consists of eight instructions: lw, sw, add, sub, AND, OR, slt and beq. Compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write.
- The following table shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction is lw – so the time required for every instruction is 800 ps. Thus, the time between the first and fourth instructions in the non-pipelined design is 3 x 800 ns or 2400 ps.
- Assume that following table shows the time taken by each and every stages of pipeline for different instruction

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



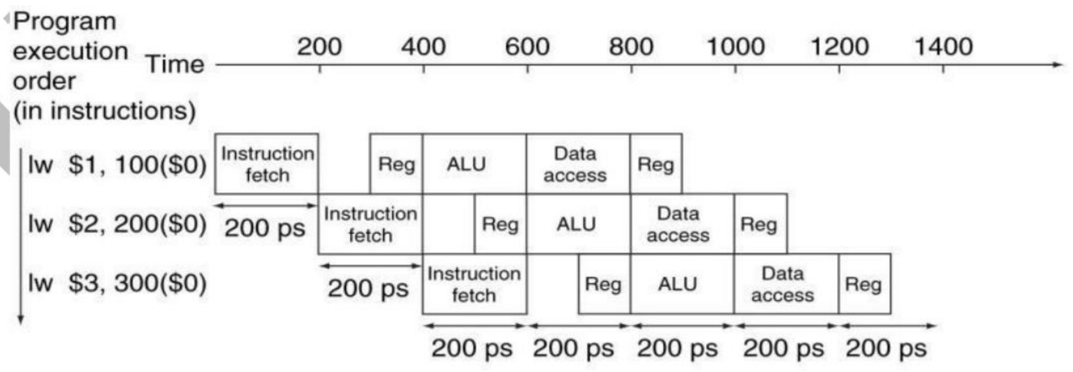


Figure: Single-Cycle, Non-Pipelined Execution in top versus Pipelined Execution in bottom.

- By comparing above two diagram, it is clear that pipeline process is best and it take reduce time to execute the instruction.
- Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction.

Six stages in the pipeline:

1. Fetch instruction: Instructions are fetched from the memory into a temporary buffer before it gets executed.

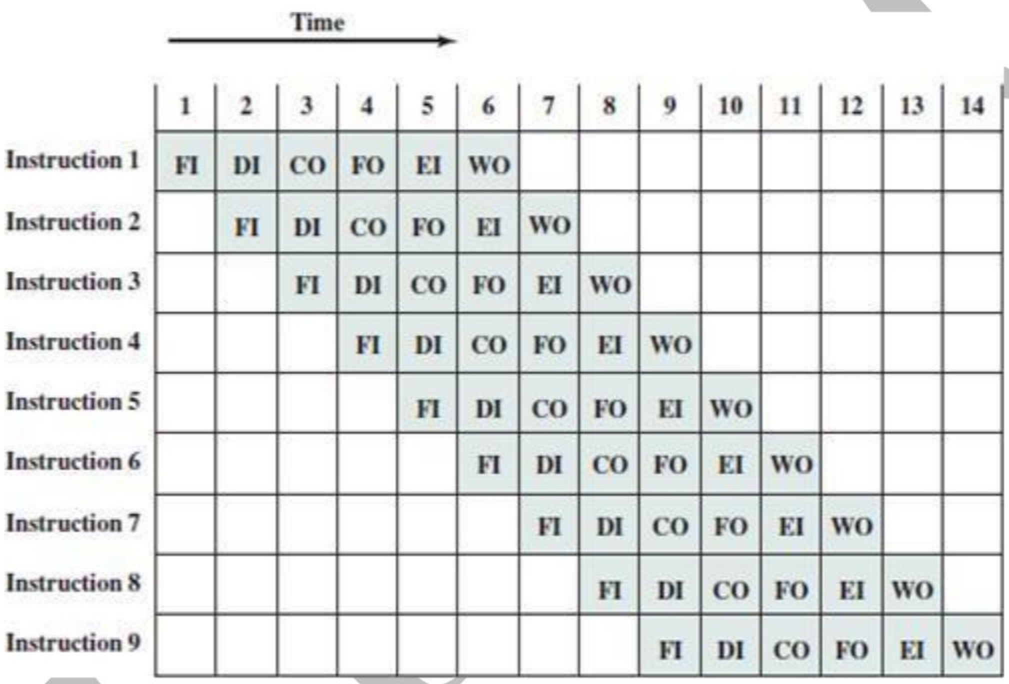
2. Decode instruction: The instruction is decoded by the CPU so that the necessary op codes and operands can be determined.

3. Calculate operand: Based on the addressing scheme used, either operands are directly provided in the instruction or the effective address has to be calculated.

4. Fetch Operand: Once the address is calculated, the operands need to be fetched from the address that was calculated. This is done in this phase.

5. Execute Instruction: The instruction can now be executed.

6. Write operand: Once the instruction is executed, the result from the execution needs to be stored or written back in the memory.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

PIPELINED DATAPATH AND CONTROL

The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle.

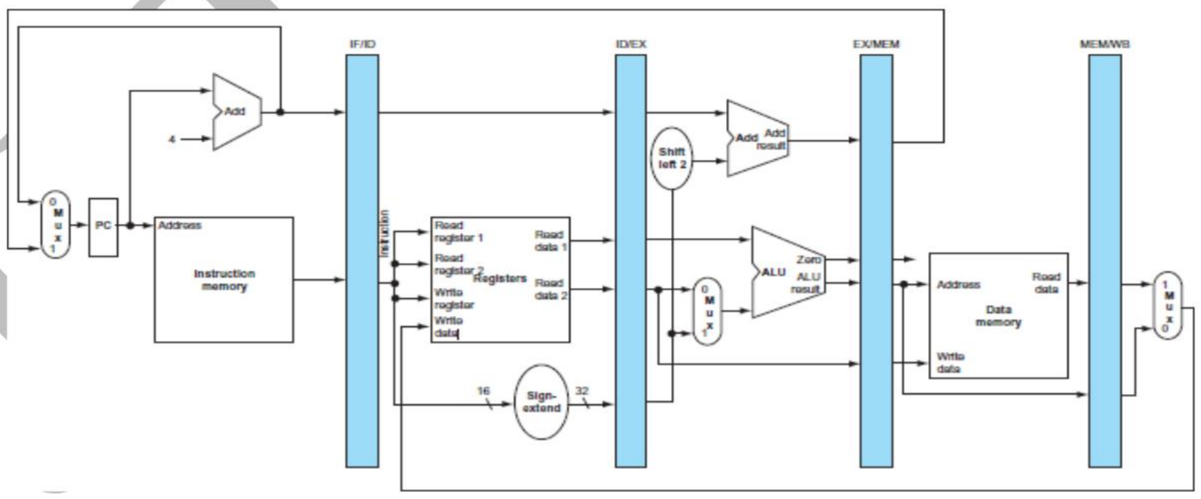
1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

- Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file.
- There are, however, two exceptions to this left -to-right flow of instructions:

1. The write-back stage, which places the result back into the register file in the middle of the datapath.
2. The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage Data fl owing from right to left does not affect the current instruction;

- The first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

- The following diagram shows the pipelined datapath with the pipeline registers highlighted.
- All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register.
- For example, the pipeline register between the IF and ID stages is called IF/ID.
- Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor, the register file, memory, or the PC.
- For example, a load instruction will place its result in 1 of the 32 registers, and any later instruction that needs that data will simply read the appropriate register.
- The pipeline registers separate each pipeline stage. They are labeled by the stages that they separate; For example, the first is labeled IF/ID because it separates the instruction fetch and instructions decode stages.
- The registers must be wide enough to store all the data corresponding to the lines that go through them.
- For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address.



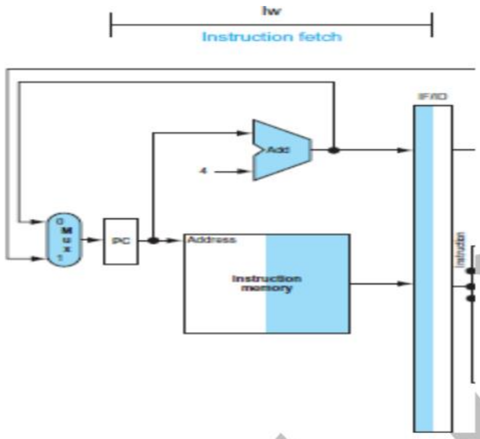
<u>Example: Load Instruction (lw)</u> lw $s1, 100($s0)

1. *<u>Instruction fetch:</u>*

- The top portion of Figure shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register.
- The PC address is incremented by 4 and then written back into the PC to be ready for

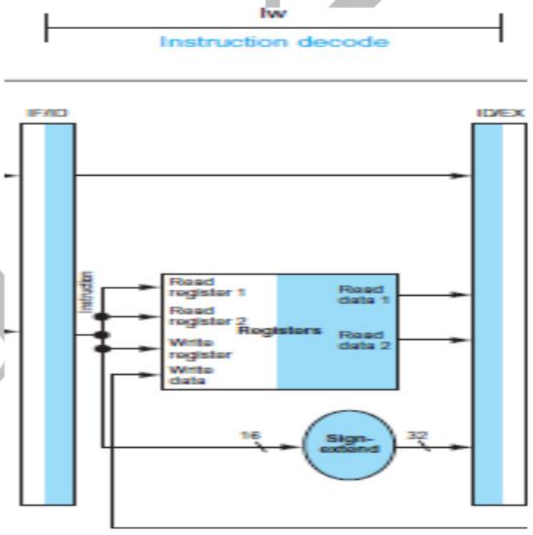
the next clock cycle.

- This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.
- The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.



2. *Instruction decode and register file read:*

- The bottom portion of Figure shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers.
- All three values are stored in the ID/EX pipeline register, along with the incremented PC address.
- We again transfer everything that might be needed by any instruction during a later clock cycle.



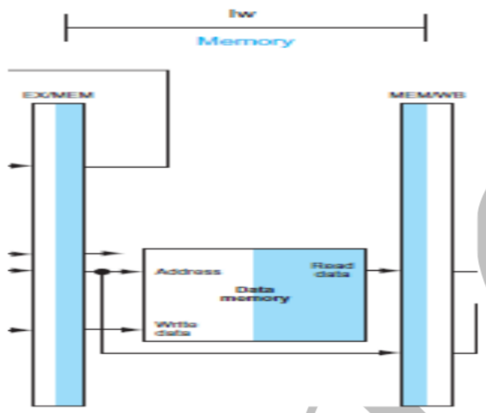
- There is no confusion when reading and writing registers, because the contents change only on the clock edge.
- Although the load needs only the top register in stage 2, the processor doesn't know what instruction is being decoded, so it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register.
- We don't need all three operands, but it simplifies control to keep all three.

3. *Execute or address calculation:*

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

4. ***Memory access:***

- The top portion of figure shows 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.



5. ***Write-back:***

- The bottom portion of figure shows the final step: Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register.
- Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath.

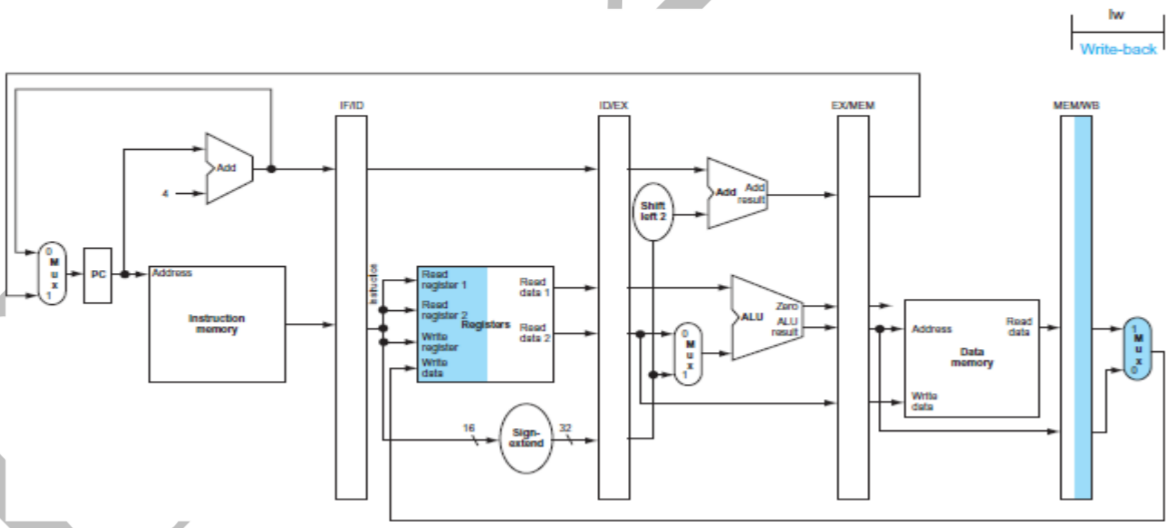
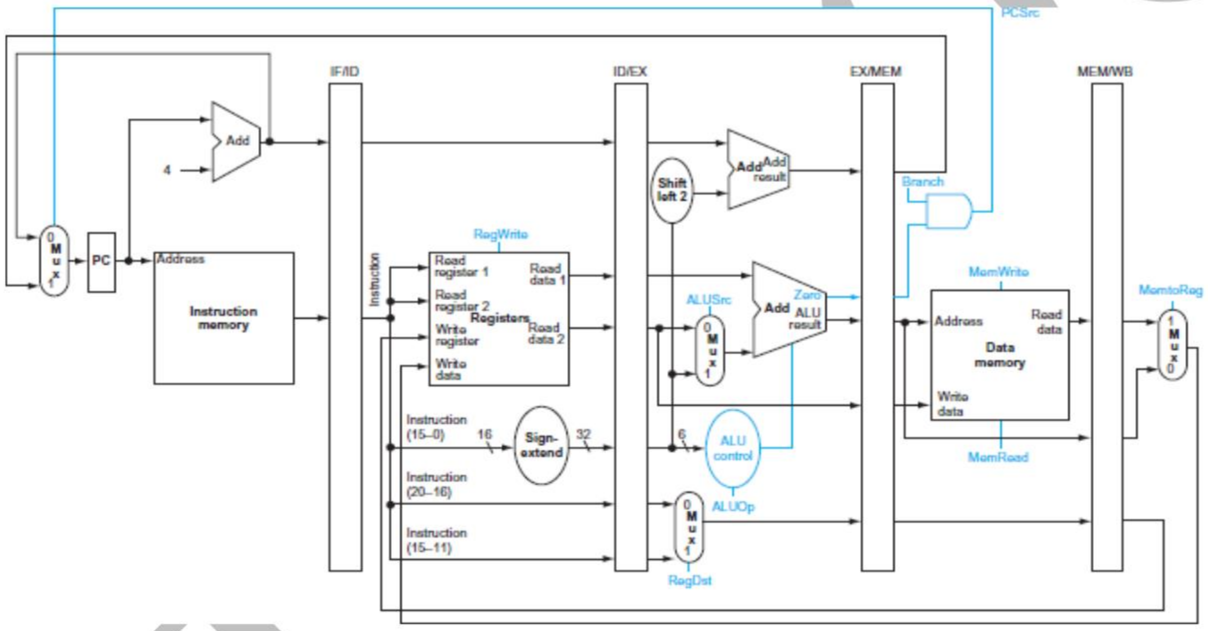


Figure: Combined Pipeline Datapath Diagram

PIPELINED CONTROL

- To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.
- ***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, the same thing happens at every clock cycle, so there are no optional control lines to set.
- ***Execution/address calculation:*** The signals to be set are RegDst, ALUOp, and ALUSrc . The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.



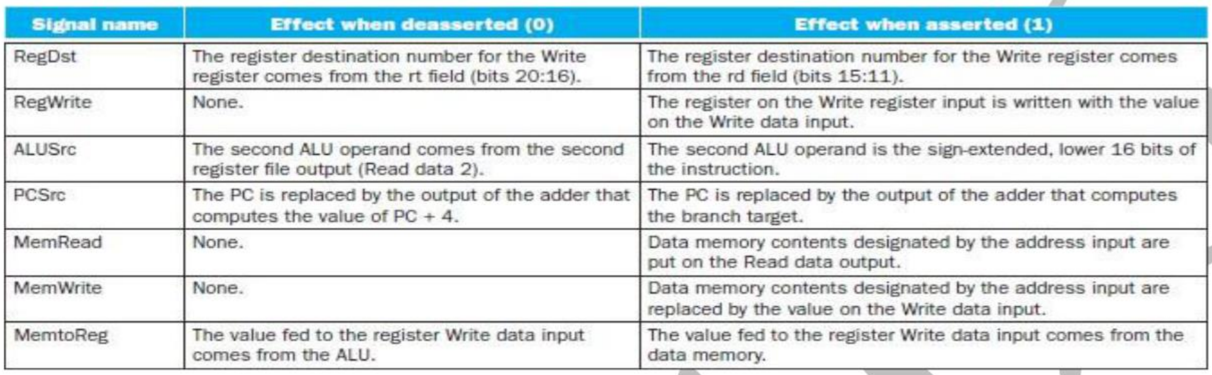
- This datapath borrows the control logic for PC source, register destination number, and ALU control. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.
- Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

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

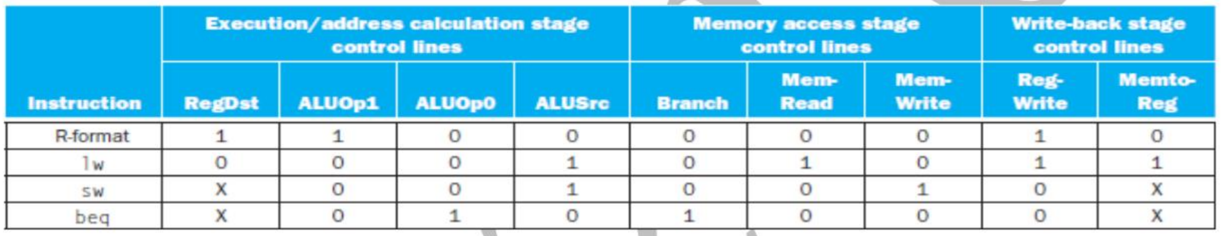
- The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column.

When a 1-bit control to a 2-way multiplexor is asserted, the multiplexor selects the input corresponding to 1.

- Otherwise, if the control is deasserted, the multiplexor selects the 0 input.
- Note that PCSrc is controlled by an AND gate in if the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

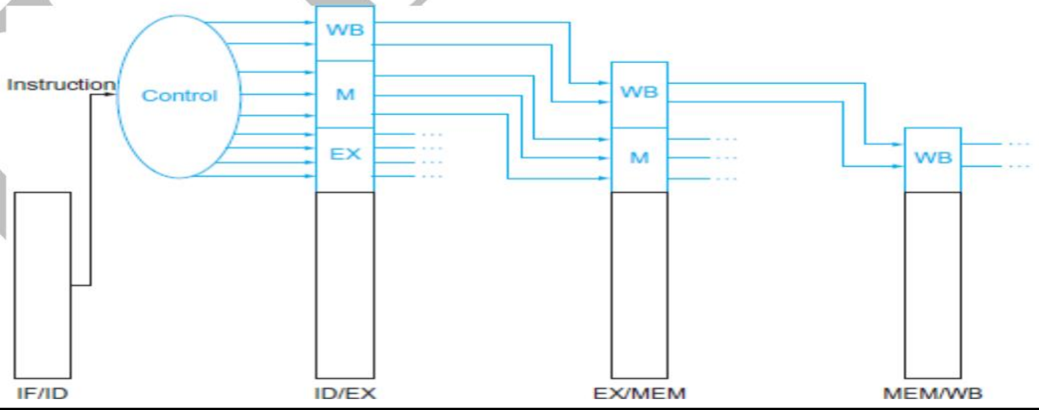


Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register 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.



	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

- **Memory access:** The control lines set in this stage are Branch, MemRead, and MemWrite. The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc selects the next sequential address unless control asserts Branch and the ALU result was 0.
- **Write-back:** The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and Reg-Write, which writes the chosen value.



- The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/ WB for use in the WB stage.

Pipeline Hazards

The condition that makes the pipeline to stall is called Hazards. The idle period in the pipeline execution is called Stall or Bubble.

Types of hazards:

1. Structural Hazard
2. Data Hazard
3. Control Hazard

1. Structural Hazard

- When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

2. Data Hazards

- **Data hazards** occur when the pipeline must be stalled because one step must wait for another to complete.
- When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- This is because of data dependence between the instructions that has been overlapped.

Consider the following example

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

- In the above instruction one of the operand ($s0) of the sub instruction will be fetched only after the add instruction store it result in the same register ($s0).
- So that sub instruction is stalled for some clock cycle which makes the pipeline process to waste the some clock cycle.

3. Control Hazards

- It is also called branch hazard. When the proper instruction cannot execute in the proper pipeline 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.

HANDLING DATA HAZARD:

Data hazard can be handled by using three methods.

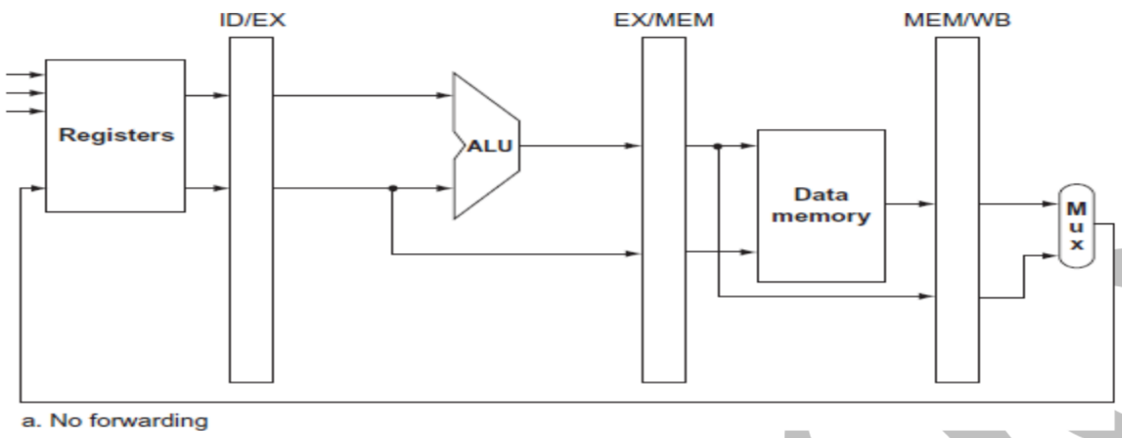
Solution to data hazard:

1. Operand forwarding(Hardware)
2. Reordering Code (software)
3. By using stall

1. Operand forwarding (Hardware):

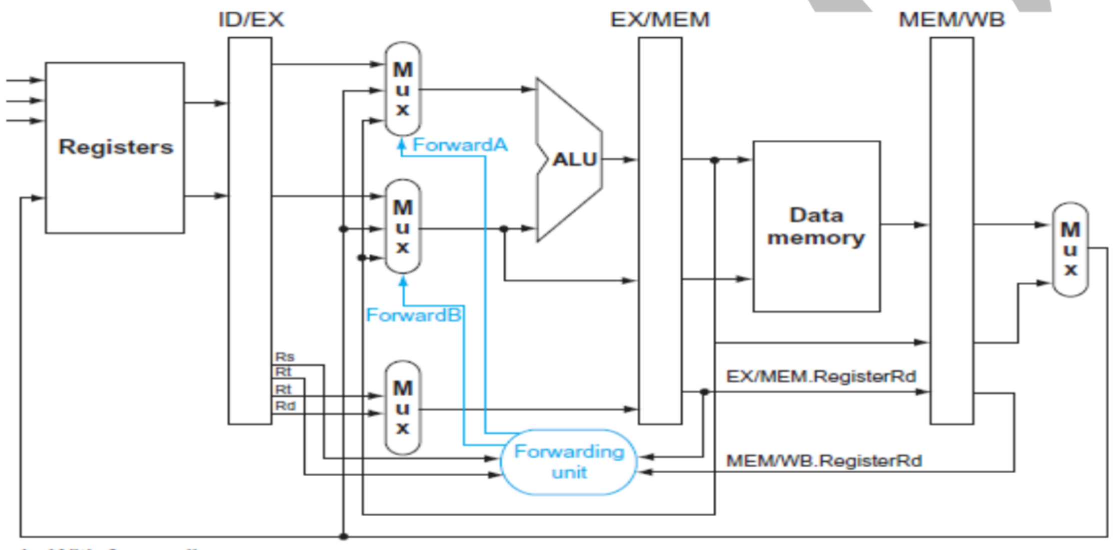
- The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard.
- Forwarding Also called **bypassing.** A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory.

No forwarding:



a. No forwarding

With Forwarding:



b. With forwarding

- On the top figure are the ALU and pipeline registers before adding forwarding. On the bottom figure, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit.
- The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware.
- Note that the ID/EX.Register Rt field is shown twice, once to connect to the Mux and once to the forwarding unit, but it is a single signal.

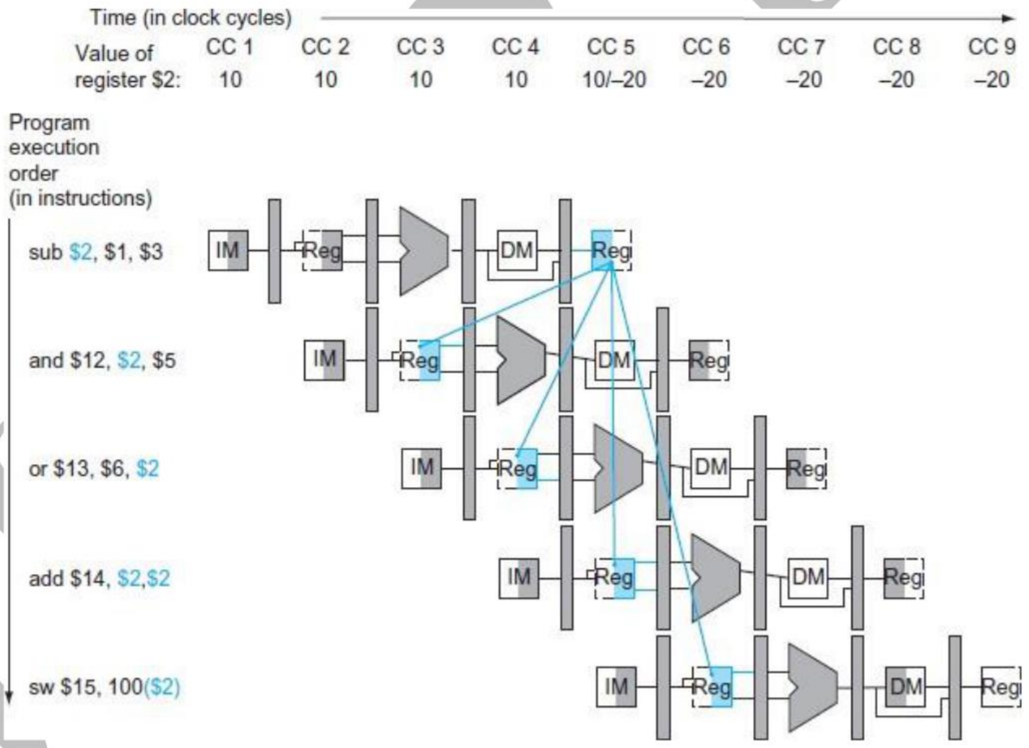
The control values for the forwarding multiplexors in the above diagram

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.

Pipeline datapath and control for data hazard:

- In the data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction.
- Something must stall the pipeline for the combination of load followed by an instruction that reads its result.
- Hence, in addition to a forwarding unit, we need a hazard detection unit. It operates during the ID stage so that it can insert the stall between the load and its use.
- Pipelined dependences in a five-instruction sequence using simplified data paths to show the dependences
- All the dependent actions are shown in color, and “CC 1” at the top of the figure means clock cycle 1.
- The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5.
- The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are pipeline data hazards.

Data Dependences without data forwarding Technique:



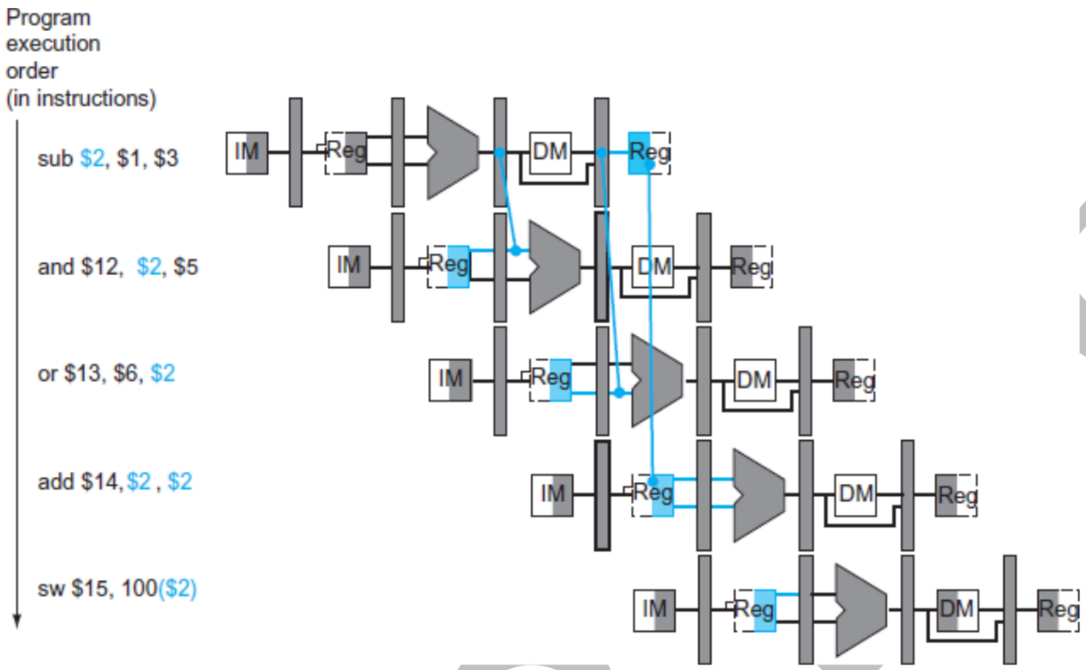
Time (in clock cycles)
CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9
Value of register $2: 10 10 10 10 10/–20 –20 –20 –20 –20
Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2,$2
sw $15, 100($2)
IM
Reg
DM

Data forwarding Technique:

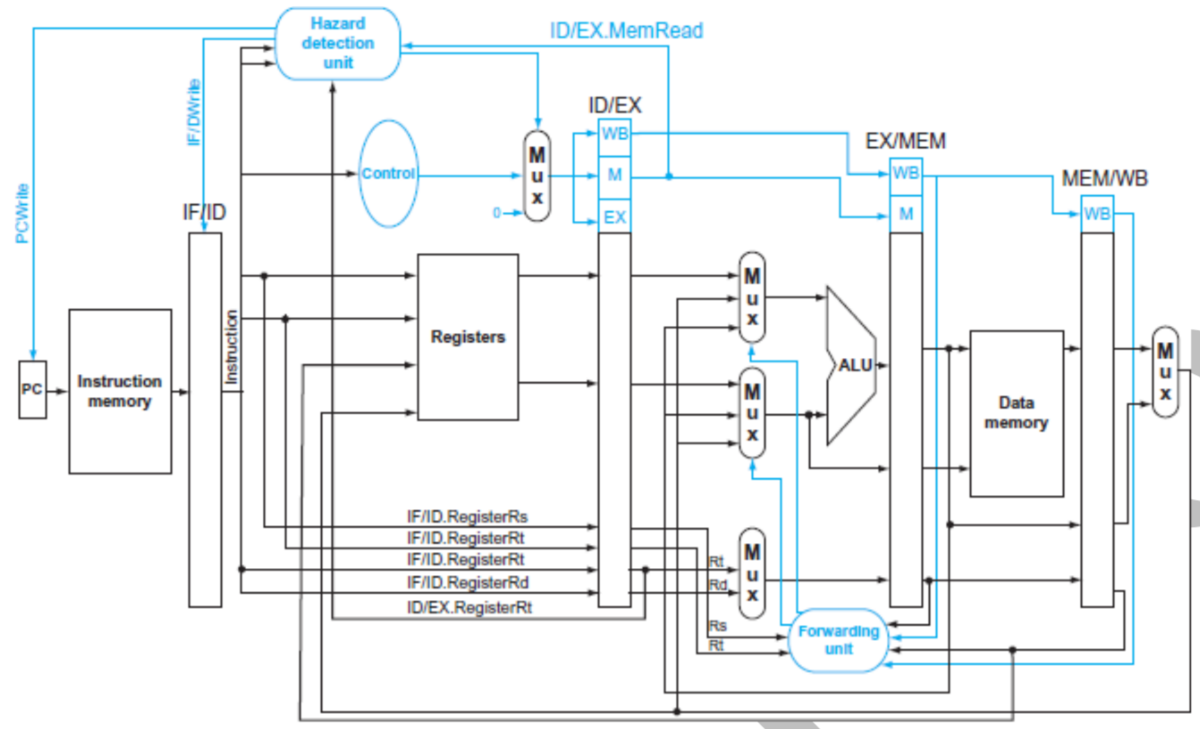
- The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction and OR instruction by forwarding the results found in the pipeline registers.
- The values in the pipeline registers show that the desired value is available before it is written into the register file.
- We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the

register file instead of a pipeline register.

- Register file "forwarding" that is, the read gets the value of the write in that clock cycle.



- The following diagram highlights the pipeline connections for both the hazard detection unit and the forwarding unit.
- As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register.
- The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s.
- The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true.



2. Reordering Code to Avoid Pipeline Stalls:

Consider the following code segment in C:

a = b + e;

c = b + f;

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as off sets from $t0:

Before Reorder:

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)

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.

- Both add instructions have a hazard because of their respective dependence on the immediately preceding lw instruction.
- Notice that bypassing eliminates several other potential hazards, including the dependence of the first adds on the first lw and any hazards for store instructions.
- Moving up the third lw instruction to become the third instruction eliminates both hazards:

After Reorder:

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)

3. Data hazard solved by using Stall

Pipeline stall

Pipeline stall also called bubble. A stall initiated in order to resolve a hazard.

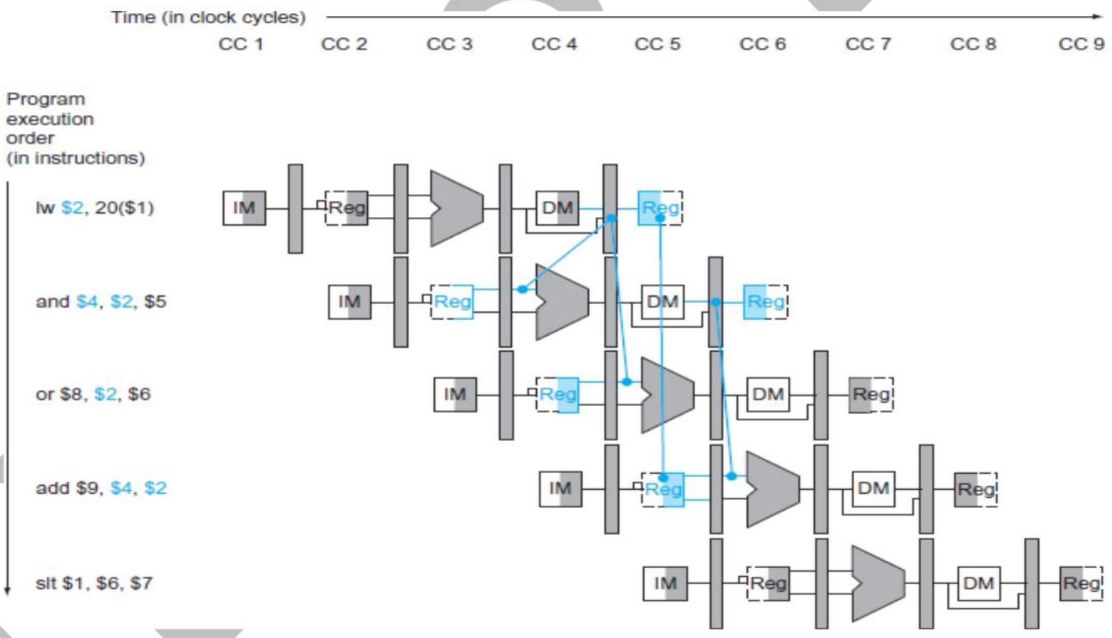
Load-use data hazard

A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

nop

An instruction that does no operation to change state.

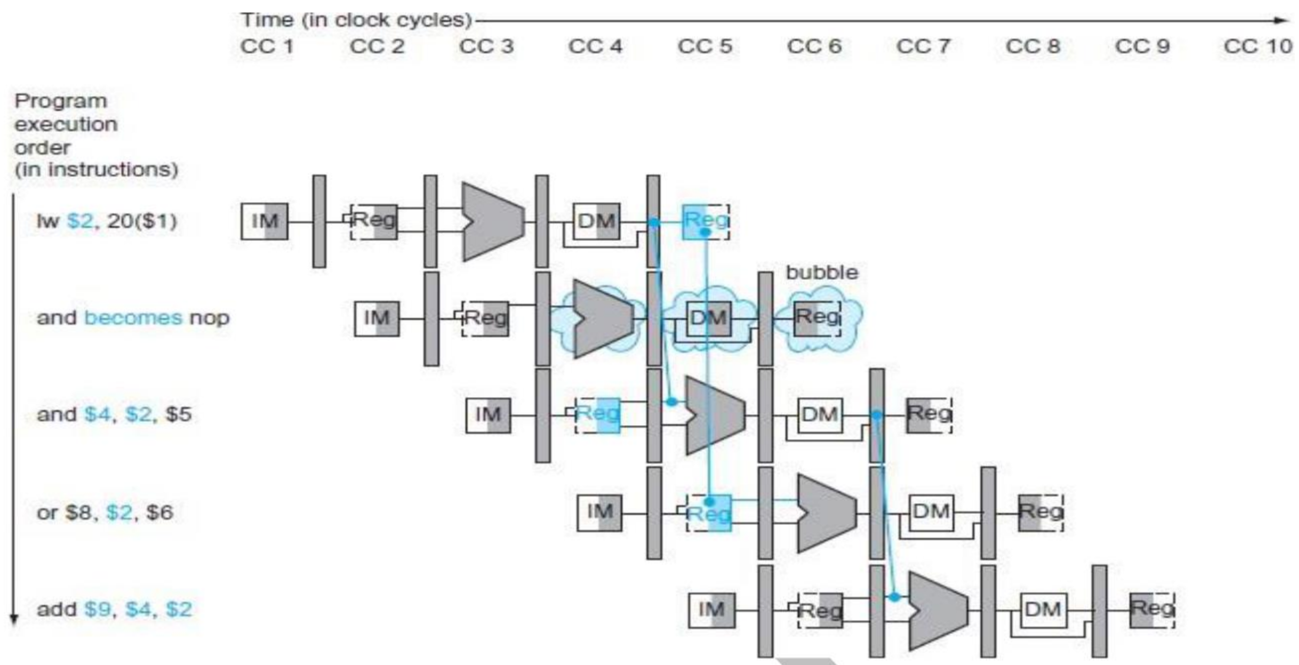
Data hazard without stall:



- The following diagram shows the AND instruction is turned into a nop and all instructions beginning with the AND instructions are delayed one cycle.
- In this example, the hazard forces the AND and OR instructions to repeat in clock cycle 4 what they did in clock cycle 3: AND reads registers and decodes, and OR is refetched from instruction memory.
- A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5.
- Likewise the OR instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5. After insertion of the bubble, all the dependences

go forward in time and no further hazards occur.

Data hazard with stall:

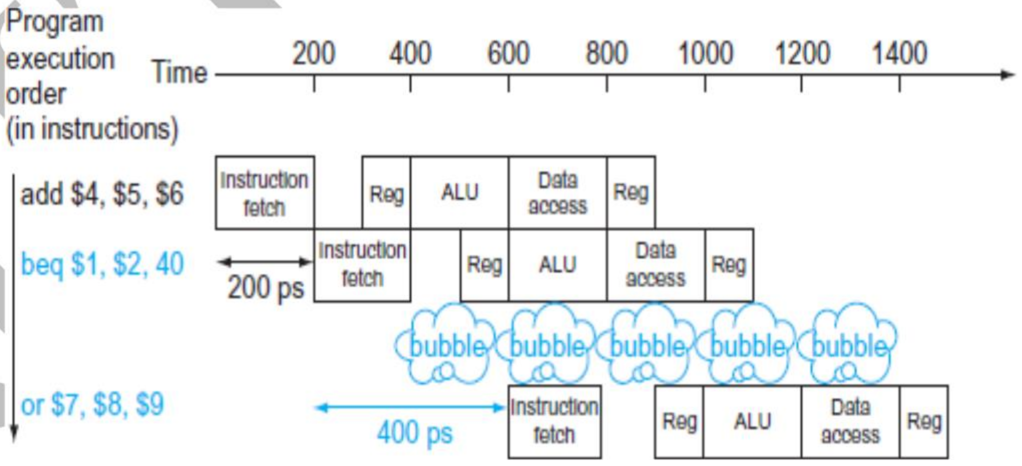


3. HANDLING CONTROL HAZARDS

- It is also called branch hazard. When the proper instruction cannot execute in the proper pipeline 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.

Performance of "Stall on Branch"

- Estimate the impact on the clock cycles per instruction (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.



- This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction.
- There is a one-stage pipeline stall, or bubble, after the branch.

Two schemes for resolving control hazards

1. Branch prediction
2. Delayed branching

1. Branch Prediction

- Prediction techniques can be used to check whether a branch will be valid or not valid. These techniques reduce the branch penalty.
- A method of resolving a branch hazard that assumes a given outcome for the branch called branch prediction.
- The common prediction techniques are:
  - Predict Never Taken
  - Predict Always Taken
  - Predict By Opcode
  - Taken or Not Taken Branch
  - Branch History Table
- In the first two approaches if prediction is wrong a page fault or prediction violation error occurs. The processor then halts prefetching and fetches the instruction from the desired address.
- In the third approach, the prediction is based on the opcode of the branch instruction.
- The fourth and Fifth approaches are dynamic. They depend on history of the previously executed conditional branch instruction.

Branch prediction Strategies:

(i). Static Branch Prediction Strategy

(ii). Dynamic Branch Prediction Strategy

(i). Static Branch Prediction

- In this strategy branch can be predicted based on branch code types statically. This means that the probability of branch with respect to a particular branch type is used to predict the branch. This branch strategy may not produce accurate results every time.
- One improvement over branch stalling is to predict that the branch will not be taken and thus continue execution down the sequential instruction stream.
- If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
- If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.
- Discarding instructions, then, means we must be able to flush instructions in the IF, ID, and EX stages of the pipeline.
- We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage;
- During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address.
- Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.

- For example, if an ALU instruction immediately preceding a branch produces one of the operands for the comparison in the branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch.

Example:

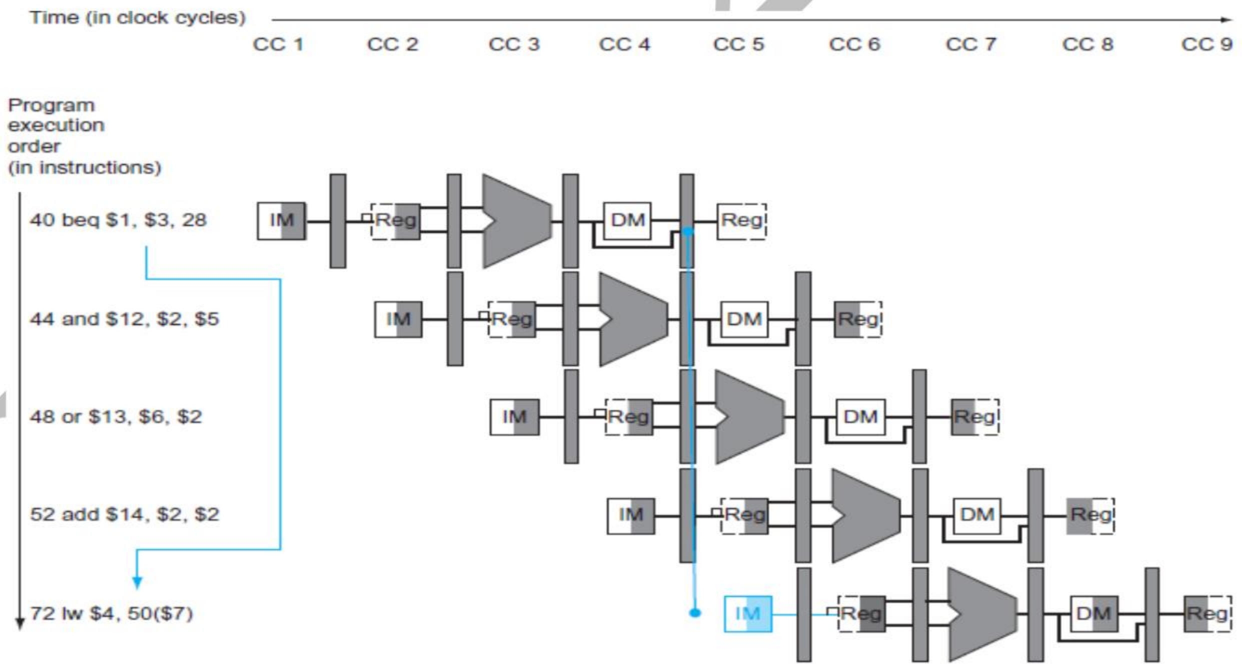
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40 + 4 + 7 * 4 = 72
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)

- The three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72.

- The number of stages in a pipeline or the number of stages between two instructions during execution.
- Pipelining does not reduce the time it takes to complete an individual instruction, also called the latency.

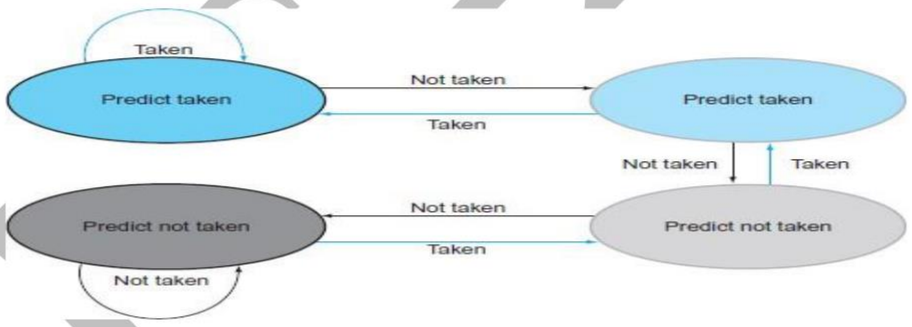
Flush

- To discard instructions in a pipeline, usually due to an unexpected event



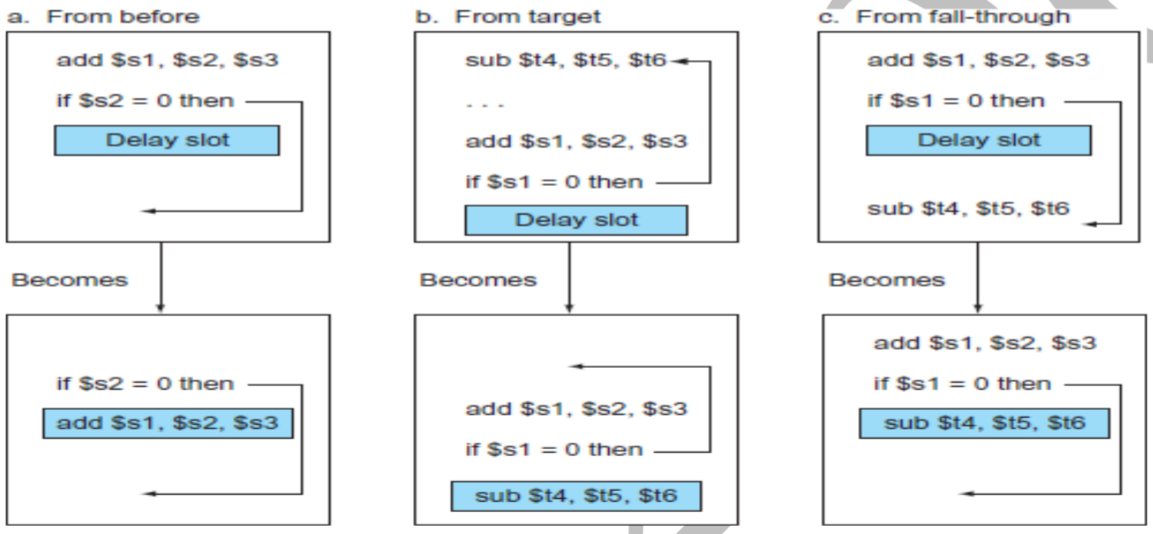
Dynamic Branch Prediction

- This strategy uses recent branch history during program execution to predict whether or not the branch will be taken next time when it occurs. It uses recent branch information to predict the next branch. This technique is called **dynamic branch prediction.**
- Prediction of branches at runtime using runtime information.
- **A branch prediction buffer** or **branch history table** is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.
- This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken.
- To remedy this weakness, 2-bit prediction schemes are often used. In a **2-bit scheme**, a prediction must be wrong twice before it is changed.
- The following diagram shows the finite-state machine for a 2-bit prediction scheme. A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage.
- If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; it can be as early as the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed.



2. Delayed branching

- The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.
- An instruction that always executes after the branch in the **branch delay slot**.
- The following figure shows the three ways in which the branch delay slot can be scheduled. The limitations on delayed branch scheduling arise from
  1. The restrictions on the instructions that are scheduled into the delay slots.
  2. Our ability to predict at compile time whether a branch is likely to be taken or not.
- Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle.



- The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code.
- In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice.
- Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of $s1 in the branch condition prevents the add instruction (whose destination is $s1) from being moved into the branch delay slot.
- In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path.
- Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c).
- To make this optimization legal for (b) or (c), it must be OK to execute the sub instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly.
- This is the case, for example, if $t4 were an unused temporary register when the branch goes in the unexpected direction.

Branch target buffer:

- A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

Correlating predictor:

- A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

Tournament branch predictor

- A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

EXCEPTIONS

- **Exceptions** and **interrupts** events other than branches or jumps that change the normal flow of instruction execution.

Exception

- Exception also called interrupt. An unscheduled event that disrupts program execution and they are used to detect overflow.
- The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow.

Interrupt

- It is an exception that comes from outside of the processor.
- We use the term *interrupt* only when the event is externally caused. Here are five examples showing whether the situation is internally generated by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Handling Exception:

- The two types of exceptions can occur in the basic MIPS architecture implementation.
  1. Execution of an undefined instruction
  2. An arithmetic overflow.

Response to an Exception:

- When an exception occurs the processor saves the address of the ending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.
- The operating system then takes the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error.
- After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.

<u>Two main methods used to communicate the reason for an exception:</u>

- The first method used in the MIPS architecture is to include a **status register** (called the Cause register), which holds a field that indicates the reason for the exception.
- A second method is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception.

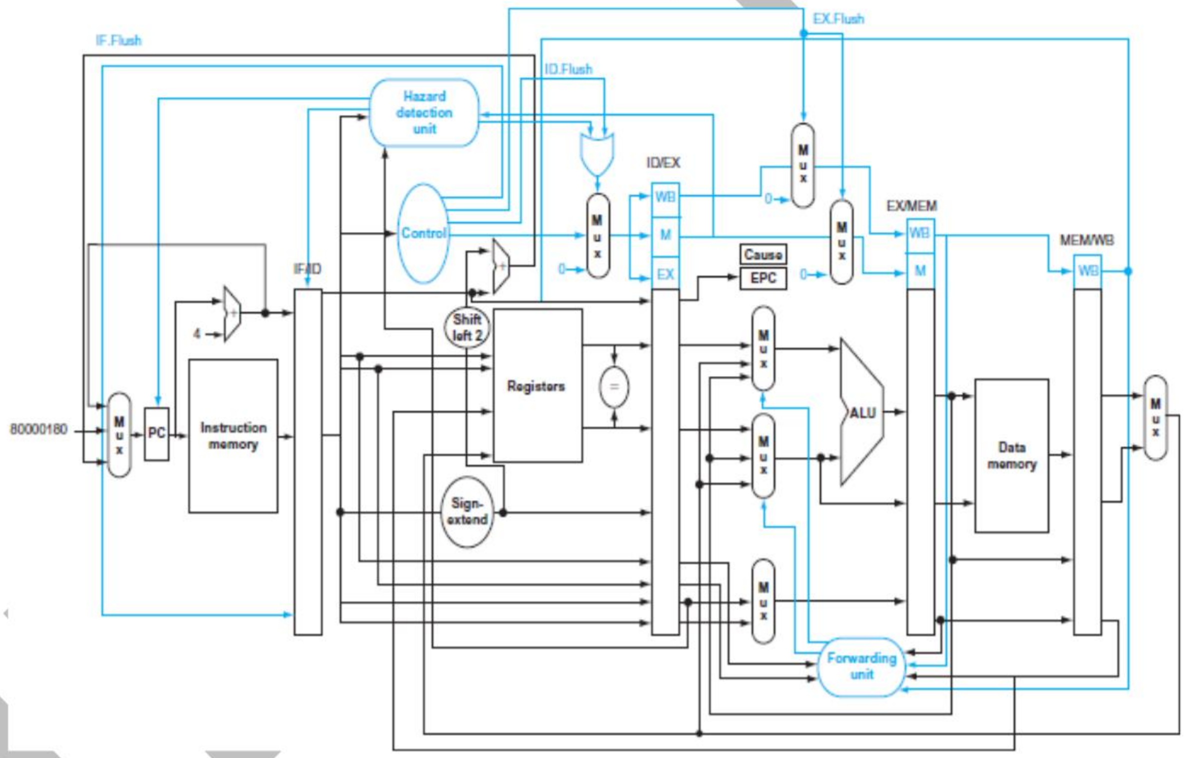
For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000$_{hex}$
Arithmetic overflow	8000 0180$_{hex}$

Add two additional registers to our current MIPS implementation:

- ***EPC:*** A 32-bit register used to hold the address of the affected instruction.
- ***Cause:*** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused.
- Assume there is a five-bit field that encodes the two possible exception sources mentioned above, with 10 representing an undefined instruction and 12 representing arithmetic overflow.

Exceptions in a Pipelined Implementation



<u>Imprecise interrupt</u>

- Imprecise interrupt also called imprecise exception. Interrupts or exceptions in pipelined computers that is not associated with the exact instruction that was the cause of the interrupt or exception.

<u>Precise interrupt</u>

- Precise interrupt also called precise exception. An interrupt or exception that is always associated with the correct instruction in pipelined computers.