UNIT III

Processor and Control

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

3.1 Basic MIPS Implementation

- MIPS have three kinds of core instruction:
    1. The arithmetic-logical instructions add, sub, and, or, and slt
    2. The memory-reference instructions load word (lw) and store word (sw)
    3. The branch instructions- branch equal (beq) and jump (j), which we add last
- To implement the three types we have same method,but independent of the exact class of instruction. 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 that we read two registers.

- These two steps are common for all instruction set.
- After these two steps the actions required to complete the instruction depend on the instruction class.
- The simplicity and regularity of the instruction set simplifies the implementation by making the execution of many instruction classes similar.
- For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.

- The memory-reference instructions use
    1. The ALU for an address calculation
    2. The arithmetic-logical instructions for the operation execution
    3. Branches for comparison.
- After using the ALU, the actions required to complete various instruction classes differ.
- A memory-reference instruction will need to access the memory either to write data for a store or read data for a load.
- An arithmetic-logical instruction must write the data from the ALU back into a register.
- Branch instruction need to change the next instruction address based on the comparison; otherwise the PC should be incremented by 4 to get the address of the next instruction.

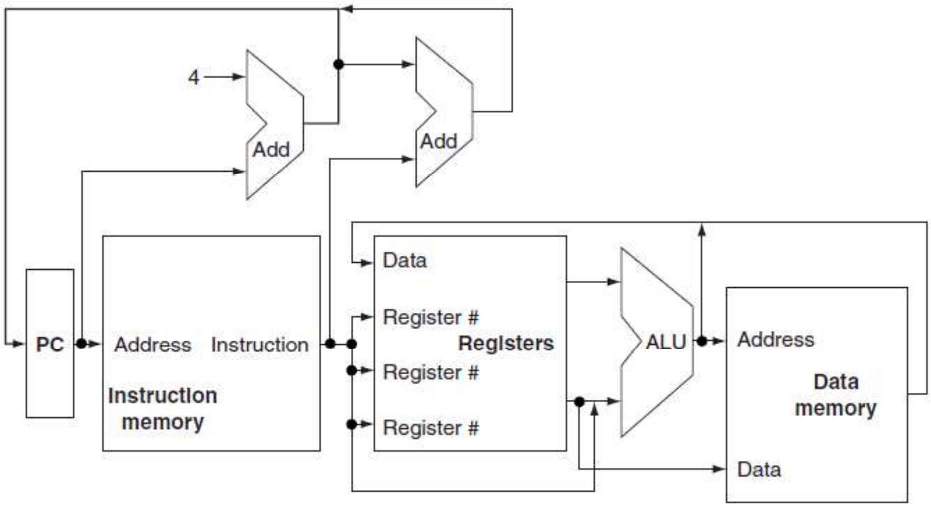


Figure: An abstract view for the implementation of the MIPS.

- It shows most flow of data through the processor but it omits two important aspects of instruction execution.
    1. In the below figure, it shows data going to a particular unit as coming from two different sources.
    2. Several units must be controlled depending on the type of instruction.

- 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, they can be operated to do the following tasks:
    1. To compute a memory address (for a load or store)
    2. To compute an arithmetic result (for an integer arithmetic-logical instruction)
    3. To compare for a branch.
- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
- If the operation is 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. Branch instruction require the use of the ALU output to determine the next instruction address, which comes from either the ALU or from an adder that increments the current PC by 4.
- The thick lines interconnecting the functional units represent buses, which consist of multiple signals.
- The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.
- It shows most of the data flow through the processor has two aspects for instruction execution:
- Above figure shows data going to a particular unit as coming from two sources.
- Several units must be controlled depending on the type of instruction.
- First, the value written into the PC can come from one of two adders, and the data written into the register file can come from either the ALU or the data memory.

- In practice, these data lines cannot simply be wired together; we must add an 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 multiplexer, although this device might better be called a **data selector.**
- The multiplexor is a combinational circuit which selects from among several inputs based on the setting of its control lines.
- The control lines are set based on information taken from the instruction being executed.
- Second, several of the units must be controlled depending on the type of instruction.
- For example, the data memory must read on a load and write on a store. The register file must be written on a load and an arithmetic-logical instruction.
- ALU must perform one of several operations.
- To overcome these problems we can use another circuit with some modification made in previous circuit.
- Compared to previous diagram, in this diagram we add three multiplexers and one major unit as control lines.

Control unit:

- A control unit that has the instruction as an input is used to determine how to set the control lines for the functional units and two of the multiplexors.

Function of third multiplexer:

- The third multiplexer, which determines whether PC + 4 or the branch destination address is written into the PC, is set based on the zero output of the ALU, which is used to perform the comparison of a beq instruction.
- The regularity and simplicity of the MIPS instruction set means that a simple decoding process can be used to determine how to set the control lines.

Function of multiplexer:

- The top multiplexer controls what value replaces the PC (PC + 4 or the branch destination address).

- The multiplexer is controlled by the gate that "ANDS” together with the Zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The middle multiplexer is used to direct the output of the ALU or the output of the data memory for writing into the register file.
- Finally, the bottommost multiplexer is used to determine whether the second ALU input is from the registers or from the offset field of the instruction.

Function of Control lines:

- The control lines are straightforward and determine the operation performed at the ALU.TheALU can perform following operations:
    1. Data memory read.
    2. Data memory writes.
    3. Write operation on registers.
- Control lines determine which operation is performed by the ALU. Control unit is used to control the actions taken by different instruction classes.

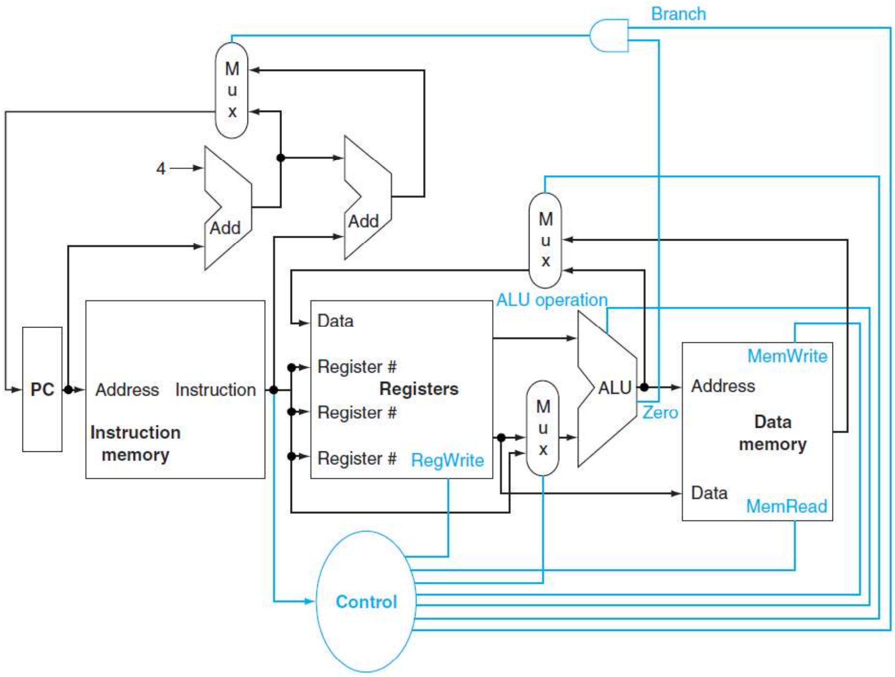


Figure: The basic implementation of the MIPS.

3.2 Building A Datapath

- To start a datapath design we must list the major components required to execute each class of MIPS instruction.
- Components required to form a data path is known as **datapath elements**.

Datapath element:

- A functional unit used to operate on or hold data within a processor.
- In the MIPS implementation the datapath elements include the instruction and data memories, the register file, the arithmetic logic unit (ALU), and adders.
- The state elements are the instruction memory and the program counter.
- The instruction memory need only provide read access because the data path does not write instructions.

Instruction memory:

- The instruction memory is called as combinational element because it will perform only read, the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed.

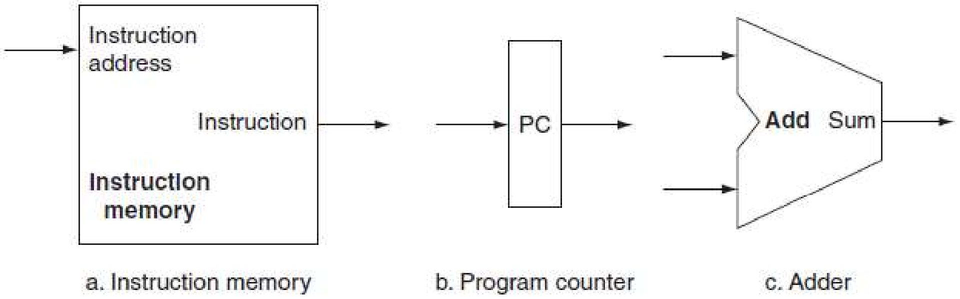


Figure: Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

Program counter (PC):

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

Adder:

- The adder is a combinational elementused to add two 32-bit inputs and place the result on its output.

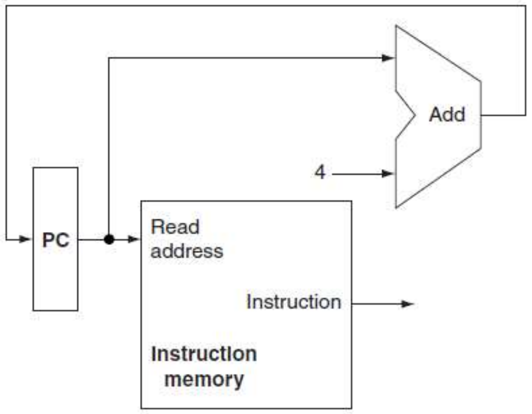


Figure : A portion of the datapath used for fetching instructions and incrementing the program counter.

Fetching phase:

- To execute any instruction, we must start by fetching the instruction from memory.
- To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction by incrementing the PC by 4 bytes.

3.2.1 Arithmetic logical instructions

- It is also called as R-format or R-type instructions.
- To perform an ALU operation these instructions read two registers and writes the result on one registers. It performs operations on the content of the registers.
- This instructions performs operations like add, sub, and, or, and slt.
- The processors having 32 general-purpose registers and some special purpose registers. General purpose and special purpose registers are stored in separate space of memory.

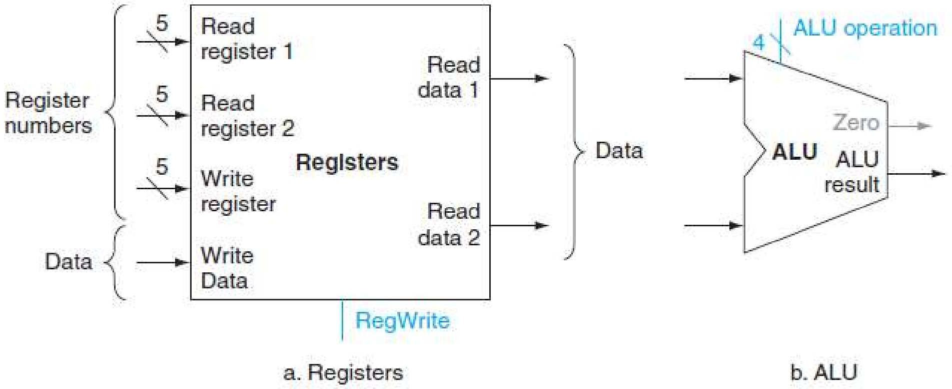
- The processor's 32 general-purpose registers are stored in a structure called a register file.

Register file:

- Register file is a state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.
- The register file contains the register state of the machine.

R-format instructions:

- The R-format instructions have three register operands to perform ALU operation.
- Two register data are read from the register file and write one data word into the register file for each instruction.
- Register number specifies which data as to be read from which register present in the register file.
- To write a data word, we will need two inputs:
    1. One to specify the register number to be written and
    2. One to supply the data to be written into the register.
- The register file always outputs the contents of whatever register numbers are on the Read register inputs. Write operation are controlled by the write control signal and it must be asserted for a write to occur at the clock edge.



a. Registers

b. ALU

Figure : The two elements needed to implement R-format ALU operations

- There are two elements needed to implement R-format ALU operations are:
    1. Register file
    2. ALU

Register file:

- The register file contains all the registers and has two read ports and one write port.
- The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs.
- Register write must be explicitly indicated by asserting the write control signal.
- Write operations are edge-triggered, so that all the write inputs must be valid at the clock edge.
- The register number inputs are 5 bits wide to specify one of 32 registers (32 = 2^5).
- The data input and two data output buses are each 32 bits wide.

ALU:

- The ALU takes two 32-bit inputs and produces a 32-bit result as well as a 1-bit signal if the result is 0.
- The operation to be performed by the ALU is controlled with the ALU operation signal and it is 4 bits wide.
- Zero detection output of the ALU is used to implement branches.

3.2.2 MIPS Instructions

- Consider the MIPS load word and store word instructions, which have the general form

  lw $t1,offset_value($t2)

  sw $t1,offset_value ($t2).

- These instructions will compute a memory address by adding the base register, which is $t2, to the 16-bit signed offset field contained in the instruction.

  Memory address=Base register + 16 bit signed offset field

- 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.
- Units need to implement load and store instruction are:
  1. Register file

2. ALU
3. Sign-extension Unit
4. Data memory Unit

Sign-extension Unit:

- 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.
- This unit will have 16-bit offset field in the instruction and extend to a 32-bit signed value.

Data memory Unit:

- The memory unit is a state element with inputs for the address and the write data, and a single output for theread result.
- There are separate read and write controls, although only one of these may be asserted on anygiven clock.
- The memory unit needs a read signal, since, unlike the register file, reading the value of aninvalid address can cause problems

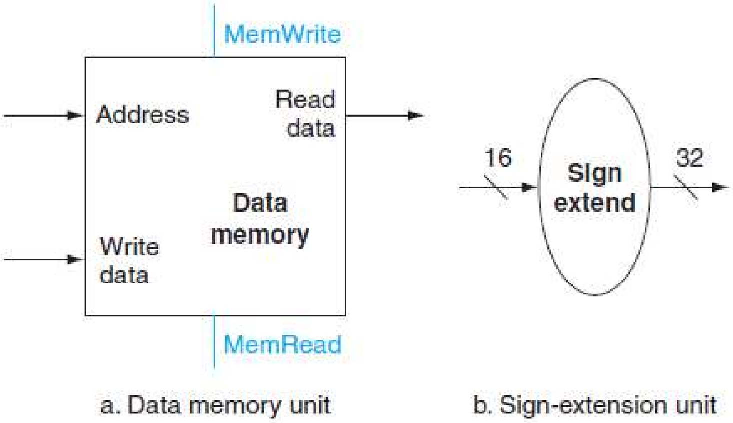


Figure: The two units needed to implement loads and stores

3.2.3 Branch Instruction

There are two types of Branch Instruction:

1. Branch taken
2. Branch not taken

Branch taken:

- If the branch condition is satisfied the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

Branch not taken:

- If the branch condition is false and the program counter(PC) becomes the address of the instruction that sequentially follows the branch.
- The beq instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the branch target addressrelative to the branch instruction address. The beq instruction has the form

  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.

Branch target address:

Branch target address = sign-extended offset field of the instruction + PC.

- The address specified in a branch, which becomes the new program counter (PC) if the branch is taken.

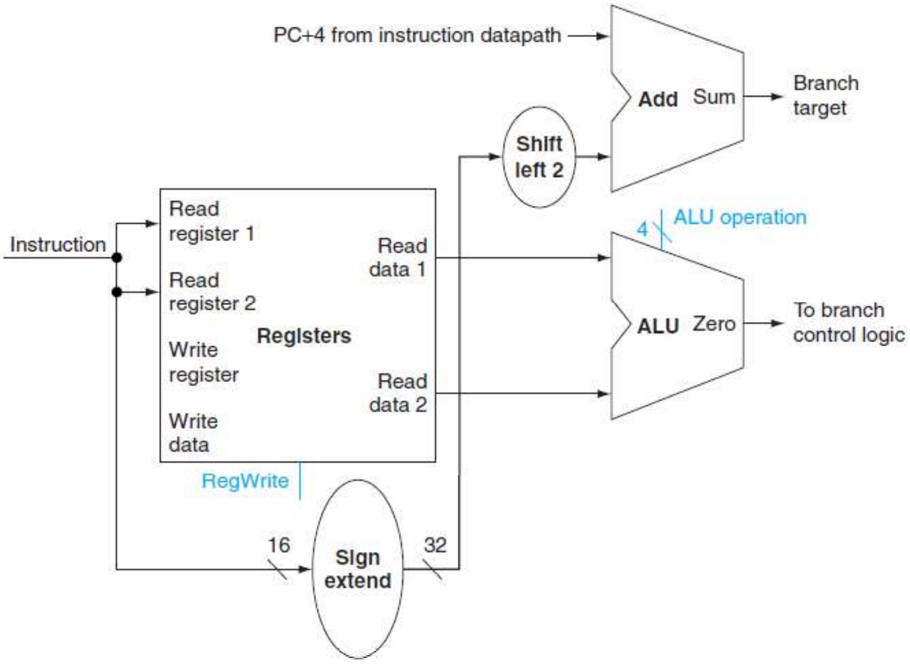


Figure : The data path for handling branch

- When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and it is called **branch istaken**.
- If the operands are not equal, the incremented PC should replace the current PC and it is called **branch** is **not taken**.
- Thus, the branch data path must do two operations: compute the branch target address and compare the register contents.
- To compute the branch target address, the branch data path includes a sign extension unit and an adder.
- To perform the compare, we need to use the register file.
- Adder circuit is used to compute the branch target and it is the sum of

- Incremented PC and the sign extended lower 16 bits of the instruction shifted left 2 units.
- The unit labeled Shift left 2 is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed because the amount of the shift is constant.
- Since we know that the offset was sign-extended from 16 bits and the shift will throw away only sign bits.
- 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.

3.2.4 Creating a Single Datapath

- The datapath components needed for the individual instruction classes are can combine and formed into a single datapath and add the control to complete the implementation.
- The single datapath will execute all instructions in one clock cycle.
- This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated.
- Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

- To share a datapath element between two different instruction classes we need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.
- The datapath from R-type and memory instructions, and the datapath for branches.
- The operations of arithmetic-logical (or R-type) instructions and the memory instructions data path are quite similar. The key differences are the following:
    1. The arithmetic-logical instructions use the ALU with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 16-bit offset field from the instruction.
    2. The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Example:1

To build a datapath for the operational portion of the memory reference and arithmetic-logical instructions that use a single register file and asingle ALU to handle both types of instructions, adding any necessary multiplexors.

Solution:

- To create a data path with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file.
- Thus, one multiplexer is placed at the ALU input and another at the data input to the register file.
- The branch instruction uses the main ALU for comparison of the register operands, so we use adder for computing the branch target address.
- An additional multiplexer is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.
- To complete this simple data path, we can add the control unit.
- The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexer, and the ALU control.

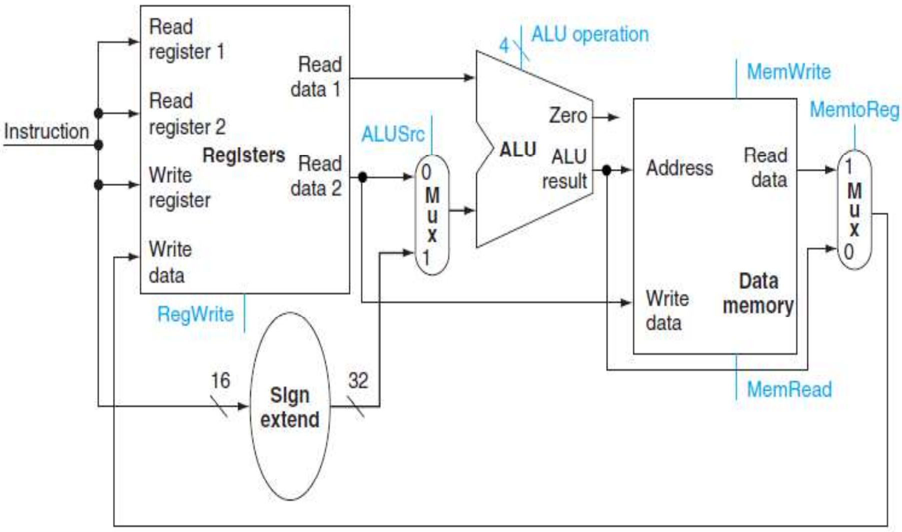


Figure: The datapath for the memory instructions and the R-type instructions.

- The ALU control is different in a number of ways so we must design the ALU first before we design the rest of the control unit.

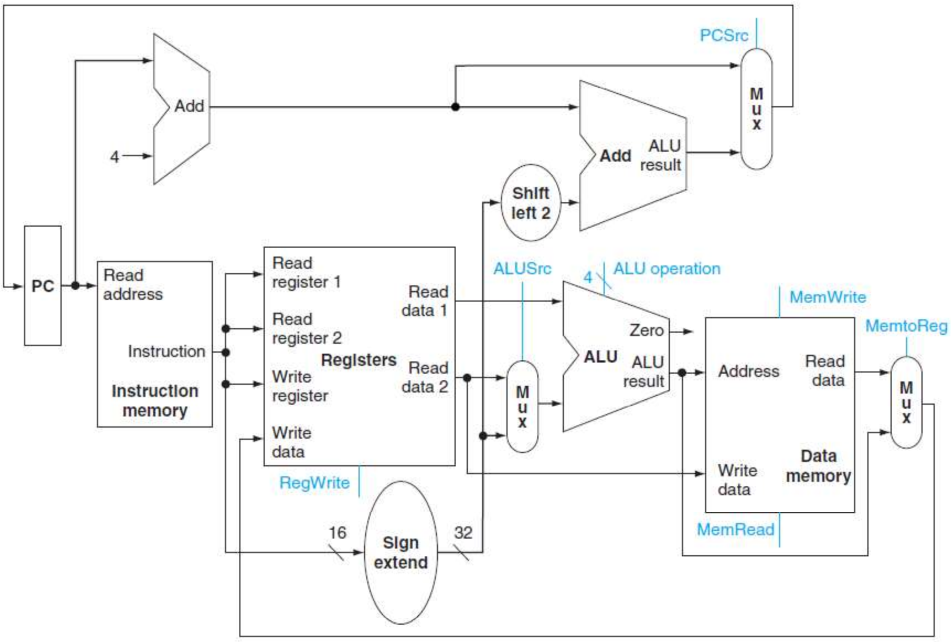


Figure:The simple datapath for the MIPS architecture combines the elements required by different instruction classes.

3.3. Control Implementation Scheme

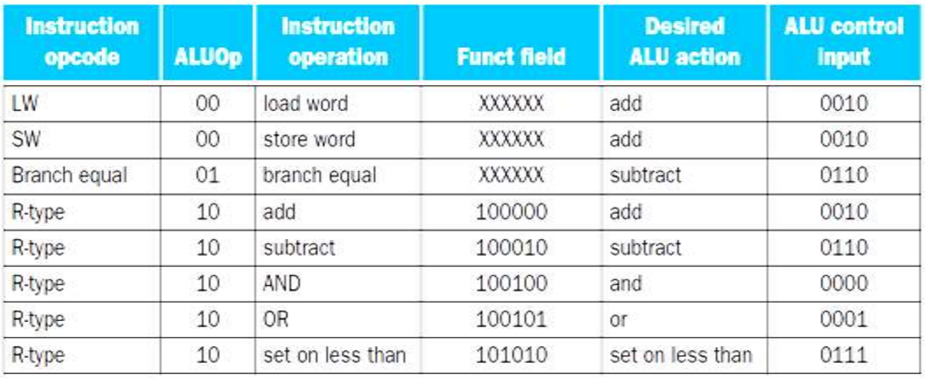
- Control implementation scheme can be build using data path and some simple control function.
- It covers load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, and, or, and set on less than.
- This implementation scheme covres the overall implementation of our MIPS subset.
- The MIPS ALU shows the 6 following combinations of four control inputs.

ALU control Lines	Function
0000	AND
0001	OR
0010	Add
0110	Sub
0111	Set on less than
1100	NOR

- Depending on the instruction class, the ALU will need to perform one of these first five functions.
- NOR is needed for other parts of the MIPS instruction set.
- For load word and store word instructions the ALU has 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 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.
- It has input function field of the instruction and a 2-bit control field it is called ALUOP.
- ALUOP indicates three kinds of operations

1. Add (00) for loads and stores
2. Subtract (01) for branch equal
3. Determined by the operation encoded in the function field (10).

- The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously.



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

- Instruction Opcode field determines the setting of the ALUOp bits. All the encodings are shown in binary.
- When the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field. We do not care about the value of the function code, and the function field is shown as XXXXXX for 00 or 01 values.
- When the ALUOp value is 10, then the function code is used to set the ALU control input.
- Here we are using multiple levels of decoding and it will provide the following functions.
- The main control unit generates the ALUOp bits
- ALUOp bits is used as input to the ALU control
- That ALU control generates the actual signals to control the ALU unit
- Using multiple levels of control can reduce the size of the main control unit.
- Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, since the control unit is often performance-critical.

Mapping 2-bit ALUOp field into 6-bit function field

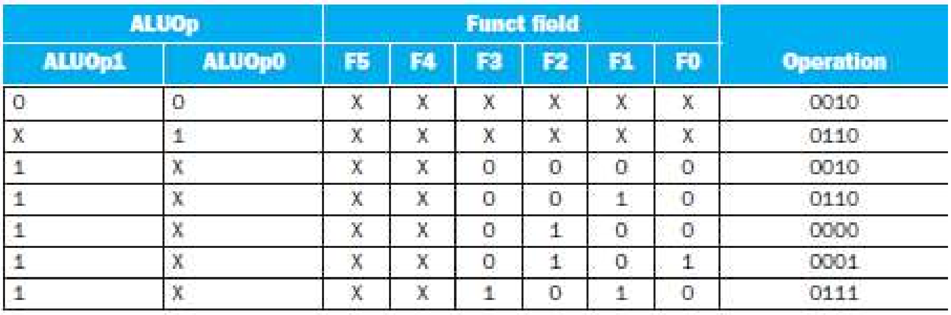
- There are several different ways to implement the mapping. From the 2-bit ALUOp field and the 6-bit function field to the four ALU operations control bits.
- There are 64 possible values are available for function field in that small values are used more frequently. The function field is used only when the ALUOp bits equal 10.
- We can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.
- To design a logic first we have to create a truth table for the function code field and the ALUOp bits.

Truth Table:

- It is a logical representation for operation by listing all the values of the inputs and showing what the resulting outputs should be.

Don't-care term:

- Don't-care term is 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 such as X or d.



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

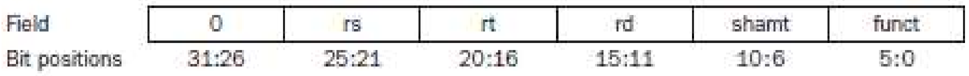
Figure: The truth table for ALU control bits

- The above truth table shows how the 3-bit ALU control is set depending on two input fields. The full truth table is very large (2^8 = 256 entries) and we don't care about the value of the ALU control for many of these input combinations.

- Table shows only the truth table entries for which the ALU control must have a specific value.
- For many instances we do not care about the values of some of the inputs and to keep the tables compact for that we include the don't-care terms (X).
- Once the truth table has been constructed, it can be optimized and then turned into gates.

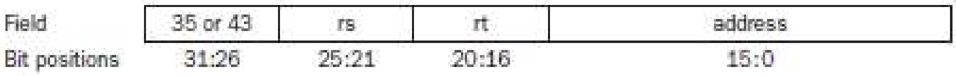
3.3.1 Designing Main Control Unit

- ALU control can be design using two ways one uses the function code and a 2-bit signal used as a control inputs.
- Now we can design main control unit for that we have to identify the fields of an instruction and the control lines.
- Control lines are needed for the data path construction. Various instruction fields are connected together to form a single data path.
- We used three instruction classes and it is important to know the format of it. Because then only we can obtain data path by connecting different instruction classes. Instruction formats of R-type, branch load and store instructions



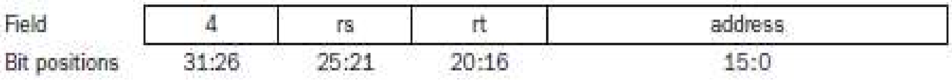
Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

R-type instruction



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

Load or store instruction



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

Branch instruction

Opcode:

- The field that denotes the operation and format of an instruction.

R-type instruction:

- An R-format instruction has opcode value of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources and rd is the destination.
- The ALU function is in the funct field and is decoded by the ALU control design.
- The R-type instructions ill support add, sub, and, or, and slt.
- The shamt field is used only for shifts.

Load or store instruction:

- Load or store instruction has opcode value as 35 or 43.
- The register rs are the base register 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.

Branch instruction:

- Branch instruction has opcode value as 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 to compute the branch target address.

There are several major observations about this instruction format such as:

1. The opcode field is always contained in bits 31:26. We will refer this field as Op[5:0].
2. 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 for store.
3. The base register for load and store instructions is always in bit positions 25:21 (rs).
4. The 16-bit offset for branch equal, load, and store is always in positions 15:0.

5. 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.
- Using this information, we can add the instruction labels and extra multiplexer to the simple data path.
- The below figure shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexers.

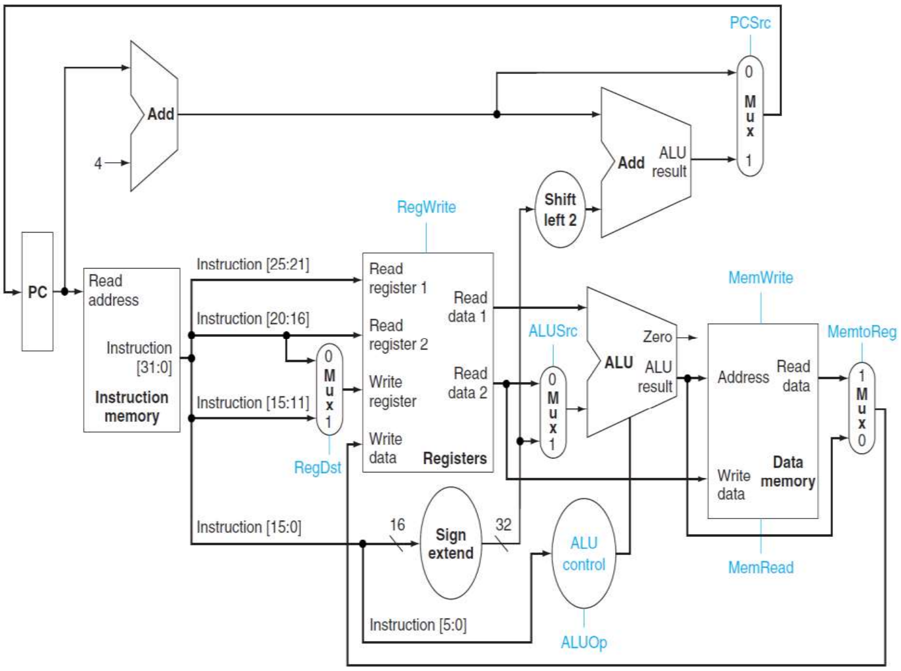
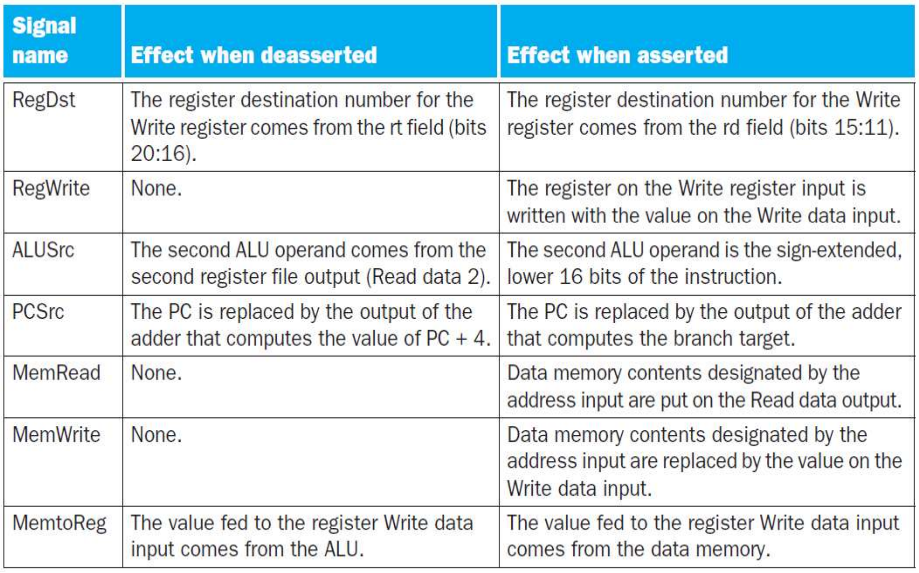


Figure: The data path with necessary multiplexors and all control lines

- Here all the multiplexers have two inputs; they each require a single control line.
- The PC does not require a write control, since it is written once at the end of every clock cycle.

- The branch control logic determines whether it is written with the incremented PC or the branch target address.
- Above figure shows seven single-bit control lines plus the 2-bit ALUOp control signal.
- We have already defined how the ALUOp control signal works so now we have to know the function of these seven control lines.



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.

Figure: The effect of seven control signals.

- These nine control signals can be set on the basis of six input signals to the control unit and it has opcode bits. 31 to 26.
- In the figure: The data path with the control unit. The input to the control unit is the 16-bit opcode field from the instruction.
- The outputs of the control unit consist of three 1-bit signals that are used to control multiplexers (RegDst, ALUSrc, and MemtoReg) and three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite).
- 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the 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.

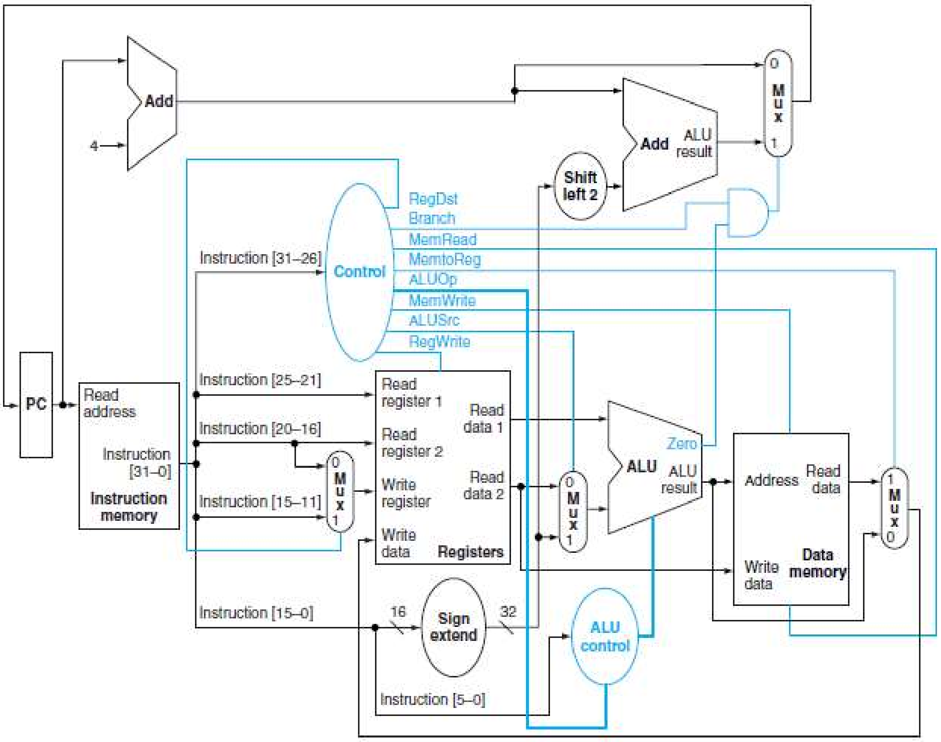


Figure: The data path with the control unit and the control signals.

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

R-type instruction:

- 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 signals ALUSrc and RegDst are set. R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory.T

- he ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field.

Load or store instruction:

- 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.

Branch instruction:

- The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU.
- 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 branch is set for a subtract (ALU control = 01), which is used to test for equality.
- The MemtoReg field is irrelevant when the RegWrite signal is 0.
- So, the register is not being written, the value of the data on the register data write port is not used.
- The MemtoReg in the last two rows of the table is replaced with X for don't care.
- Don't cares can be added to RegDst when RegWrite is 0.

3.3.2 Operation of the Data path

Consider three kinds of instruction classes are

1. R-type instruction
2. Load or store instruction
3. Branch instruction

- It is important to know the flow of control through the data path for these three different instruction classes.

R-type instruction data path:

- Figure shows the operation of the data path for an R-type instruction data path.
- In R-type instruction consider add $t1,$t2,$t3 and remaining four operations occurs in one clock cycle.

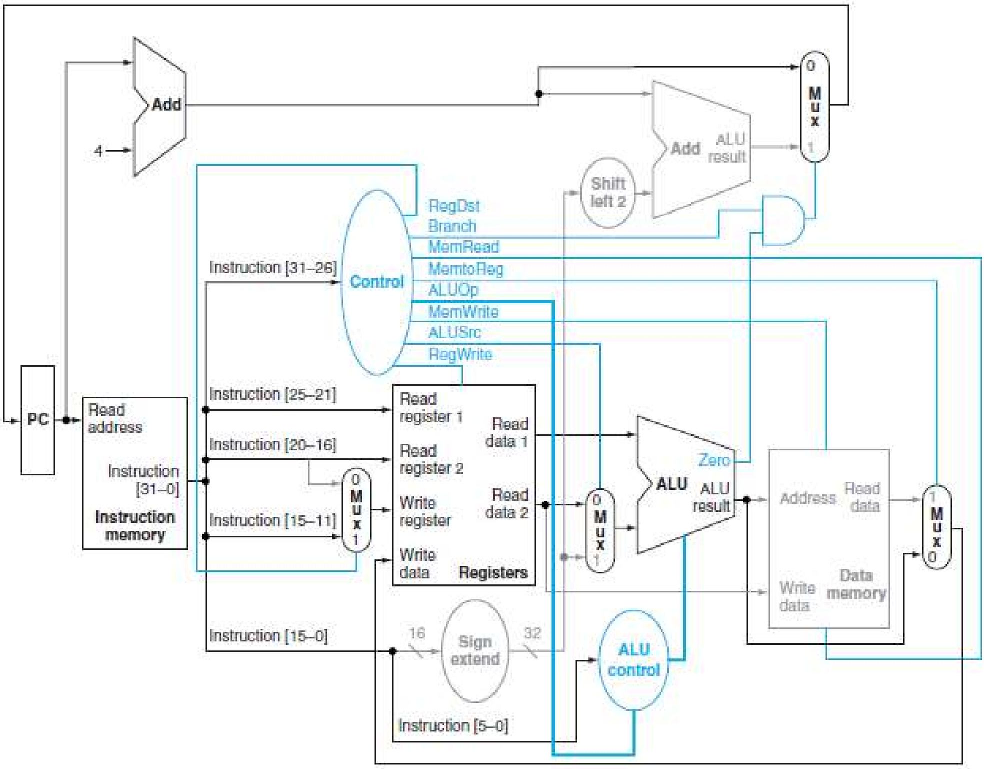


Figure: The data path in operation for an R-type instruction

- There are four steps used to execute this instruction and these steps are ordered by the flow of information:
  1. The instruction is fetched, and the PC is incremented.
  2. Two registers, $t2 and $t3, are read from the register file, and the main control unit computes the setting of the control lines during this step.
  3. The ALU operates on the data read from the register file using the function code to generate the ALU function.
  4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

Load instruction data path:

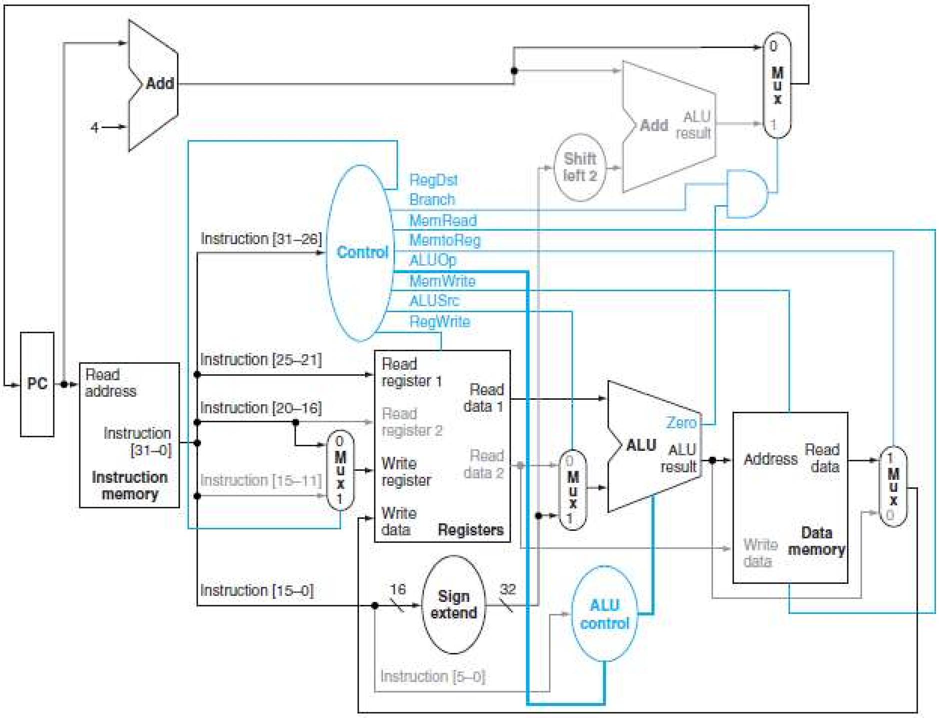


Figure: The data path in operation for a load instruction.

- There are five steps involved in execution of load instruction.
    1. An instruction is fetched from the instruction memory, and the PC is incremented.
    2. A register ($t2) value is read from the register file.
    3. The ALU computes the sum of the value read from the register file and the sign extended lower 16 bits of the instruction (offset).
    4. The sum from the ALU is used as the address for the data memory.
    5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1) .

Branch instruction data path:

- Branch-on-equal instruction has the following steps to execute a branch instruction such as
    - beq $t1,$t2,offset.
- It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.

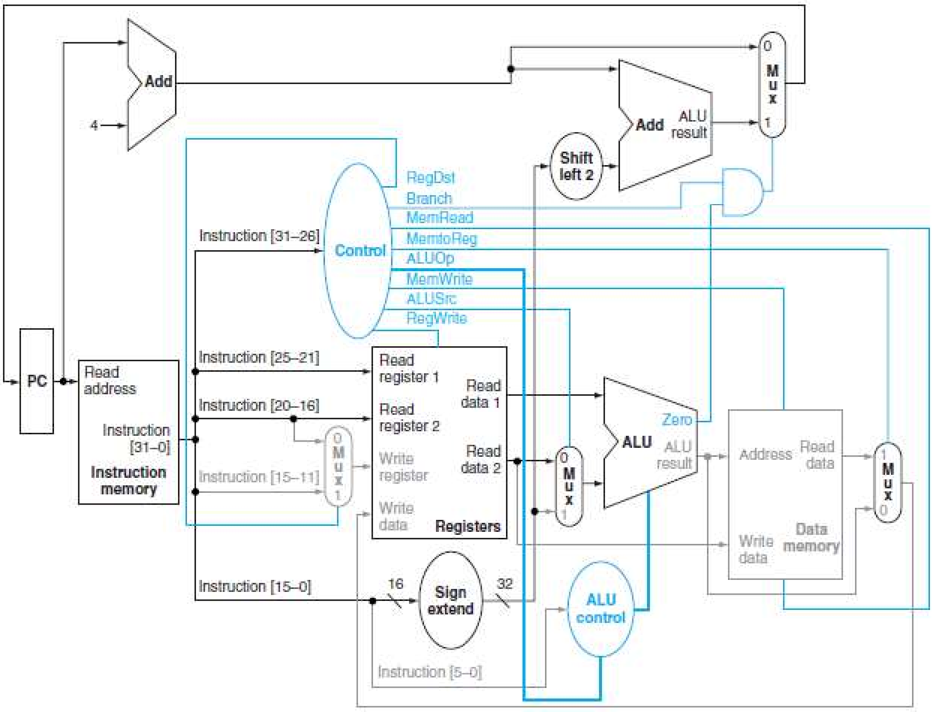


Figure: The data path in operation for a branch equal instruction.

- There are four steps in execution:
  1. An instruction is fetched from the instruction memory, and the PC is incremented.
  2. Two registers, $t1 and $t2, are read from the register file.
  3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
  4. The Zero result from the ALU is used to decide which adder result to store into the PC.

3.3.3 Single-cycle implementation

- It is also called single clock cycle implementation. It is an implementation in which an instruction is executed in one clock cycle.

- For example, consider jump instruction to show how the data path and control can be extended to handle other instruction in the instruction set.

Field	000010	address
Bit positions	31:26	25:0

Figure: Instruction format for the jump instruction

- The jump instruction looks somewhat like a branch instruction but computes the target PC differently and is not conditional.

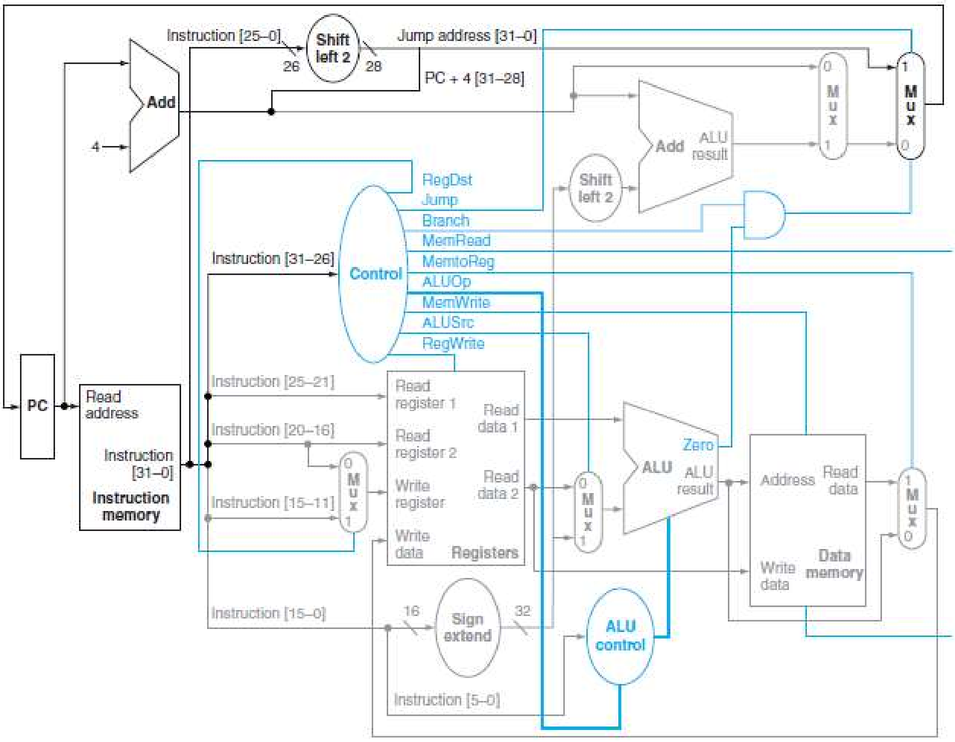


Figure: The simple control and data path are extended to handle the jump instruction.

- As like branch, the low order 2 bits of a jump address are always 00_{two}. The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction.
- 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

- The 26-bit immediate field of the jump instruction
- The bits 00_{two}

- An additional multiplexer is used to choose between the jump target and either the branch target or the sequential instruction following this one.
- This multiplexer 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, 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.
- One additional control signal is needed for the additional multiplexor. This control signal is called Jump. It is when the opcode is 2.

Why a Single-Cycle Implementation Is Not Used Today

- The single-cycle design is not used in modern designs because
  1. It is inefficient
  2. The clock cycle must have the same length for every instruction
  3. Overall performance is very poor because it has long clock cycle.

A Multi-cycle Implementation:

- It is also called multiple clock cycle implementations. It is an implementation in which an instruction is executed in multiple clock cycles.

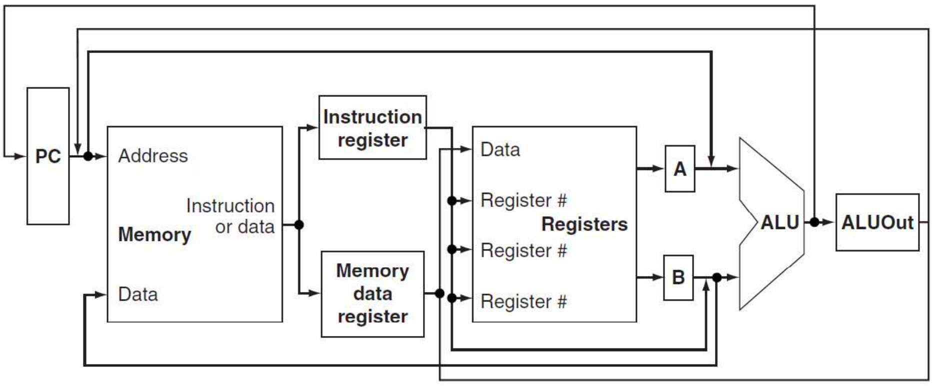


Figure: The high level view of the multi-cycle data path.

- In a multi-cycle implementation, each step in the execution will take 1 clock cycle.

- The multi-cycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles.
- This sharing can help to reduce the amount of hardware required.

Advantages:

1. It allows the instructions to take different numbers of clock cycles.
2. It share functional units within the execution of a single instruction.

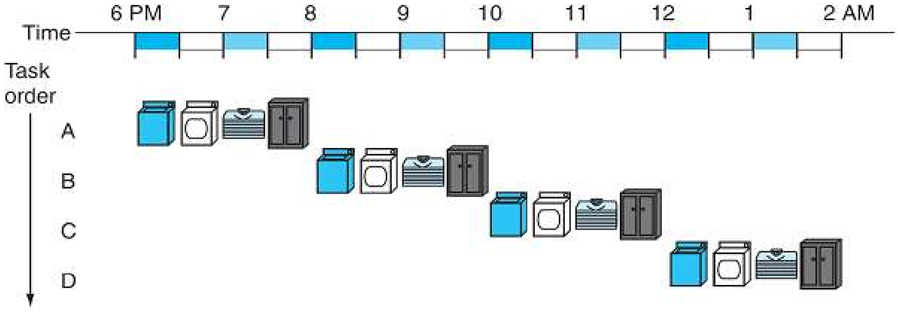
The major difference between the data path for the single -cycle over multi cycle are:

1. A single memory unit is used for both instructions and data.
2. There is a single ALU, rather than an ALU and two adders.
3. One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

3.4 Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- Pipelining is the most technique used to increase the speed and performance of the processor.
- Anyone who has done a lot of laundry has intuitively used pipelining.

Non- pipelined approach to laundry:



1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.

3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

Pipelined approach to laundry:

1. Place dirty load of clothes in the washer.
2. As soon as the washer is finished with the first load and placed in the dryer.
3. Load the washer with the second dirty load.
4. When the first load is dry then moves the wet load to the dryer and the next dirty load into the washer.
5. Next your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer.

- Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task.
- Sequential laundry takes 8 hours for four loads of wash, while pipelined laundry takes just 3.5 hours.To, pipelined laundry is potentially four times faster than non pipelined laundry process.

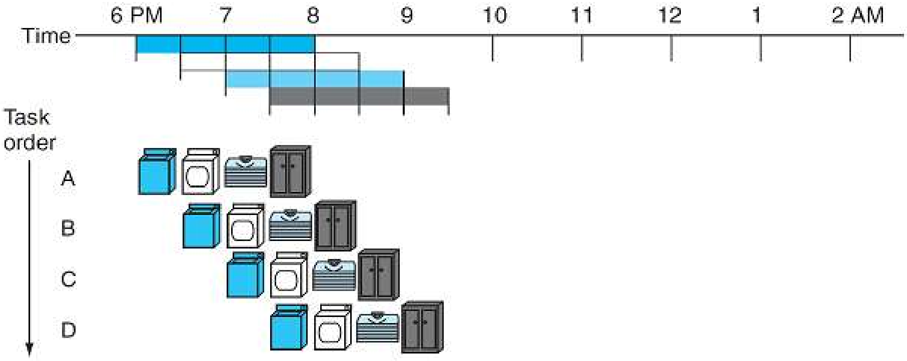


Figure 1 The laundry analogy for pipelining

Pipelining in processor:

To execute a MIPS instruction through the pipeline it takes five steps:

1. Fetch instruction from memory.

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

3.4.1 Single-Cycle versus Pipelined Performance

Consider three kinds of different instructions classes are

1. Load and store word instruction.
2. R-type instruction.
3. Branch instruction.

- From these three instruction classes we have eight instructions such as load word (lw), store word (sw), add (add), subtract (sub), and (and), or (or),set-less-than (slt), and ranch-on-equal (beq).
- For these eight instructions we have to find the average time taken to execute the instructions in both single clock cycle through pipelined implementation.

The operation times for the major functional units are:

- 200 ps for memory access,
- 200 ps for ALU operation, and
- 100 ps for register file read or write.

- In single clock cycle model every instruction takes exactly 1 clock cycle, so it will produce the slow speed to execute the instruction.
- **Figure** compares non pipelined and pipelined execution of three load word instructions.
- The time between the first and fourth instructions in the non pipelined design is 3 x 800 ns or 2400 ps.
- The time between the first and fourth instructions in the pipelined design is 3 x 200 ns or 600 ps.
- All the pipeline stages take a single clock cycle, so the clock cycle must belong enough to accommodate the slowest operation.

- The single cycle design must take the worst case clock cycle of 800 ps even though some instructions can be as fast as 500 ps.
- The pipelined execution clock cycle must have the worst case clock cycle of 200 ps even though some stages take only 100 ps.

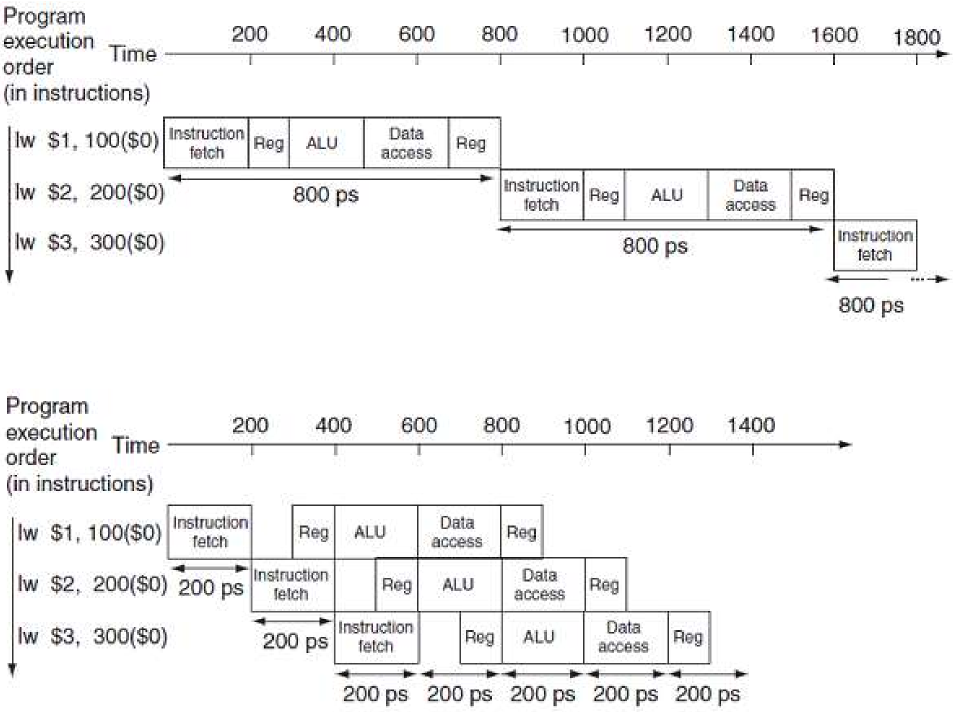
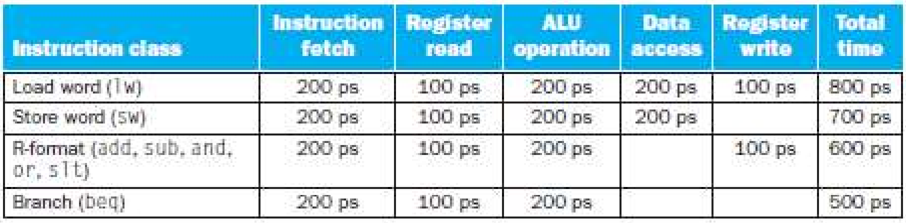


Figure: Single cycle pipelined execution.



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (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

- If the stages are perfectly balanced, then the time between instructions on the pipelined processor assuming ideal conditions is equal to

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

- Under ideal conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.
- The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps non pipelined time, or a 160 ps clock cycle. The example shows that the stages may be imperfectly balanced.

3.4.2 Designing Instruction Sets for Pipelining

- For designing instruction set for pipelining process we have to consider some important factor such as
    1. Length of the instruction
    2. Instruction format
    3. Memory operands
    4. Operand Alignment

- To explain the pipelining process we take the MIPS instruction set, so using these four factors we can design for pipelined execution.

Length of the instruction:

- All MIPS instructions are the same length. It makes much easy to fetch instructions in the first pipeline stage and to decode them in the second stage.
- Many instruction set has different length. An instruction set like the IA-32, instructions vary from 1 byte to 17 bytes, pipelining is considerably more challenging.
- To perform pipelining in IA-32 instructions we need to perform following task:
- IA-32 architecture translates IA-32 instructions into simple micro operations like MIPS instructions. Then the pipeline has micro operations rather than the native IA-32 instructions.

Instruction format:

- MIPS have only a few instruction formats. The source register fields being located in the same place in each instruction.

- If instruction format is symmetric then the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched.
- If instruction formats were not symmetric then we have to split second stage into two parts. It will increase the stages of pipelining process.

Memory operands:

- MIPS instruction set has only two memory operands (loads or stores). It can use the execute stage to calculate the memory address and then access memory in the following stage.
- In IA-32 instruction set, a stage 3 and 4 has expanded to an address stage, memory stage, and then execute stage.

Operand Alignment:

- Fourth, operands must be aligned in memory, Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

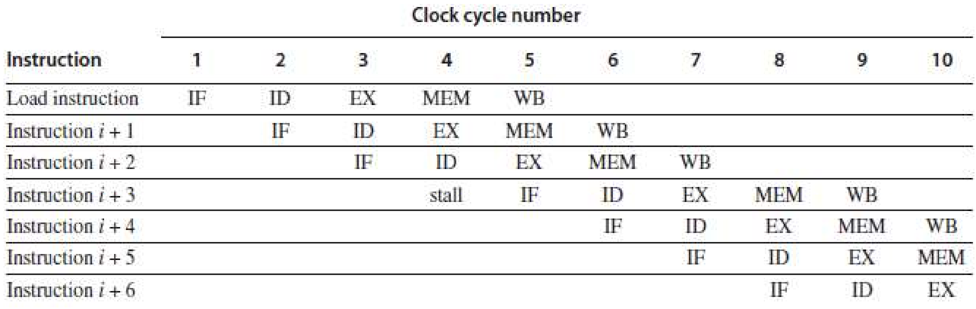
3.4.3 Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards. There are three types of hazards:
    1. Structural hazards
    2. Data hazards
    3. Control hazards

Structural hazards:

- The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.
- A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer.
- If our roommate are doing something else and without put clothes away.

- In the same clock cycle the first instruction is accessing data from memory and the fourth instruction is fetching an instruction from that same memory. So it cause structural hazard.



	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction *i* + 1		IF	ID	EX	MEM	WB				
Instruction *i* + 2			IF	ID	EX	MEM	WB			
Instruction *i* + 3				stall	IF	ID	EX	MEM	WB	
Instruction *i* + 4						IF	ID	EX	MEM	WB
Instruction *i* + 5							IF	ID	EX	MEM
Instruction *i* + 6								IF	ID	EX

- The load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall, no instruction is initiated on clock cycle 4 (which normally would initiate instruction *i* + 3).
- Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally.
- The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8.
- Sometimes these pipeline diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction *i* + 3 does not begin execution until cycle 5.

Data Hazard:

- Data hazard occurred when a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- In pipeline process if one step must wait for another to complete means it cause data hazards.
- For example, an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

add $s0, $t0, $t1

sub $t2, $s0, $t3

- Subtract instruction has to wait until the add instruction is executed. Because it has to get $s0 value from the add instruction itself.
- Without intervention, a data hazard could severely stall the pipeline. The add instruction does not write its result until the fifth stage, meaning that we would have to add three bubbles to the pipeline.
- In fifth clock cycle, only add write the result but subtract has to read the value in second clock cycle itself.

Solution for Data Hazard:

- 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.
- For the code above as soon as the ALU creates the sum for the add instruction we can supply it as an input for the subtract.
- Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing.

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.

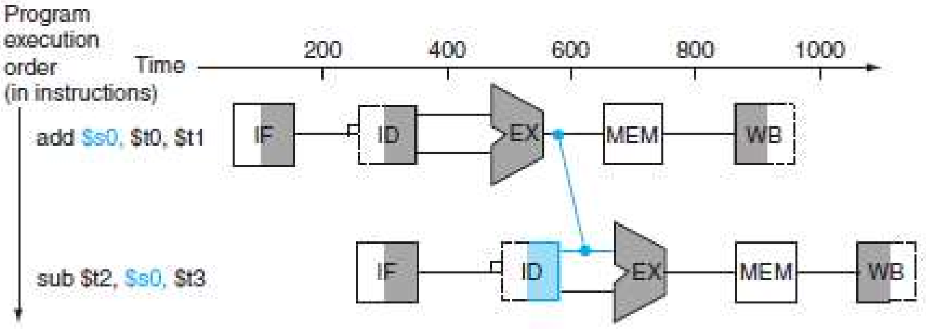


Figure: Graphical Representation of Forwarding.

- MIPS instruction set has five stages for pipelining process:

  IF - instruction fetch stage

  ID - instruction decode/register file read stage,

  EX - execution stage

  MEM - memory access stage,

  WB -the write back stage

- The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub.
- Forwarding paths are valid only if the destination stage is later in time than the source stage.

Load-use data hazard:

- A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested.

Pipeline stall:

- It is also called bubble. A stall initiated in order to resolve a hazard.

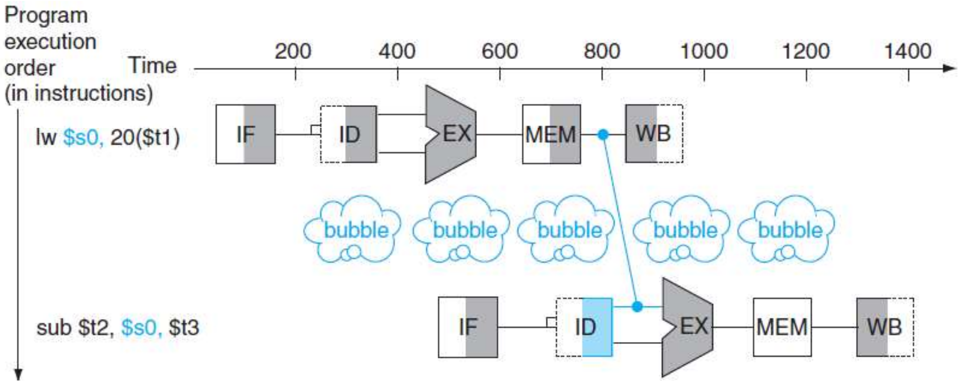


Figure needs a stall even with forwarding when a load tries to use the data.

- Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible.

- This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary.

Reordering Code to Avoid Pipeline Stalls

Example:1

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 offsets from $t0:

```
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 following code segment and reorder the instructions to avoid any pipeline stalls.

Solution:

- Both add instructions have a hazard because of their respective dependence on the immediately preceding lw instruction.
- By passing eliminates several other hazards including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction eliminates both hazards.

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

add $t5, $t1,$t4

sw $t5, 16($t0)

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

Control Hazards: (branch hazards)

- The third type of hazard is called a **control hazard**, arising from the need to make a decision based on the results of one instruction while others are executing. Also called **branch hazard**.
- When the instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed.
- That is the flow of instruction addresses is not what the pipeline expected. Consider the branch instruction; we must begin with fetching the instruction following the branch on the very clock cycle.

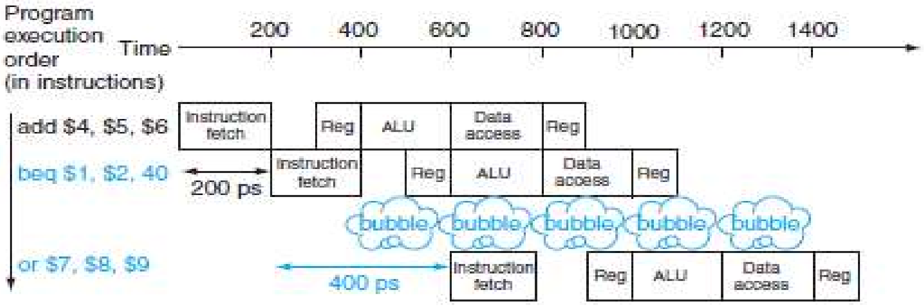


Figure showing stalling on every conditional branch as solution to control hazards.

- The pipeline cannot possibly know what the next instruction should be, because it only received the branch instruction from memory.
- To avoid stall, we fetch a branch after that waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.
- Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address and update the PC during the second stage of the pipeline.
- Even with this extra hardware, the pipeline involving conditional branches like **Figure 6**.

- The lw instruction, executed if the branch fails, is stalled one extra 200-ps clock cycle before starting.
- If we cannot resolve the branch in the second stage, as is often the case for longer pipelines and the cost of is too high for most computers. To resolve the control hazard we are using branch prediction.

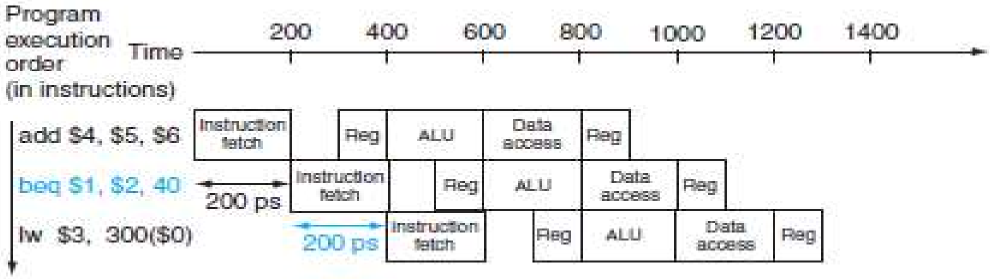


Figure shows the pipeline when the branch is not taken.

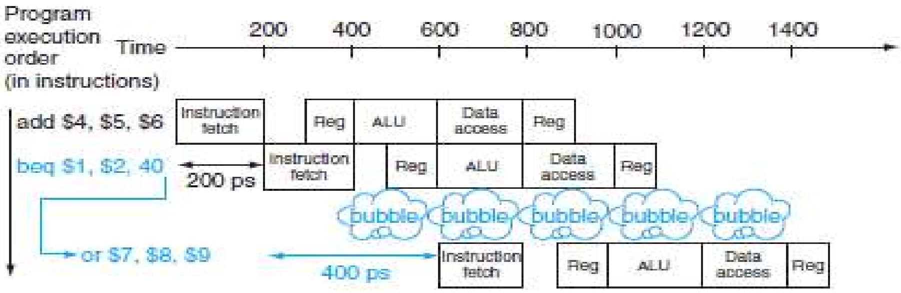


Figure shows the pipeline when the branch is taken.

Branch prediction:

- A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

Dynamic hardware predictors:

- Dynamic hardware predictors make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program.
- In dynamic prediction a person should look at how dirty the uniform was and guess at the formula, adjusting the next guess depending on the success of recent guesses.

- One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken. This using the recent past behavior to predict the future.

Advantages of pipeline:

- Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed.
- Pipelining improves instruction throughput rather than individual instruction execution time or latency.

Latency (pipeline):

- The number of stages in a pipeline or the number of stages between two instructions during execution.

3. 5 Pipelined Data path And Control

- Pipelining has five stages.
  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
- Instructions and data move generally from left to right through the five stages as they complete execution. There are 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 data path
  2. The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage
- These two process stage will perform data flowing from right to left.
- Data flowing from right to left does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements.
- Note that the first right-to-left arrow can lead to data hazards and the second leads to control hazards.

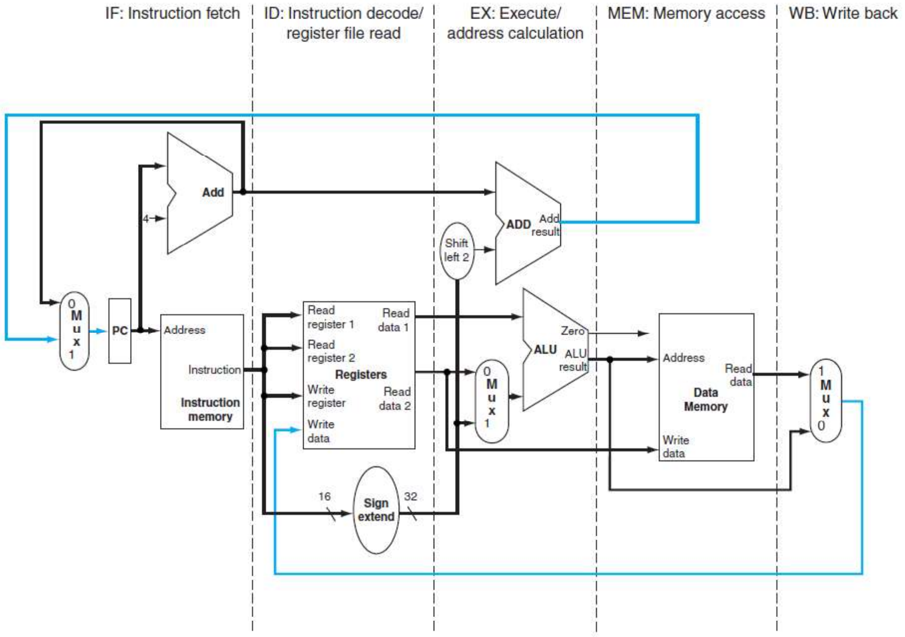


Figure: The single-cycle data path

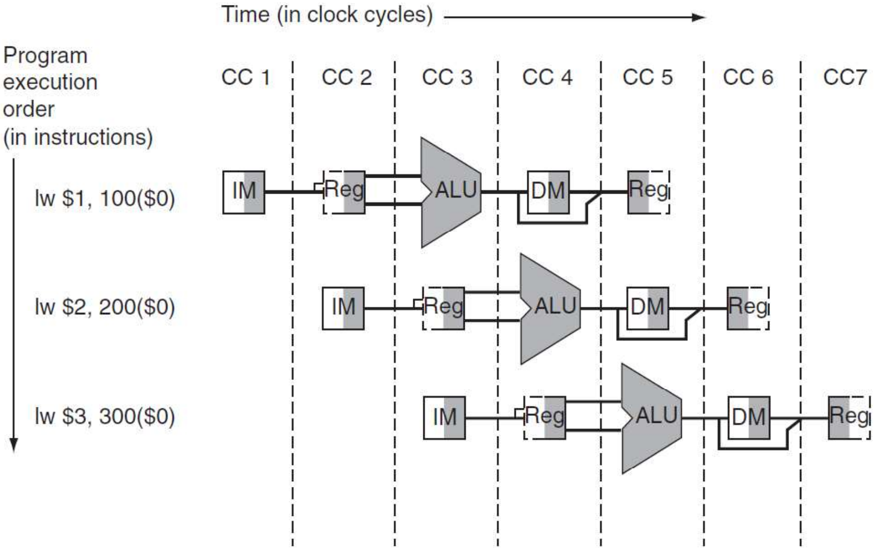


Figure: Instructions being executed using the single-cycle data path

- Consider three load word instructions and how data path are created during pipelining process. Three load word instructions are

  lw $1, 100($0)

  lw $2, 200($0)

lw $3, 300($0)

- IM represents the instruction memory and the PC in the instruction fetch stage. Reg stands for the register file and sign extender in the instruction decode/register file read stage (ID).
- To maintain proper time order, this data path breaks the register file into two logical parts:
    1. Registers read during register fetch (ID).
    2. Registers written during write back (WB).
- The register file is written in the half of the clock cycle and the register file is read during the second half.

Pipelined Data path:

- Pipelined data path can be implemented using the pipeline registers. 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.

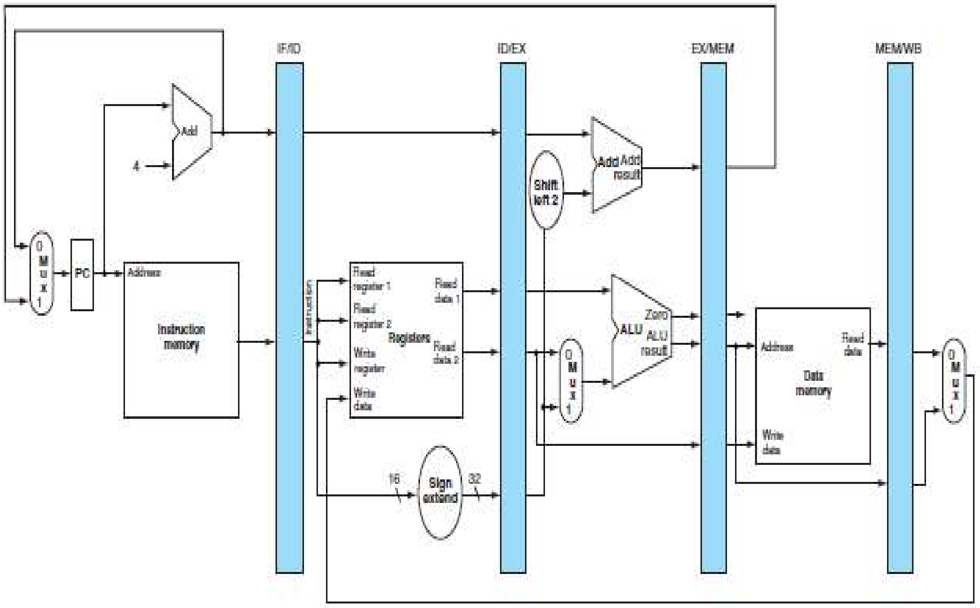


Figure: Pipelined version of the data path

- 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 has a separate pipeline register is redundant to the state that is updated.
- Every instruction updates the PC whether by incrementing it or by setting it to a branch destination address.
- The PC is part of the visible architectural state and its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded.

Load word and Store word Instruction:

- The data path of load word and store word instruction is created by executing five stages of pipelined execution. Load word is active in all five stages.
- The right half of registers or memory is used for read and the left half used for written. Both load and store has five stages. The five stages are the following:

1. Instruction fetch:

Load word:

- The instruction is being read from memory using the address in the PC and then 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.

Store word:

- The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified

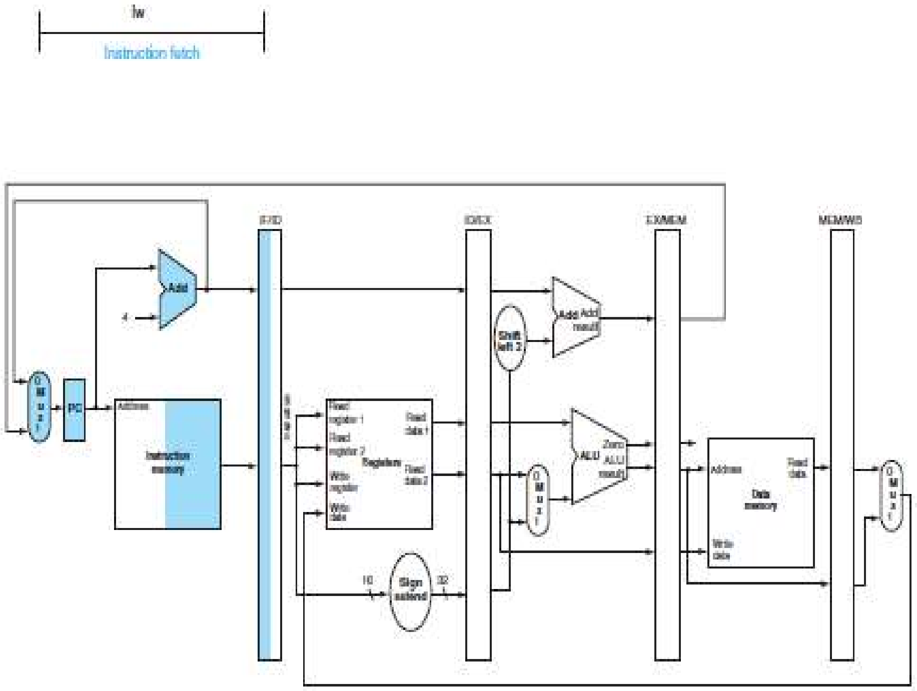


Figure: Instruction fetch for load word and store word

2. Instruction decode and register file read:

Load word:

- 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. A
- Again we need to transfer everything that might be needed by any instruction during a later clock cycle.

Store word:

- The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate.
- These three 32-bit values are all stored in the ID/EX pipeline register.

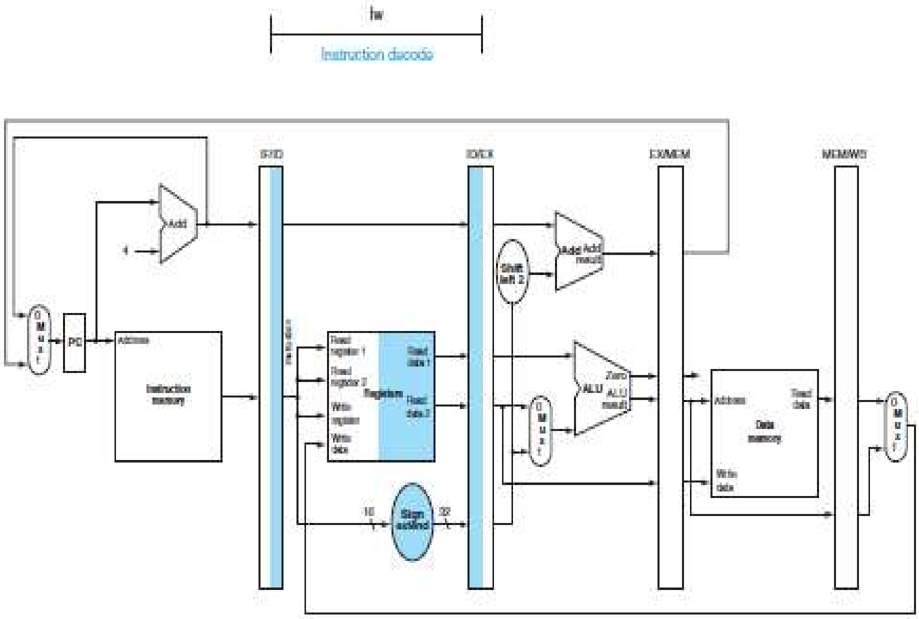


Figure: Instruction decode for load word and store word

3. Execute or address calculation:

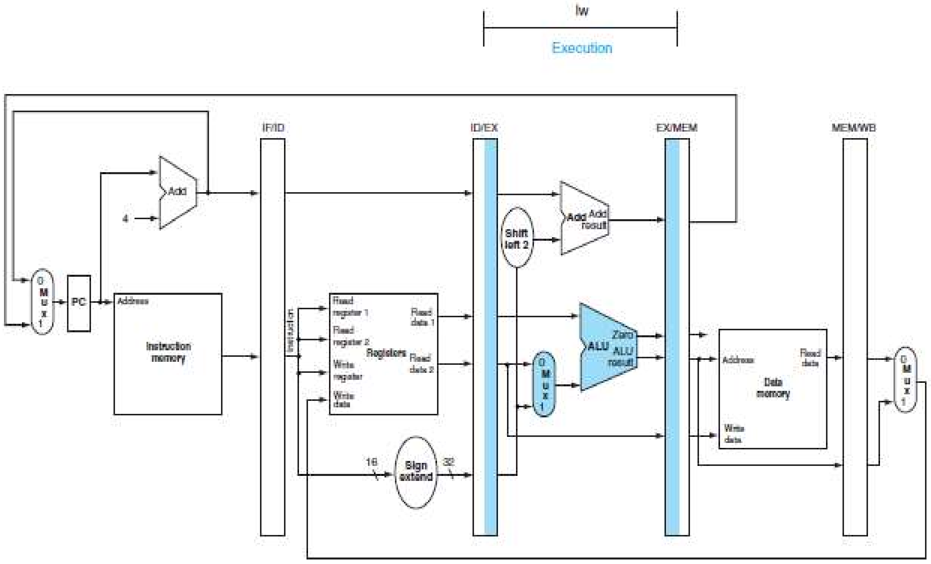


Figure: Third pipeline stage of load instruction

Load word:

- 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.

Store word:

- The effective address is placed in the EX/MEM pipeline register.

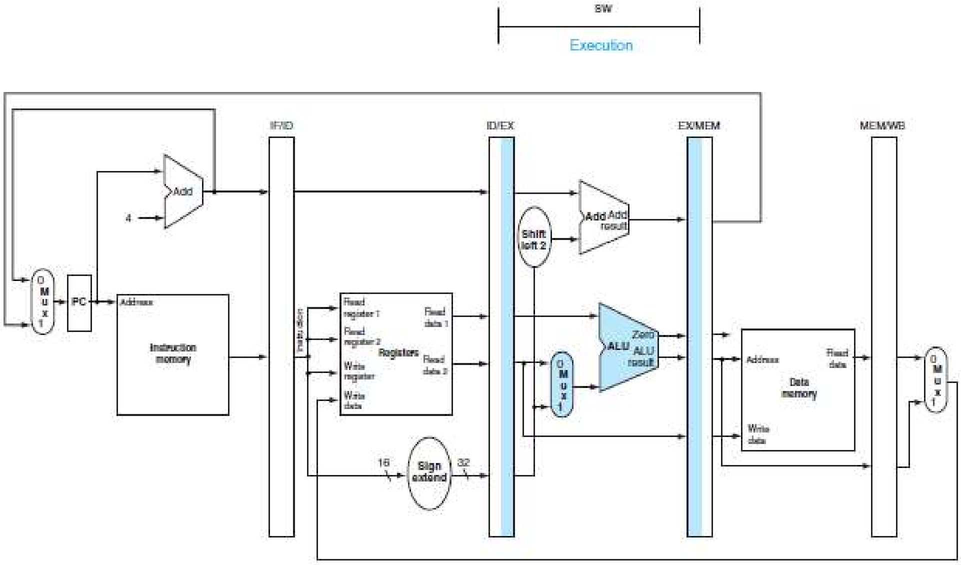


Figure: Third pipeline stage of store instruction

4. Memory access:

Load word:

- 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.

Store word:

- The Figure shows the data being written to memory. The register containing the data to be stored was read in an earlier stage and stored in ID/EX.
- The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage. The effective address is stored into EX/MEM.

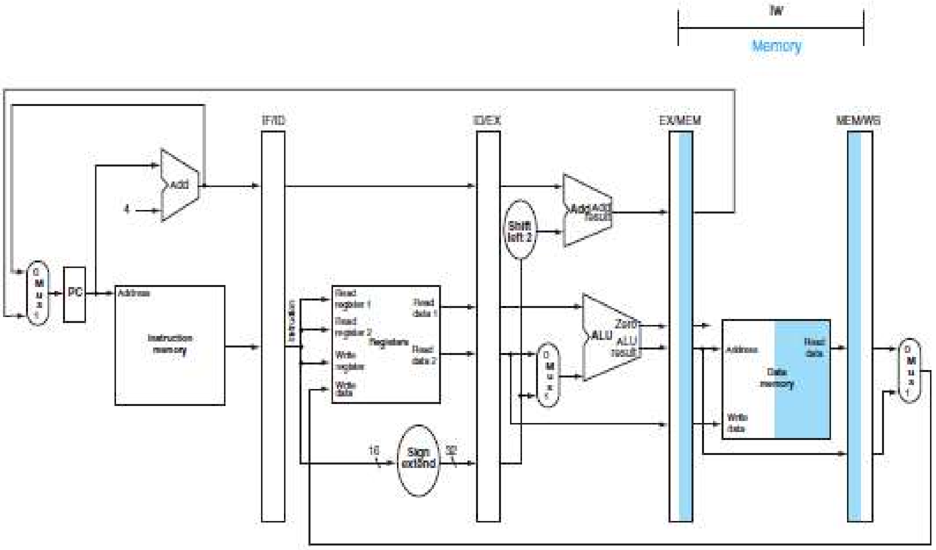


Figure: Fourth pipeline stage of load instruction

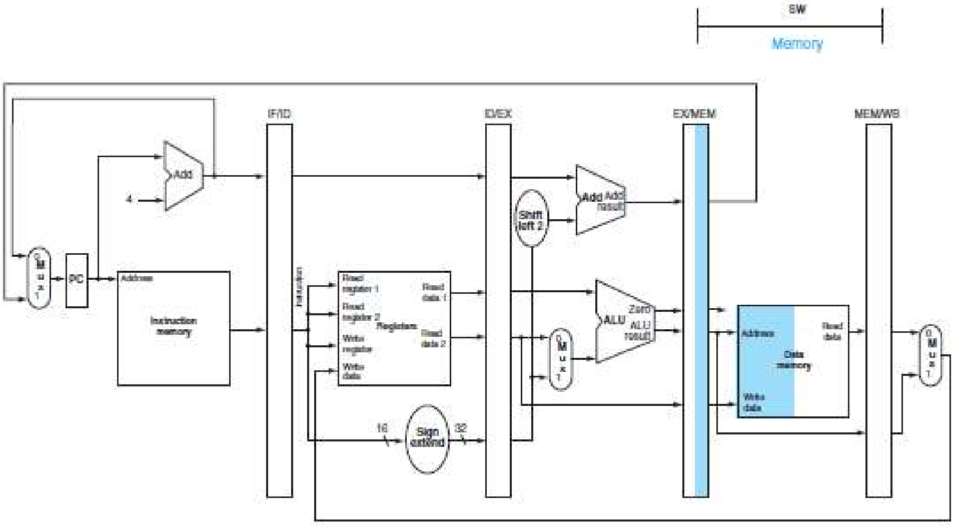


Figure: Fourth pipeline stage of store instruction

5. Write back:

- Reading the data from the MEM/WB pipeline register and writing it into the register file

Store word:

- For this instruction nothing happens in the write-back stage. Because every instruction behind the store is already in progress, we cannot accelerate those instructions.
- Each logical component of the data path in load and store instruction are memory, register read ports, ALU, data memory, and register write port .
- It can be used only within a single pipeline stage. Otherwise it causes structural hazard.

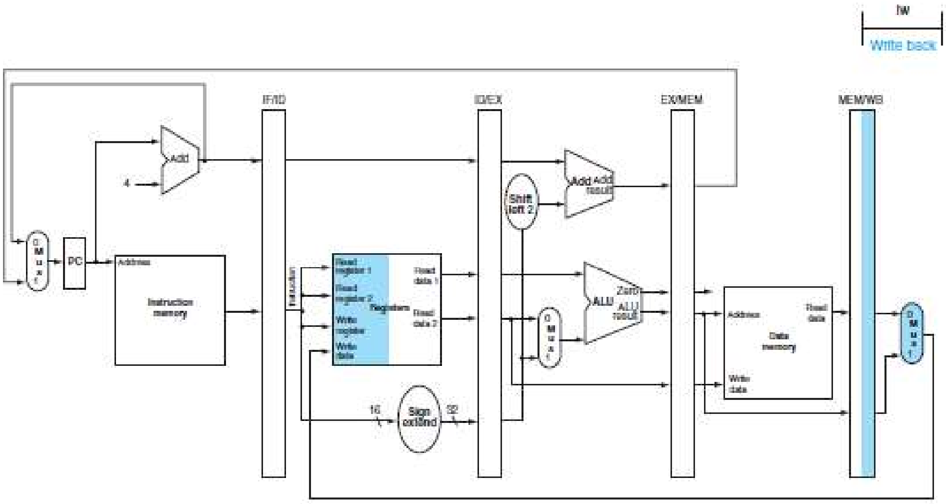


Figure: Fifth pipeline stage of load instruction

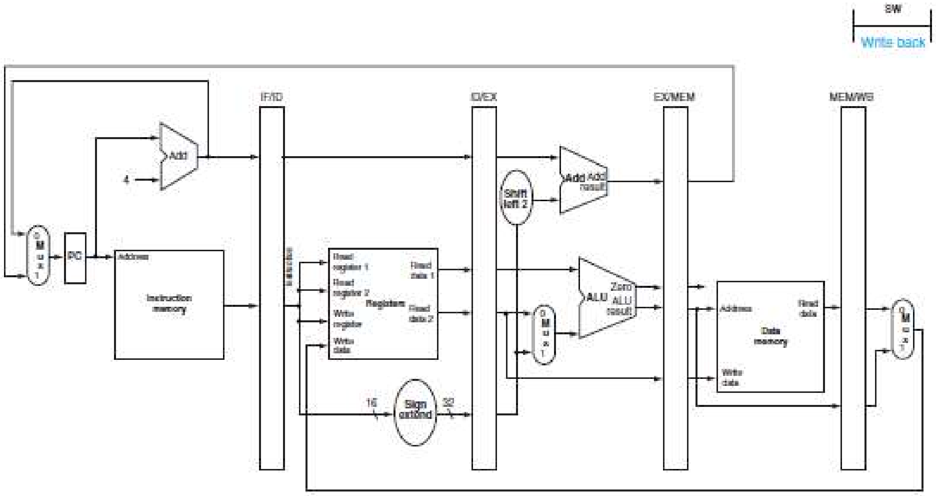


Figure: Fifth pipeline stage of store instruction

- Load instruction must preserve the destination register number. Load must pass the register number from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage.
- Figure shows the correct version of the data path. Passing the write register number first to the ID/EX register, then to the EX/MEM register, and finally to the MEM/WB register. The register number is used during the WB stage to specify the register to be written.

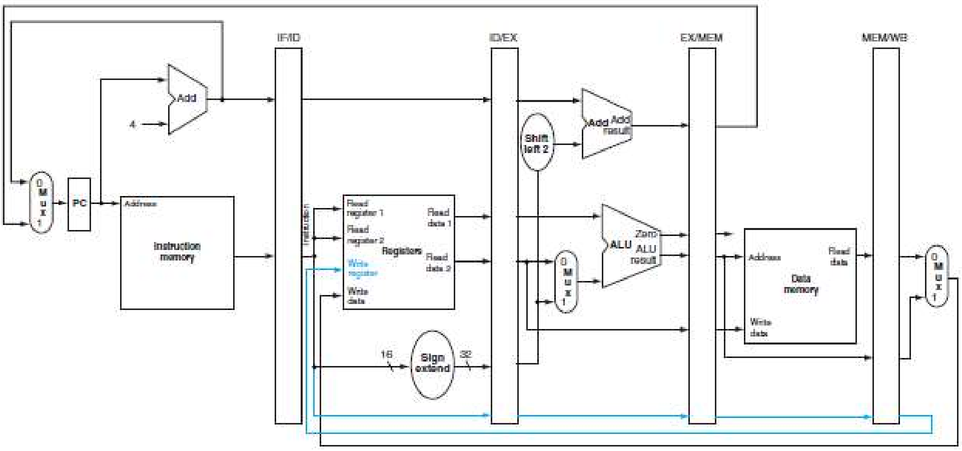


Figure: The corrected pipeline data path to handle the load instruction.

3. 5.1 Graphically Representation of Pipeline

- Pipelining can be difficult to understand because many instructions are executed simultaneously in a single data path in every clock cycle. There are two types of graphically representation for pipeline such as
    1. Multiple-clock-cycle pipeline diagrams
    2. Single-clock-cycle pipeline diagrams

Multiple-clock-cycle pipeline diagrams:

- The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five instruction sequence:

lw $10, 20($1)

sub $11, $2, $3

add $12, $3, $4

lw $13, 24($1)

add $14, $5, $6

- Instructions are executed from top to bottom and clock cycles move from left to right.

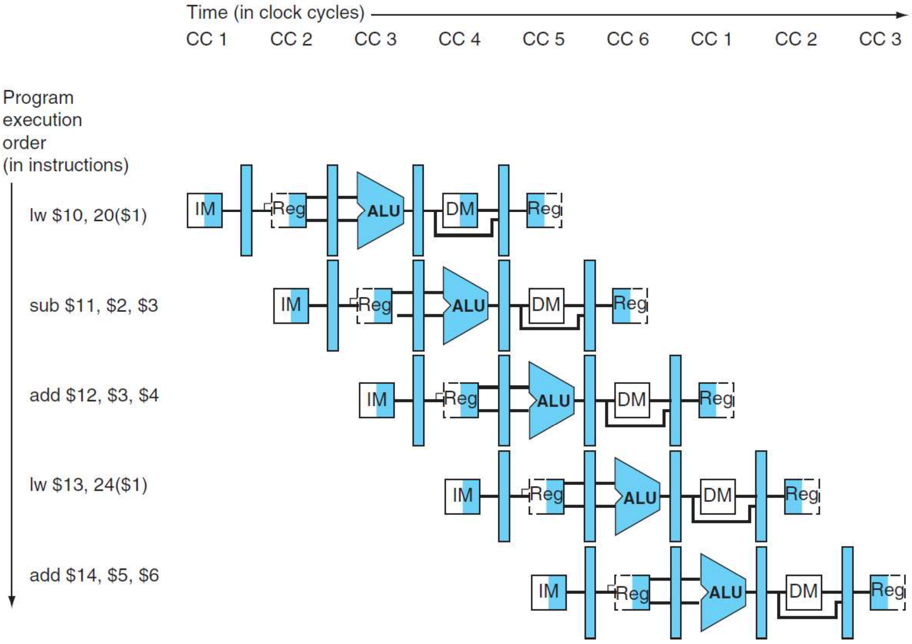
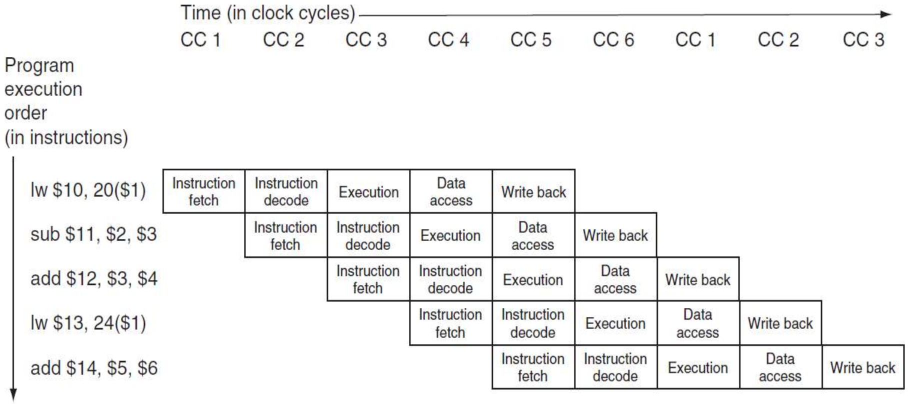


Figure: Multiple-clock-cycle pipeline diagram of five instructions.


Time (in clock cycles)
Program execution order (in instructions)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 1	CC 2	CC 3
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Figure: Multiple-clock-cycle pipeline diagram of five instructions

Single-clock-cycle pipeline diagram:

- It shows the state of the entire data path a single clock cycle. It can be used to know the details of what is happening within the pipeline during each clock cycle.
- A single-clock-cycle diagram represents a vertical slice through a set of multiple-clock-cycle diagram. It shows the usage of the data path by each of the instructions in the pipeline at the designated clock cycle.

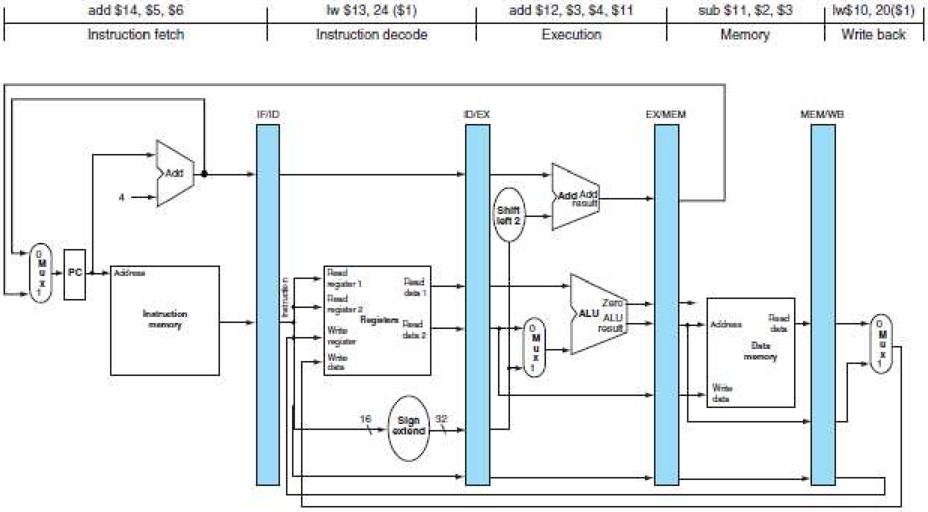


Figure: The single-clock-cycle diagram.

3.6 Handling Data Hazards

- Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.Let's consider the following sequence of instruction.

sub $2, $1,$3	**# Register $2 written by sub**
and $12,$2,$5	**# 1st operand($2) depends on sub**
or $13,$6,$2	**# 2nd operand($2) depends on sub**
add $14,$2,$2	**# 1st($2) & 2nd($2) depend on sub**
sw $15,100($2)	**# Base ($2) depends on sub**

- Register $2 is used in all the five instruction so last four instruction has to wait until the first instruction was executed.
- The last four instructions are all dependent on the result in register $2 of the first instruction.
- If register $2 had the value 10 before the subtract instruction and –20 afterwards after subtract instruction. So last four instructions must use $2 value as -20 for their execution.

Multiple-clock-cycle pipeline representation:

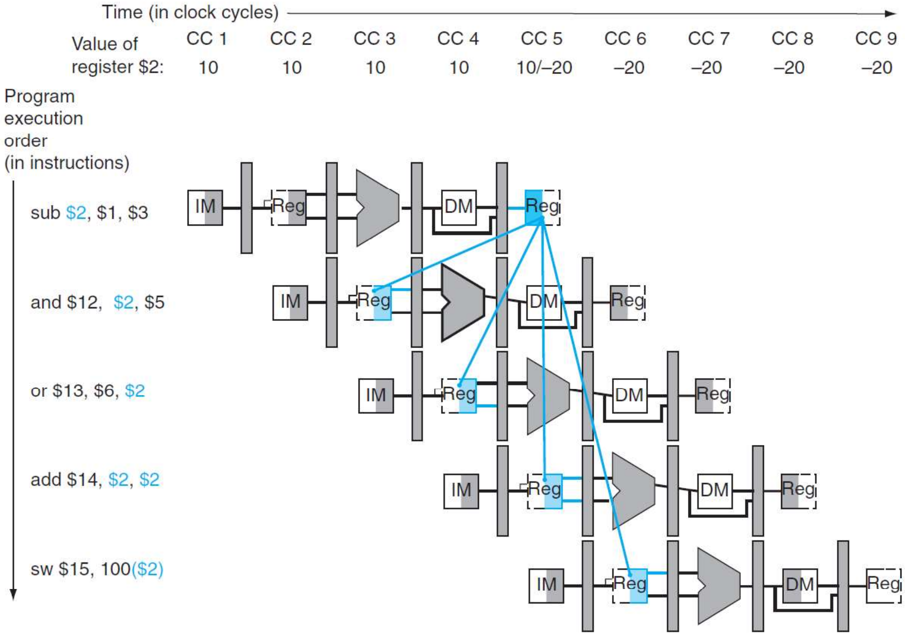


Figure: Pipelined dependences in a five-instruction sequence

- Above instructions are executed in pipeline using multiple clock cycle representation.
- Value of $2 changes in clock cycle 5 because at that point only the sub instruction will write the value of its result.

Register File:

- First instruction (sub) will return the value of $2 and remaining instructions will read the value from $2.

- The proper value of $2 will be available at the end of fifth clock cycle so remaining instruction must wait until fifth clock cycle it cause the data hazard.
- Data Hazard can be resolved using the design of the register file hardware.
- If register is read and written in the same clock cycle means, write is in first half of the clock cycle and read is in the second half of the clock cycle.
- So in such cases the read delivers what is written. Implementing these kinds of register file does not have data hazard.

Forwarding:

- Forwarding is a method used to avoid data hazards in pipeline process.
- Normally the desired result is available at the end of the EX stage or clock cycle 3.
- To perform And and OR instruction we need the result value at the beginning of the EX stage.
- We can execute this segment without stalls if forwarding the data as soon as it is available to any units that need it before it is available to read from the register file.

How does forwarding work?

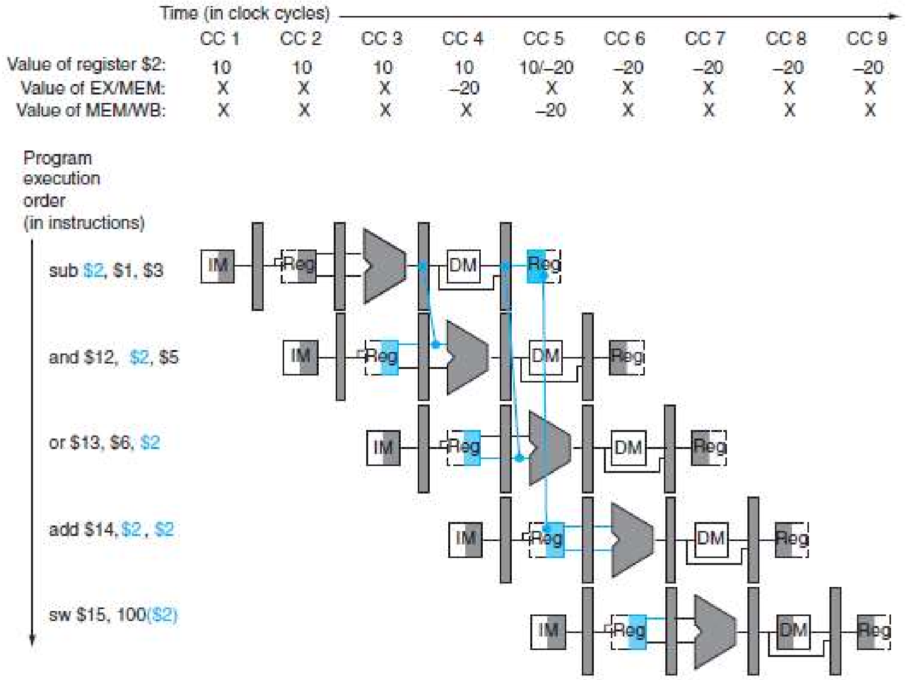
- Assume forwarding is applied only in the EX stage it will either perform an ALU operation or an effective address calculation.
- If EX stage wants to use a register value means it must write in the WB stage itself.
- In forwarding technique we have to use notation to know the fields of the pipeline registers.
- For example, “ID/EX.RegisterRs.
- The above notation refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file.
- The first part of the name (ID) indicates the name of the pipeline register.
- The second part is the name (EX) indicates the name of the field in that register.
- Using this notation, the two pairs of hazard conditions are occurred.

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

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

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

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


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
Value of EX/MEM: X X X –20 X X X X X
Value of MEM/WB: X X X X –20 X X X X
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
Reg

- The first hazard will occur in register $2 between the result of sub $2,$1,$3 and the first read operand of and $12,$2,$5.
- This hazard can be detected when AND instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

  EX/MEM.RegisterRd = ID/EX.RegisterRs = $2
- This policy is inaccurate for all conditions because some instructions do not write registers.
- Examining the WB control field of the pipeline register during the EX and MEM stages determines if RegWrite is asserted.
- Once if we detect the hazards half of the problem is resolved but we must still forward the proper data.

- The dependence begins from a pipeline register rather than waiting for the WB stage to write the register file. Pipeline register hold the required data to be forwarded for later instructions.
- If we can take the inputs to the ALU from *any* pipeline register rather than just ID/EX, then we can forward the proper data.
- By adding multiplexors to the input of the ALU and with the proper controls, we can run the pipeline at full speed in the presence of these data dependences.

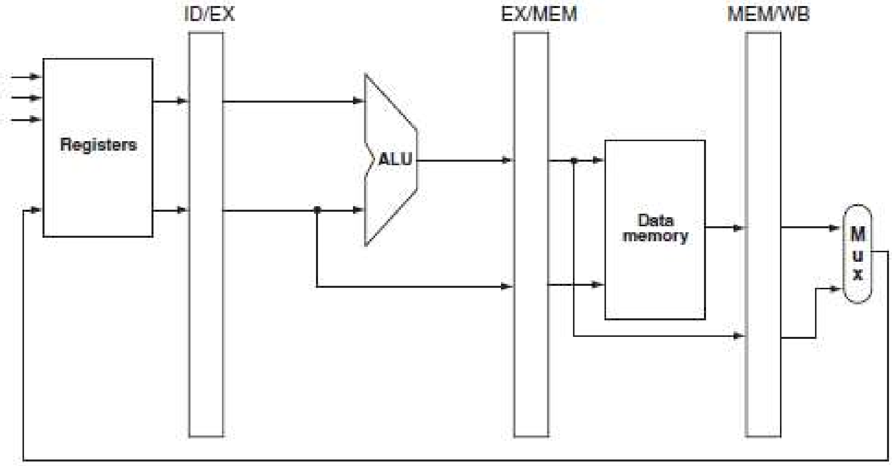


Figure: No forwarding on ALU and pipeline registers

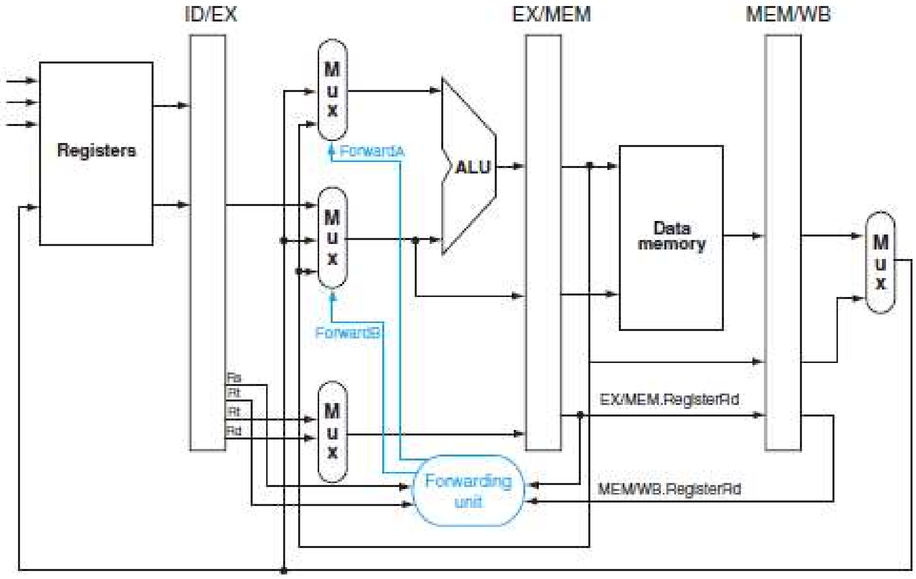
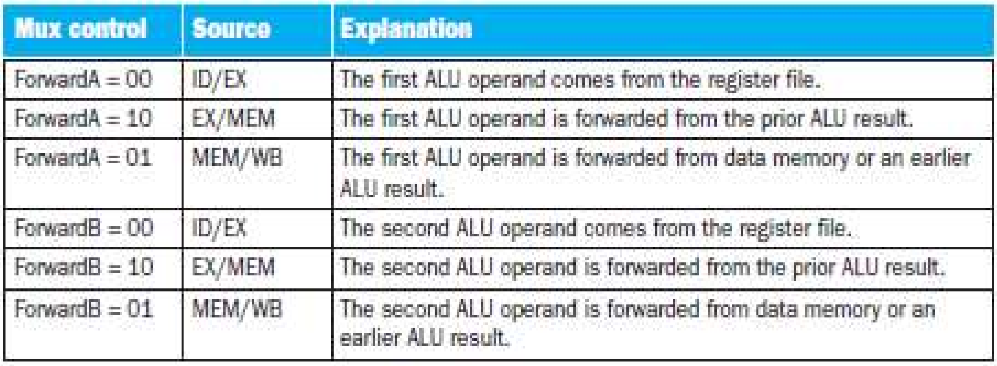


Figure: With forwarding on ALU and pipeline registers



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.

- This forwarding control will be in the EX stage because the ALU forwarding multiplexors are found in that stage.
- Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values.
- We already have the rt field (bits 20–16). Before forwarding, the ID/EX register had no need to include space to hold the rs field. Hence, rs (bits 25–21) is added to ID/EX.
- Conditions for detecting hazards and the control signals to resolve them:

1. 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
```

2. MEM hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd _ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd _ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

3.6.1 Data Hazards and Stalls

- So 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.
- Checking for load instructions, the control for the hazard detection unit is this single condition:

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

- The first line tests to see if the instruction is a load or not.
- The load instruction that reads data memory.
- The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage.
- If the condition holds, the instruction stalls 1 clock cycle.
- After this 1-cycle stall, the forwarding logic can handle the dependence and execution proceeds.
- If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled.
- If IF stage not stalled we would lose the fetched instruction.
- To avoid stalled in these two stage we can prevent the PC register and the IF/ID pipeline register from changing.
- The instruction in the IF stage will continue to be read using the same PC.
- The registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register.

NOP (No operation):

- An instruction that does no operation to change state.
- NOP is acase where pipeline stage executing an instruction but that does not make any changes in the state.
- NOP act like a bubble in the pipeline stage.

Inserting NOPs in pipeline:

- NOPs are like bubbles, to identify hazard in ID stage we had to insert bubble into the pipeline.
- Once the bubble is inserted in the ID stage it will change the control field of EX, MEM, and WB field of the ID/EX pipeline register to 0.
- If the control values are 0 then no registers or memories are written.
- If AND instruction is NOP then all instruction beginning with the AND instruction are delayed one cycle.
- A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop.
- AND instruction is really fetched and decoded in clock cycles 2 and 3 but its EX stage is delayed until clock cycle 5.
- OR instruction is fetched in clock cycle 3, but its IF stage is delayed until clock cycle 5.
- After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

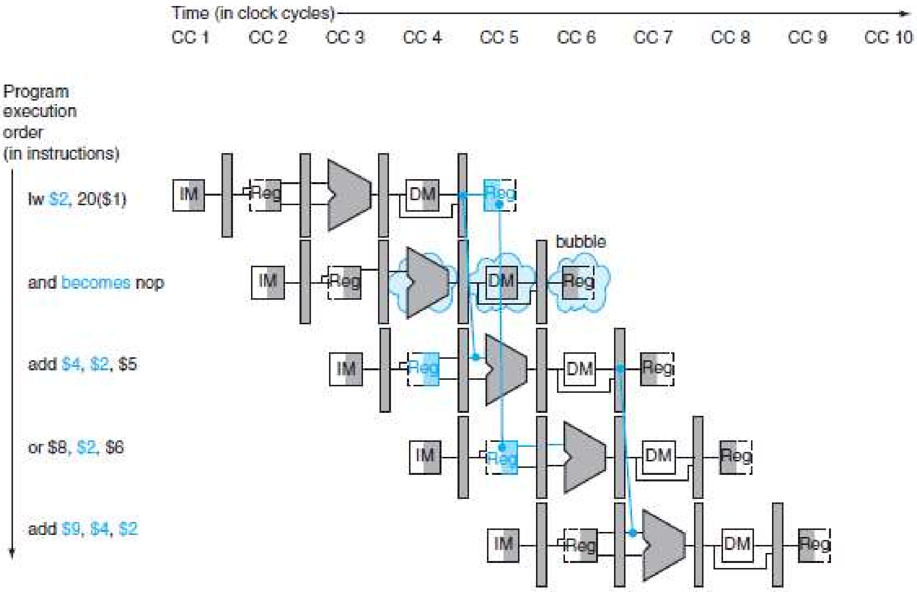


Figure: The way stalls are really inserted into the pipeline.

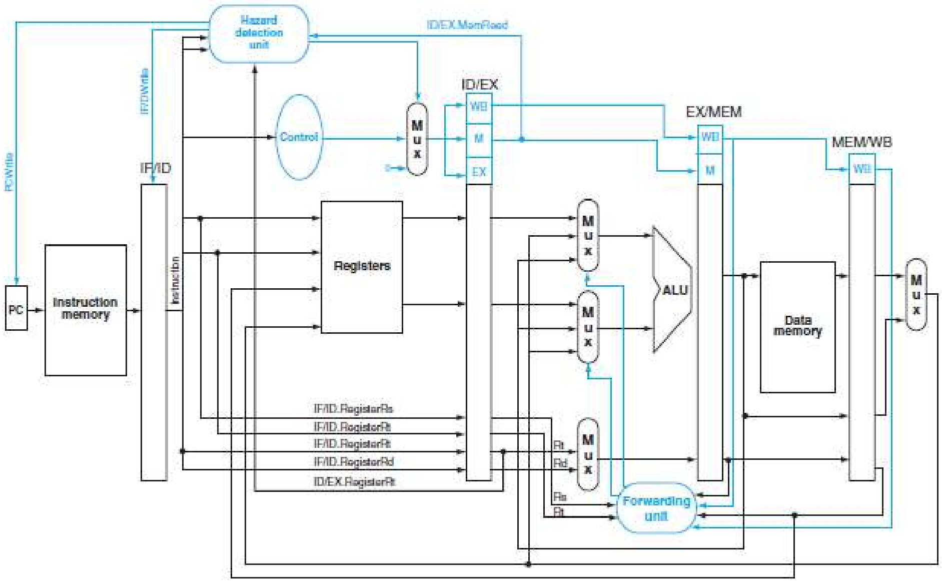


Figure: Pipelined connection for both the hazard detection unit and the forwarding unit

Hazard detection unit:

- The hazard detection unit controls the writing of the PC and IF/ID registers.
- It also controls the multiplexer that chooses between the real control values and all 0s.

Forwarding unit:

- Forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register.
- Using these two units we can resolve the data hazard occurred in the pipelining process.

3.7 Control Hazards

- Control hazard occurs when we execute the branch instruction in pipeline process. Control hazard are relatively simple to understand and if occurs at less frequently.
- To avoid Control hazard in pipeline process we can use simple method and no need to go for any special techniques like data hazard.

- Figure shows a sequence of instructions and indicates when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage.
- The numbers to the left of the instruction (40, 44 . . .) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage, clock cycle 4 for the beq instruction above.
- The three sequential instructions that follow the branch will be fetched and begin execution before beq branches to lw at location 72.
- To resolve the Control hazard we have to know whether branch not taken or not. There are two schemes used to resolve the Control hazard such as
    1. Branch Not Taken
    2. Branch Prediction

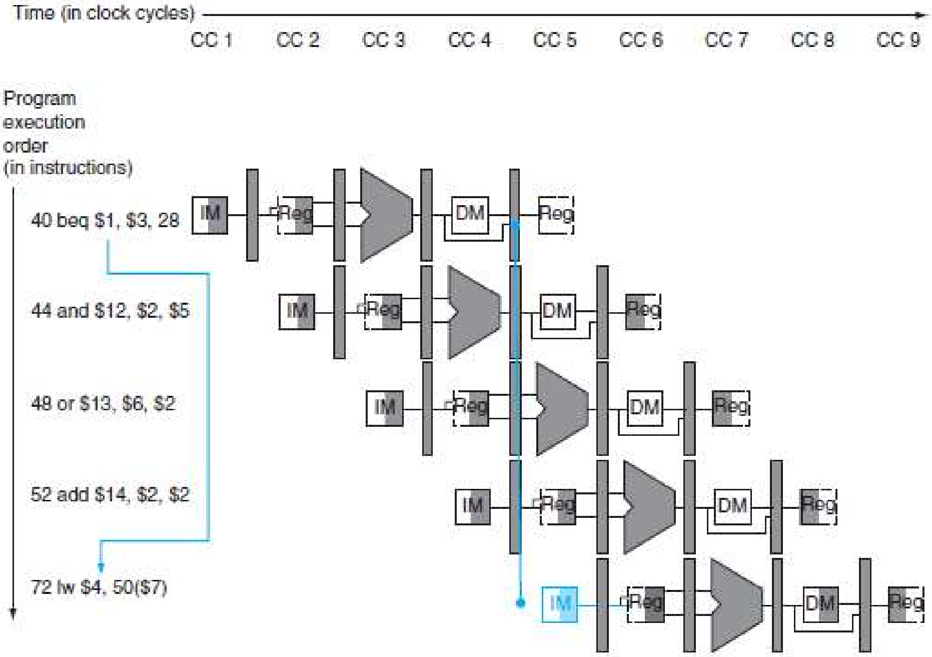


Figure: Pipeline on the branch instruction

3.7.1 Branch Not Taken

- If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.

- Discarding instructions means we must be able to flush instructions in the IF, ID, and EX stages of the pipeline.
- Flush is a method used to discard instructions in a pipeline usually due to an unexpected event.
- If branch instruction is fetched immediately it stall execution until the pipeline determine the outcome of the branch and knows which instruction address is to fetch.
- If branches are not taken means no need to discard any instruction and pipelining will execute the instruction continuously.
- Branches are taken then only pipeline has stalled so by reducing the delay of branches we can improve the performance of pipelining process in this condition.

3.7.2 Reducing the Delay of Branches

- One way to improve branch performance is to reduce the cost of the taken branch.
- The next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed.
- Executing branch instruction earlier in the pipeline will increase the speed of performance.
- The MIPS architecture was designed to support fast single cycle branches that could be pipelined with a small branch penalty.
- The designers observed that many branches had only simple test for that it does not require a full ALU operation .
- For more complex branch decision we need to use an ALU to perform a required comparison.
- Moving the branch decision up requires two actions to occur earlier:
  - Computing the branch target address
  - Evaluating the branch decision.

Computing the branch target address:

- The branch address calculation will be performed for all instruction but used only when it needed. To calculate branch target address we need Pc value and IF/ID field value.
- Already we 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 that will give new branch target address.

Evaluating the branch decision:

- Branch decision is a complex task and evaluating branch decision is not an easy task.Branch decisions are
    - Branch equal
    - Branch not equal
- For example, consider branch equal decision itself. We have to compare the two registers read during the ID stage to see if they are equal.
- Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware.
- To implement branch-on-equal we will forward results to the equality test logic that operates during ID.

Two complicating factors:

- 1. During ID, we must decode the instruction to decide whether a bypass to the equality unit is needed, and complete the equality comparison.
- If the instruction is a branch we can set the PC to the branch target address. Forwarding for the operands of branches handled by the ALU forwarding logic unit.
- But introduction of the equality test unit in ID will require new forwarding logic.
- The bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.

- 2. The values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed.

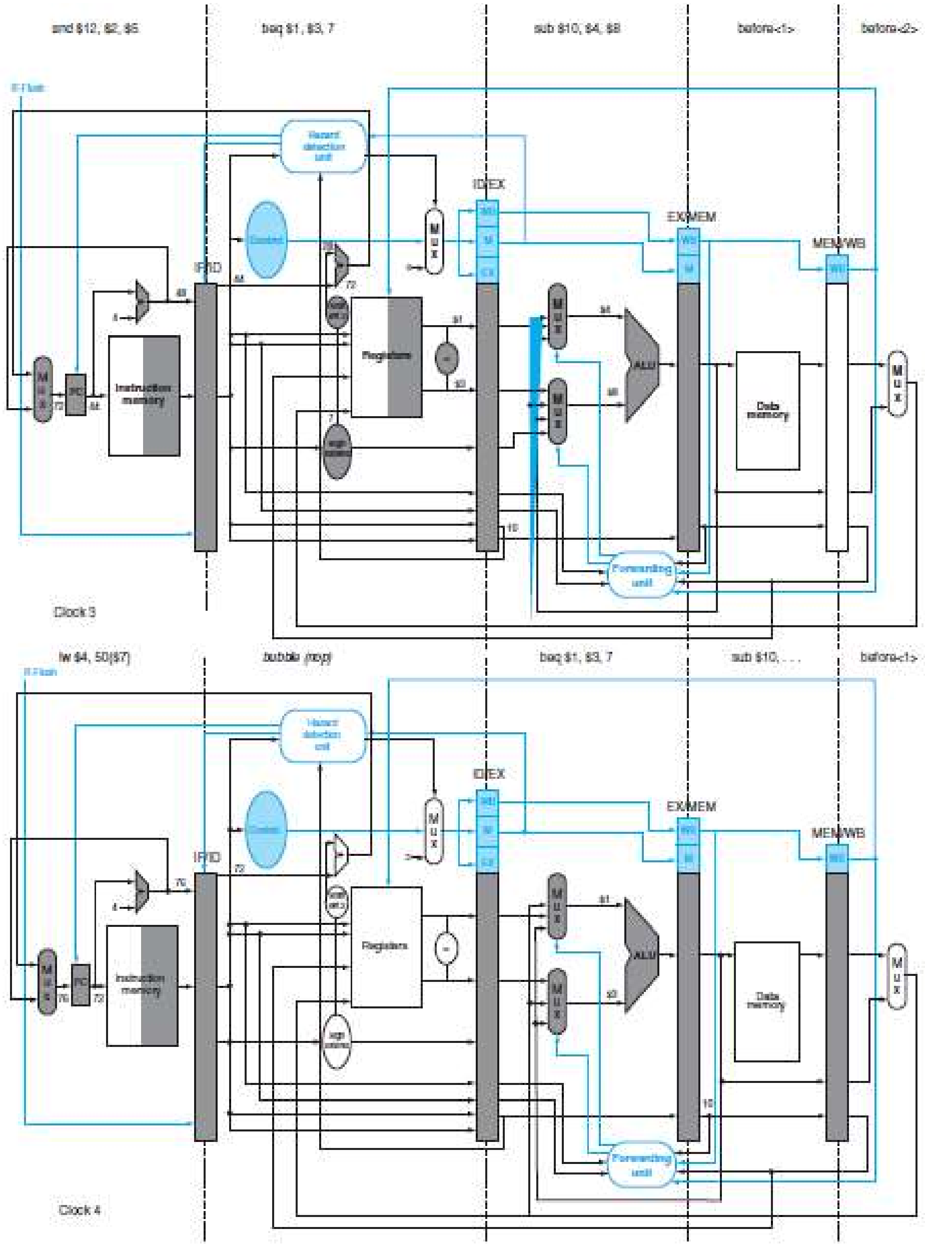


Figure: The ID stage of clock cycle 3 determines that a branch must be taken.

- 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.

- To overcome these difficulties, moving the branch execution to the ID stage it reduces the penalty of a branch to only one instruction if the branch is taken.

Branch Prediction:

- Branch prediction is a method of resolving a branch hazard. It assumes the outcomes of branch and proceeds for that assumption rather than waiting to determine the actual outcome.
- A simple form of branch prediction is we have to assume branch is not taken. This assumption is possible only for 5 stage pipeline.
- For deeper pipelines this assumption is not suitable because it will increase the branch penalty when measured in clock cycles. For such kind of pipelines we have to add more hardware to predict branch behavior during program execution.
- Solution for increasing branch penalty we have new technique called dynamic branch prediction.

3.7.3 Dynamic Branch Prediction

- Dynamic branch prediction is a prediction of branches at runtime using runtime information.
- To implement dynamic branch prediction method we have to use one buffer that is called branch prediction buffer or branch history table.
- A branch prediction buffer 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. Prediction is just a hint that is assumed to be correct, so fetching begins in the predicted direction.
- If the hint is wrong then the incorrectly predicted instructions are deleted.
- The prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

Loops and Prediction:

Example:1

Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

Solution:

- The steady-state prediction behavior will mispredict on the first and last loop iterations.
- Mispredicting the last iteration is inevitable since the prediction bit will say taken: the branch has been taken nine times in a row at that point.
- The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that exiting iteration.
- Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80%

Drawback:

- The accuracy of the predictor would match the taken branch frequency for these highly regular branches.

Two-bit prediction scheme:

- To overcome the drawback of branch prediction buffer method, new method called Two-bit prediction schemes is used.
- In a 2-bit scheme, a prediction must be wrong twice before it is changed.
- 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 then fetching begins from the target as soon as the PC is known.\ Otherwise, sequential fetching and executing continue.

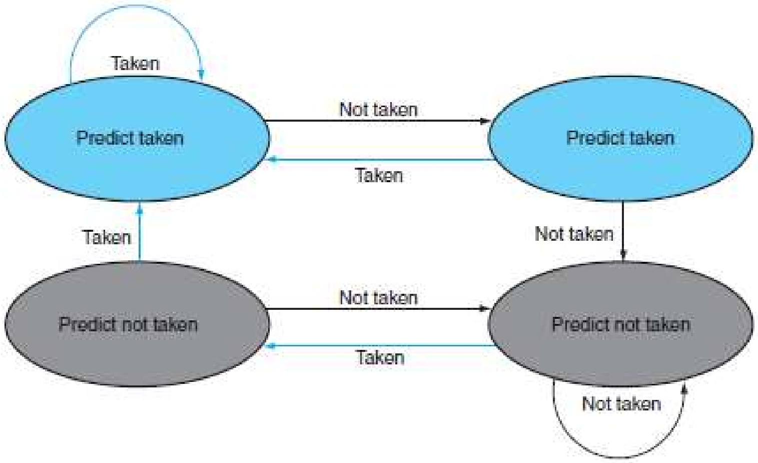


Figure: The states in a 2-bit prediction scheme

3.8 Exceptions

- Control is the most challenging aspect of processor design because of two reasons:
    1. It is the hardest part to get right
    2. It is the hardest part to make fast.
- One of the hardest parts of control is implementing exceptions and interrupts.
- Handling exceptions and interrupts are more complex task than handling branches or jumps.
- That changes the normal flow of instruction execution.
- An exception and interrupts are initially created to handle unexpected event from within the processor; arithmetic overflow is an example of an exception.

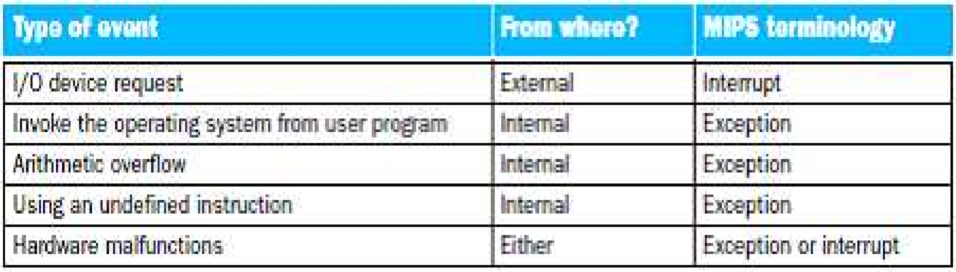
Exception:

- Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

Interrupt:

- An exception that comes from outside of the processor.
- An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor.

- Interrupt refer to any unexpected change occurred only when the event is externally caused.



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

- The above examples show whether the situation is generated internally by the processor or externally generated. To detect two types of exceptions we need to implement control.
- Two types of exceptions arise
  1. From the portions of the instruction set
  2. Implementation
- Detecting exceptional conditions and taking the appropriate action is on the critical timing path of a processor. Processor must determines the clock cycle time and thus performance.
- To design a control unit we must have proper attention to exceptions.
- Because exception can reduce performance and it leads complex task ot get the correct design.

3.8.1 Exceptions in MIPS Architecture

- In MIPS Architecture two types of exceptions occur at
  1. Execution of an undefined instruction
  2. An arithmetic overflow
- Once exception occur then the processor must save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.
- After transferring control to the operating system can take the appropriate action to provide some service to the user program.
- Operating system will provide the following services to user program.

1. Taking some predefined action in response to an overflow
2. Stopping the execution of the program and reporting an error.

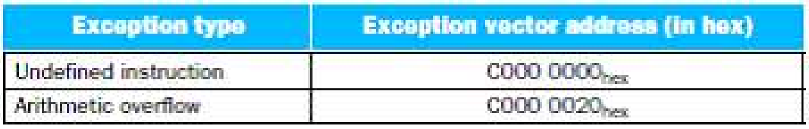
- After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its instruction execution.
- Operating system uses the EPC to determine where to restart the execution of the program.
- To handle the exception by operating system it must know the reason for the exception.
- MIPS architecture has two methods to find the reason for the exception
    1. Status register
    2. Vectored interrupts

Status registers:

- It is also called the cause register. It holds a field that indicates the reason for the exception.

Vectored interrupts:

- An interrupt for which the address to which control is transferred is determined by the cause of the exception.
- The operating system knows the reason for the exception by the address at which it is initiated.
- The addresses are separated by 32 bytes or 8 instructions.



Exception type	Exception vector address (in hex)
Undefined instruction	C000 0000$_{\text{hex}}$
Arithmetic overflow	C000 0020$_{\text{hex}}$

- The operating system must record the reason for the exception and may perform some limited processing in this sequence.
- When the exception is not vectored, a single entry point for all exceptions can be used.

- If single entry point is used for all exceptions then the operating system decodes the status register to find the reason for exception.
- To handle exception we can add a few extra registers and control signals to their basic implementation.

3.8.2 Implementation of exception in MIPS architecture

- To implement exception system in MIPS architecture assume it has single entry point for all exception and has address value is 8000 0180.
- This address value indicates it is an arithmetic overflow exception.
- To handle this exception we need to add two additional registers to our current MIPS implementation.Two additional registers are EPC and cause register.

EPC:

- A 32-bit register used to hold the address of the affected instruction.This register is needed even when exceptions are vectored.

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 that the low-order bit of this register encodes the two possible exception: undefined instruction = 0 and arithmetic overflow = 1.

Exception in pipelined Implementation:

- In a pipelined implementation exceptions are treated as another form of control hazard.
- In MIPS exception address we add an additional input to the PC multiplexer that sends 8000 0180_{hex} to the PC.
- Cause register record the cause of the exception
- The 8000 0180_{hex} input to the multiplexor is the initial address to begin fetching instructions in the event of an exception.

- If we do not stop execution in the middle of the instruction then following things will happen.
- The programmer will not be able to see the original value of register $1.
- Register $1 is used to cause the overflow because it will be tackle the destination register of the add instruction.
- Because of careful planning, the overflow exception is detected during the EX stage. So we can use the EX. Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage.
- The final step is to save the address of the offending instruction in the Exception Program Counter (EPC), save the address + 4, so the exception handling routine must first subtract 4 from the saved value.
- Multiple exceptions can occur simultaneously in a single clock cycle.

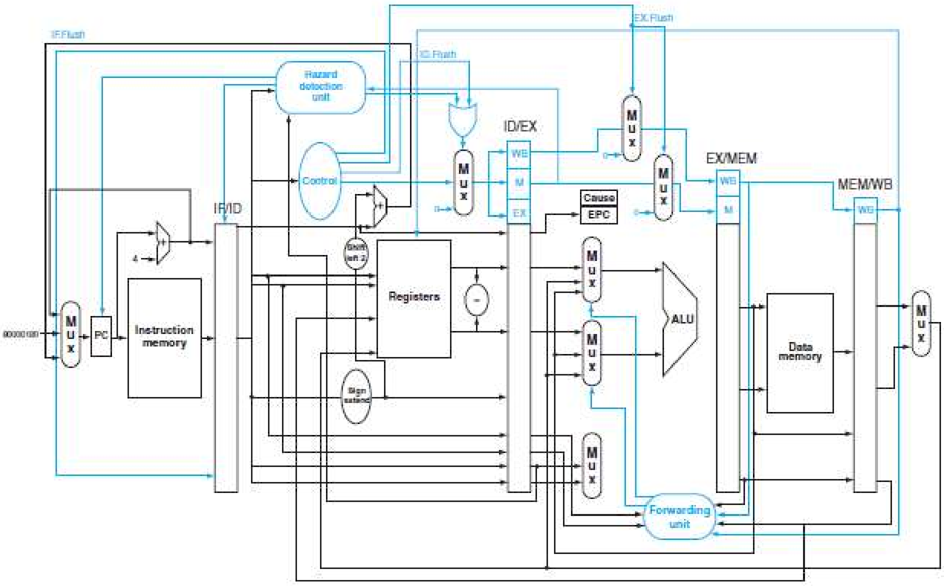


Figure: The data path with controls to handle exceptions

- The normal solution is to prioritize the exceptions so that it is easy to determine which is serviced first.
- The EPC captures the address of the interrupted instructions, and the MIPS Cause register records all possible exceptions in a clock cycle, so the exception software must match the exception to the instruction.

- An important clue knows in which pipeline stage a type of exception can occur. For example, an undefined instruction is discovered in the ID stage, and invoking the operating system occurs in the EX stage.