

1. What is MIPS, MIPS Instruction, MIPS Implementation

MIPS is implementation of a RISC architecture

MIPS R2000 ISA

- ❖ Designed for use with high-level programming languages
 - small set of instructions and addressing modes, easy for compilers
- ❖ Minimize/balance amount of work (computation and data flow) per instruction
 - allows for parallel execution
- ❖ Load-store machine
 - large register set, minimize main memory access
- ❖ fixed instruction width (32-bits), small set of uniform instruction encodings
 - minimize control complexity, allow for more registers

MIPS Instructions

- ❖ MIPS instructions fall into 5 classes:
 - Arithmetic/logical/shift/comparison
 - Control instructions (branch and jump)
 - Load/store
 - Other (exception, register movement to/from GP registers, etc.)

Three instruction encoding formats

R-type (6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit shamt, 6-bit function code)

31-26	25-21	20-16	15-11	10-6	5-0
<i>opcode</i>	<i>rs</i>	<i>rt</i>	<i>rd</i>	<i>shamt</i>	<i>function</i>

I-type (6-bit opcode, 5-bit rs, 5-bit rt, 16-bit immediate)

31-26	25-21	20-16	15-0
<i>opcode</i>	<i>rs</i>	<i>rt</i>	<i>imm</i>

J-type (6-bit opcode, 26-bit pseudo-direct address)

31-26	25-0
<i>opcode</i>	<i>pseudodirect jump address</i>

Basic MIPS Implementation

For all instructions, the first two steps are common:

- ❖ Set the program counter (PC) to the memory location that contains the code and fetch the instruction from that memory location.
- ❖ Read one or two registers, using the fields of the instruction.

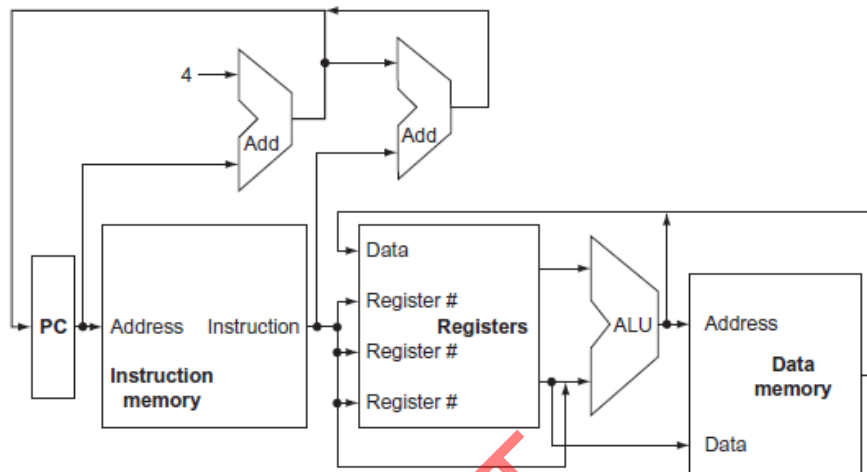


Fig: An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

- ❖ After these two steps, the actions required to complete the instruction depends on the instruction class.

Three commonly used instruction classes are

1. Memory Reference
2. Arithmetic Logical
3. Branches

For each of the three instruction classes the actions are exactly the same, independent of the exact opcode.

- ❖ Across instructions there are some similarities.
 - For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.
 - The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and 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 read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register.
- ❖ Lastly, for a branch instruction, we may 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 A shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection.
- ❖ The program counter provides the address of the next instruction to be fetched.
- ❖ After the instruction is fetched, the register operands are fetched.
- ❖ Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a 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.

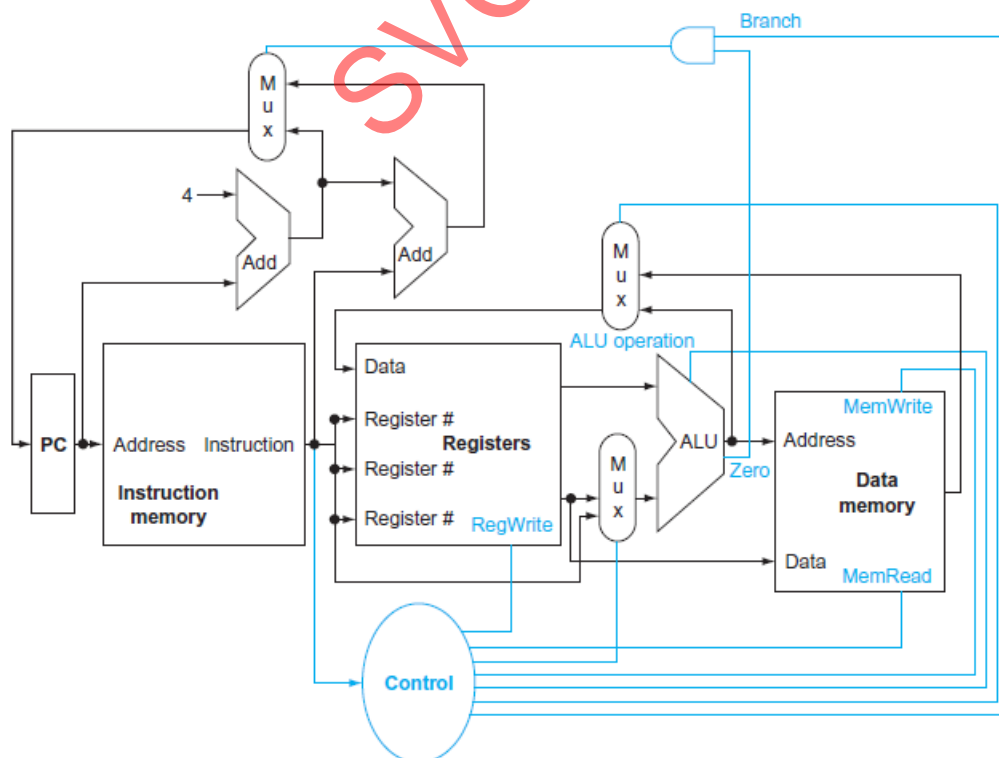


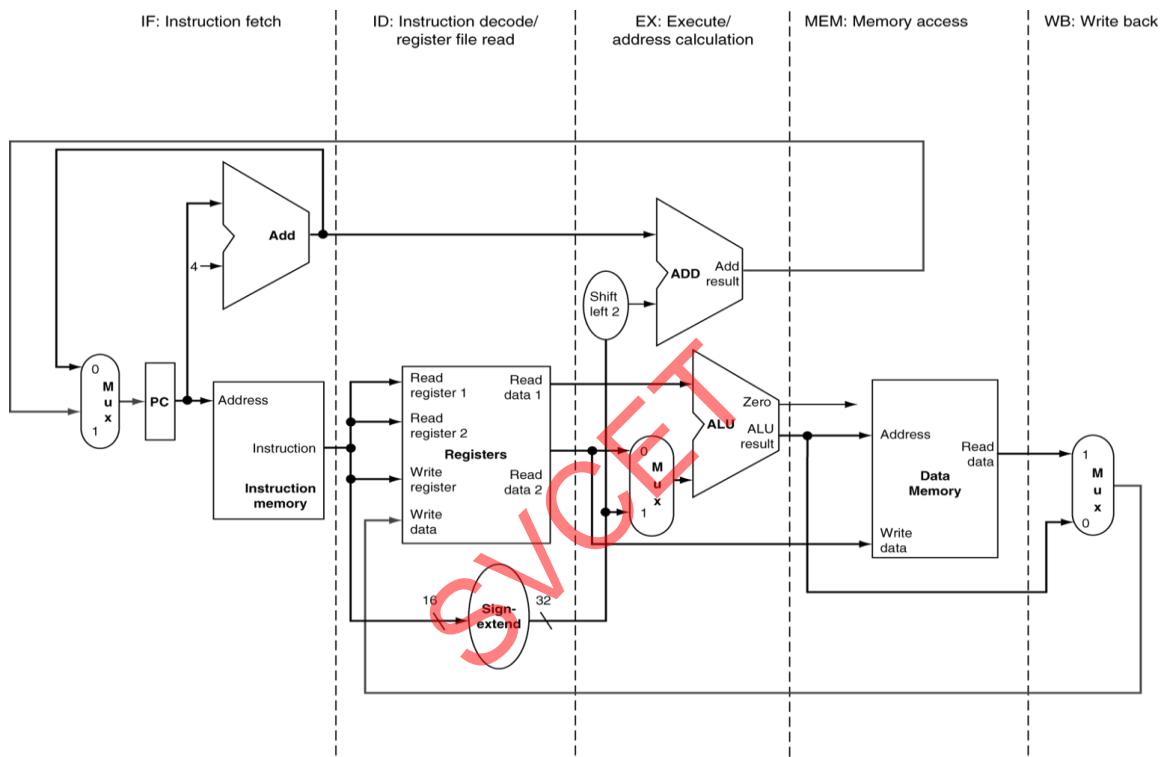
Fig B: The basic implementation of the MIPS subset, including the necessary multiplexors and control lines

- ❖ The result from the ALU or memory is written back into the register file.
- ❖ Branches require the use of the ALU output to determine the next instruction address.
- ❖ The thick lines interconnecting the functional units represent buses, which consist of multiple signals.
- ❖ Figure B shows the datapath of Figure A with the three required multiplexors added, as well as control lines for the major functional units.
- ❖ A control unit, uses the instructions as an input, to determine how to set the control lines for the functional units and two of the multiplexors.
- ❖ 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.
- ❖ The top multiplexor (“Mux”) controls what value replaces the PC ($PC + 4$ or the branch destination address); the multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal used to indicate if the instruction is a branch.
- ❖ The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.
- ❖ Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction.
- ❖ The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.

2. Explain Datapath in pipeline or pipelined datapath?

The data path is separated into five pieces, with each piece named corresponding to a stage of instruction execution:

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



Refer Above diagram, instructions and data move generally from left to right through the five stages as they complete execution.

There are, however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the data path.
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

Data flowing from right to left does not affect the current instruction; these reverse data movements influence only later instructions in the pipeline.

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

To retain **the value of an individual instruction for its other four stages**, the value read from instruction memory must be **saved in a register**. Similar arguments apply to every pipeline stage, so we must **place registers wherever there are dividing lines between stages**.

The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID. Notice that there is no pipeline register at the end of the write-back stage.

All instructions must update some state in the processor -the register file, memory, or the PC. Its contents must be saved in the pipeline register when an exception occurs, while the contents of the pipeline registers can be discarded. The registers can be referred from the below diagram.

Consider load instruction abbreviation lw and how the operations of this instruction in a pipe line is done are as follows: (lw \$s0, 32(\$t0))

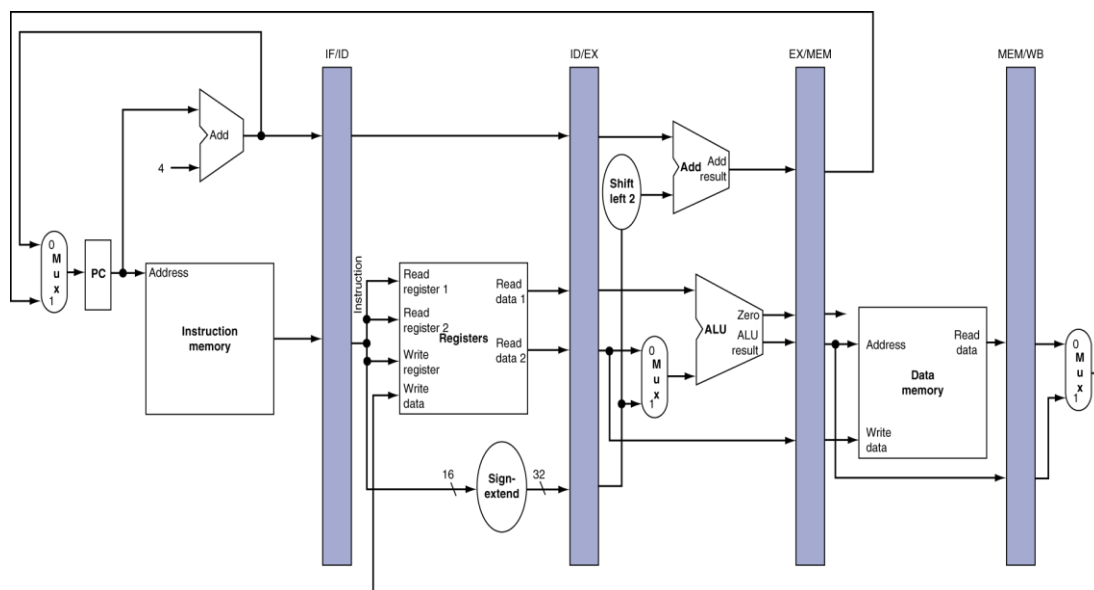
1. Instruction fetch: The top portion of Figure shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an branch instruction, **such as beq.**

2. Instruction decode and register file read: The bottom portion of Figure shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field of the load instruction (**here in our example is 32**), which is sign-extended to 32 bits, and the register numbers to read the two registers (**\$s0 and \$t0**). All three values are stored in the ID/EX pipeline register, along with the incremented PC address.

3. Execute or address calculation: Figure shows that the load instruction reads the contents of register1 (**\$t0**) and the sign-extended immediate (**value 32**) from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.

4. Memory access: The top portion of Figure shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

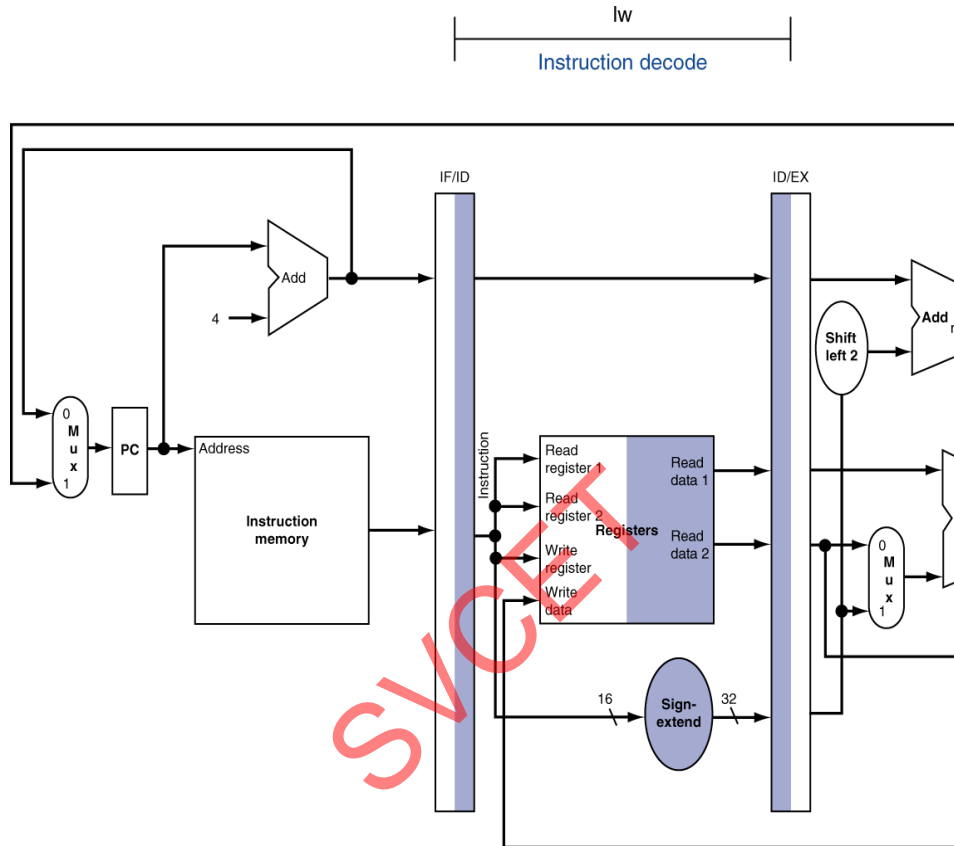
5. Write-back: The bottom portion of Figure shows the final step: **reading** the data from the MEM/WB pipeline register and writing it into the register file (**\$s0**) in the middle of the figure.



The following figure shows the reading and writing operations done during a stage (here instruction decode stage). It show that reading operation is performed from IF/ID register to get input for the stage and after completion of execution of that stage the results are written in ID/EX register.

Note: The read and write operations are shaded differently.

Similarly other stages follows with the same read / write operations with respect to the execution of each stage and pipelined buffers.



3. Explain control path in pipeline or pipelined control?

Now in this section we add control to the pipelined data path. The PC is written on each clock cycle, so there is no separate write signal for the PC. There are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

Each control line is associated with a component, that is active in only a single pipeline stage. we can divide the control lines into five groups according to the pipeline stage.

1. Instruction fetch: The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.

2. Instruction decode/register file read: As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.

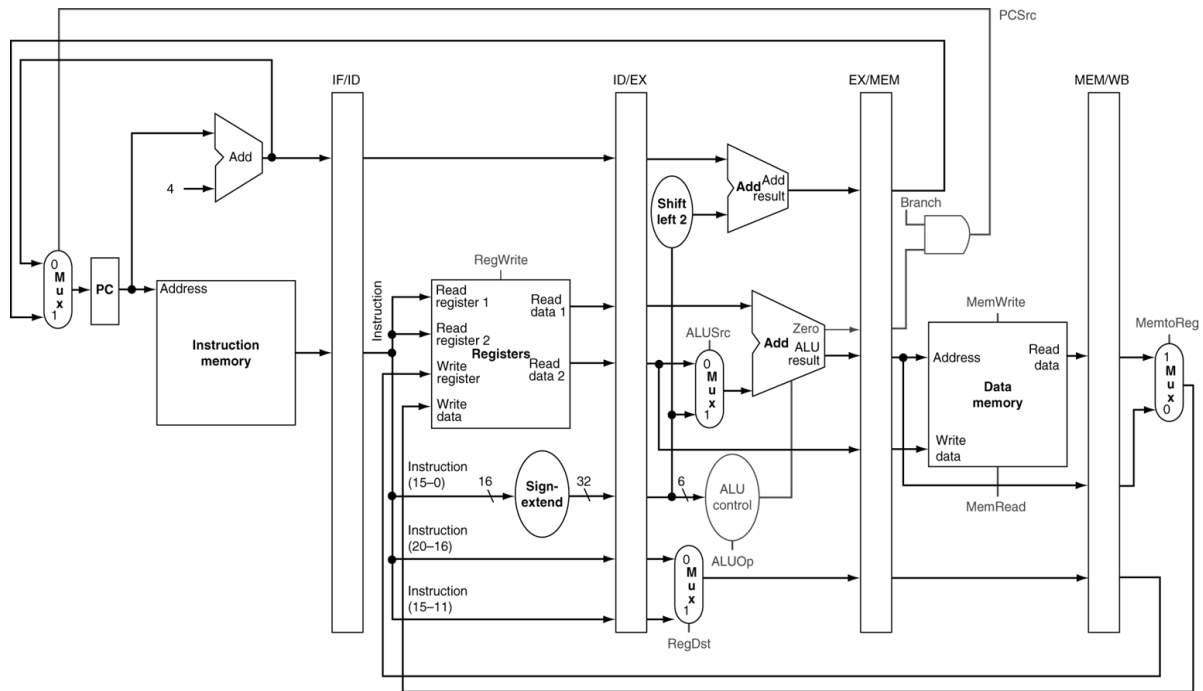
3. Execution/address calculation: The signals to be set are **RegDst**, **ALUOp**, and **ALUSrc**. The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.

4. Memory access: The control lines set in this stage are **Branch**, **MemRead**, and **MemWrite**. The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc in the figure selects the next sequential address unless control asserts Branch and the ALU result was 0.

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

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

FIGURE 4.48 A copy of Figure 4.16. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined



- ❖ Control signals derived from instruction as in single-cycle implementation, which shows the signals of different instructions that are active during its respective stages (refer table 4.49).
- ❖ Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.
- ❖ Since the control lines start with the EX stage, we can create the control information during instruction decode. The figure 4.50 shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline, just as the destination register number for loads moves down the pipeline.

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

FIGURE 4.49 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.

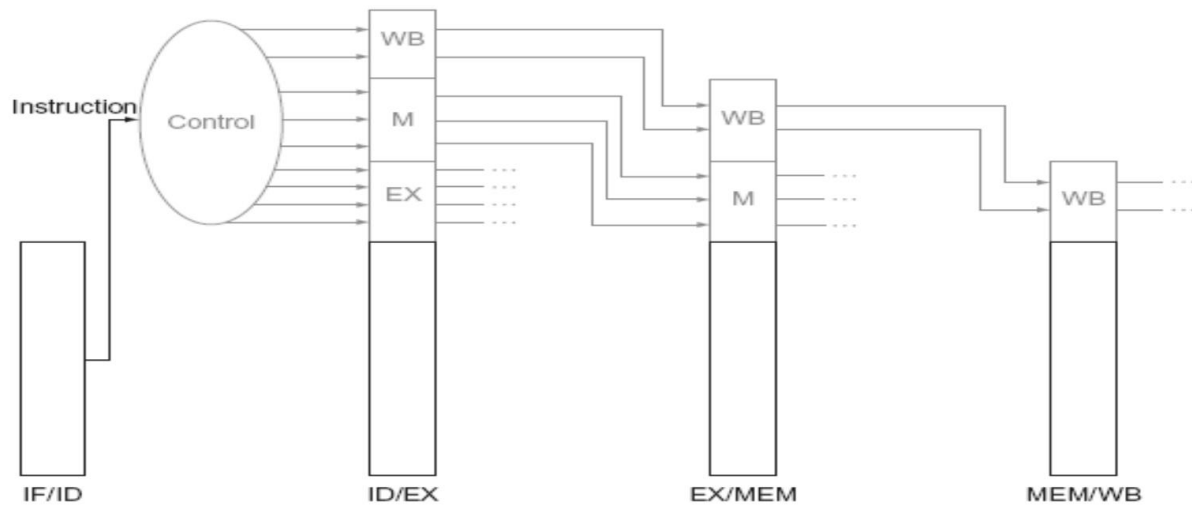


FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

SVCET

4. Explain Pipeline with example

“a pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion. Pipelining is an implementation technique in which multiple instructions are overlapped in execution.”

EXAMPLE: LAUNDRY EXAMPLE

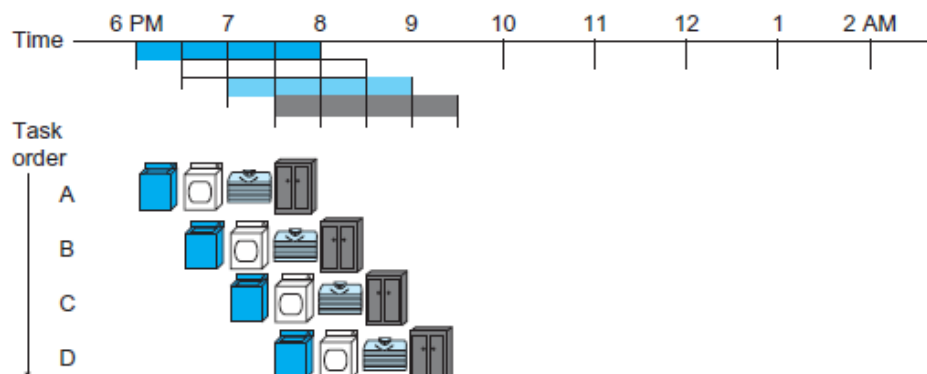
The non-pipelined approach to laundry would be as follows

- STAGES:**
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.

If there are 4 people to use a single washing machine, the following diagram shows the non-pipelined approach. Queue waiting time is long. Idle time of each stages are long between usages.



The pipelined approach, overlapping of each stages without disturbing the other stages. Now each person's waiting time is reduced and each stage idle time is reduced.



The same principles apply to processors where the pipeline instruction-execution is applied. The MIPS instructions classically take five steps or stages:

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

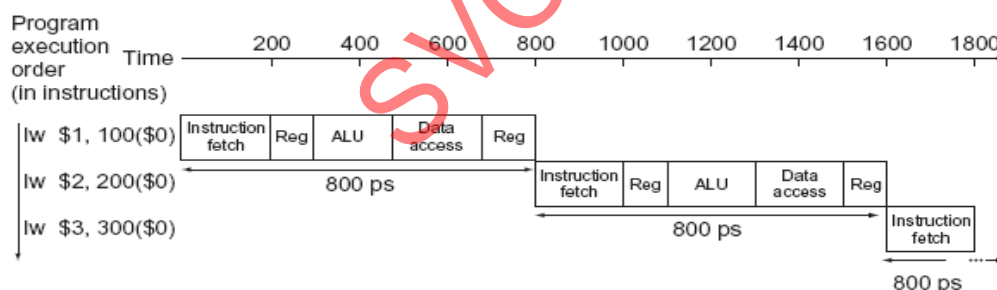
All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation.

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

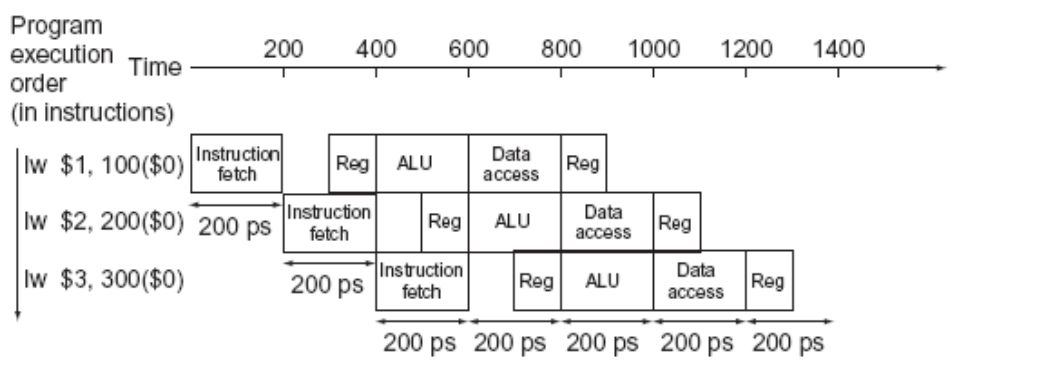
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.

EXAMPLE : sequences of load Instruction (Non-pipelined approach)



Pipelined approach - sequences of load Instruction, but stages are overlapped during execution as given below.



Total time taken for execution is 1400 ps, while in Non-pipelined approach is 2400 ps.

But as per formula

$$\begin{aligned}\text{Time taken for pipelined approach} &= \text{time taken (non-pipelined approach)} / \text{No of stages} \\ &= 2400 \text{ ps} / 5 \\ &= 480 \text{ ps}\end{aligned}$$

But the practical results show, it is 1400 ps.

So only when the No. of instructions in pipelined execution is high enough, the theoretical execution speed can be achieved or nearly achieved. Pipelining improves performance by increasing instruction throughput.

Designing instruction sets for pipelining

1. All MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage.
2. MIPS has only a few instruction formats, with the source register fields located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time, the hardware is determining what type of instruction was fetched.
3. Memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage.
4. Operands must be aligned in memory. Hence a single data transfer instruction will not require two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

5. Explain the different types of 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 3 types of Hazards:

- ❖ **Structural hazard:** When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- ❖ **Data hazard / pipeline data hazard:** When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- ❖ **Control hazard / branch hazard:** When the instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

Structural hazard

The hardware cannot support the combination of instructions that we want to execute in the same clock cycle. Assume that we had a single memory instead of two memories. If the pipeline had a fourth instruction, then in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

Data Hazards:

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Data hazards arise when an instruction's execution dependence on the data provided by another instruction which is still in the pipeline.

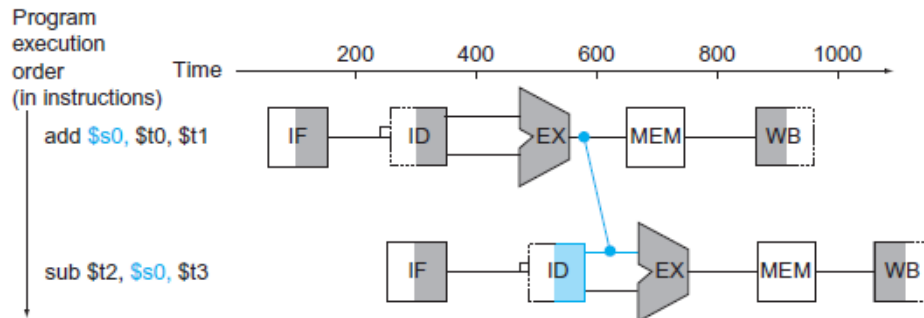
For example, suppose we have an add instruction followed immediately by a subtract instruction

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

The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

To resolve the data hazard, as soon as the ALU creates the sum for the add instruction, it can be given as an input for the subtract before writing it into s0. This is called **forwarding or bypassing**.

Figure below 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.



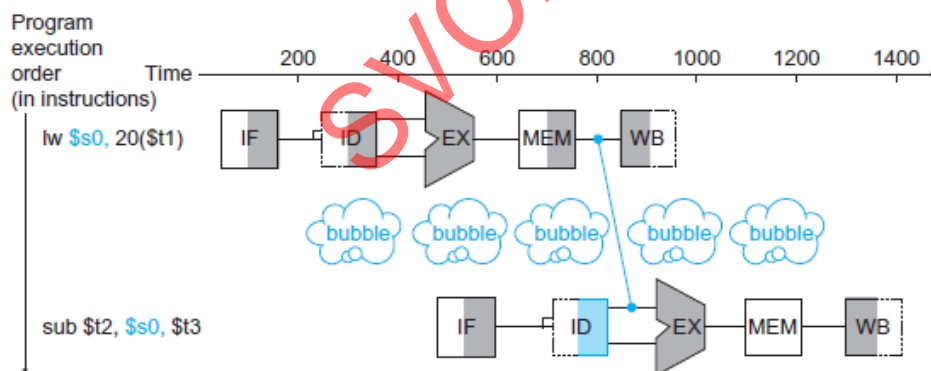
Graphical representation of forwarding

Forwarding cannot prevent all pipeline stalls.

Suppose the first instruction was a **load of \$s0 instead of an add**, the desired data would be available only after the **fourth stage of the first instruction** in the dependence, which is **too late** for the input of the **third stage of sub instruction**.

Hence even with forwarding, we would still have a one stage stall because of **load-use data hazard**. Load-use data hazard is a form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction. This figure shows below an important pipeline concept, called a **pipeline stall or bubble**.

Forwarding is harder if there are multiple results to forward per instruction.

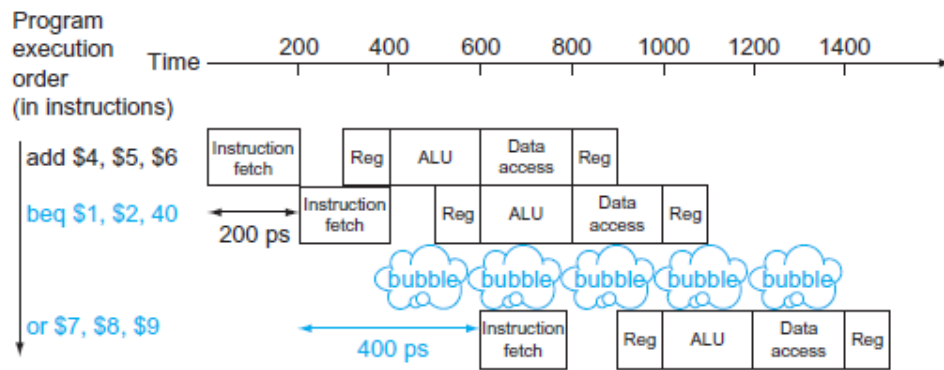


we need a stall even with forwarding when an R-format instruction following a load tries to use the data.

Control Hazard or Branch Hazard

Control hazard arises when a decision need to be made as to which instruction need to be executed based on the results of an instruction execution. The popular decision instruction is **the branch instruction**.

After receiving the branch instruction from memory, the pipeline would not know what instruction should be executed next. One possible solution is **to stall immediately after** fetching a branch instruction till the pipeline knows what instruction to fetch next.



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

This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated.

The **cost of this option is too high** for most computers to use and so a second solution to the control hazard is to **predict**. This option does not slow down the pipeline when you are correct.

- ❖ One simple approach is **to predict always that branches will not be taken**.
- ❖ If everything goes right, the pipeline proceeds at full speed.
- ❖ Only when branches are **taken, then the pipeline stalls** (prediction goes wrong).

In cases of loops, branches are **predicted to be taken** because they jump back to the top of the loop. **In case of Conditional statements**, branches are **predicted to be not taken** because they jump towards forward direction with reference to the program flow.

There are two types of branch prediction.

Static branch prediction

- ❖ Based on typical branch behavior of the branching statements.
- ❖ Example: loop and if-statement branches
 - loop: predict taken
 - If: Predict not taken

Dynamic branch prediction

- ❖ Hardware measures actual branch behavior of branches.
 - e.g., record recent history of each branch
- ❖ Assume future behavior will continue the trend and refers the stored history.
 - When history data goes wrong, its stall while re-fetching the new branch and updates the history accordingly

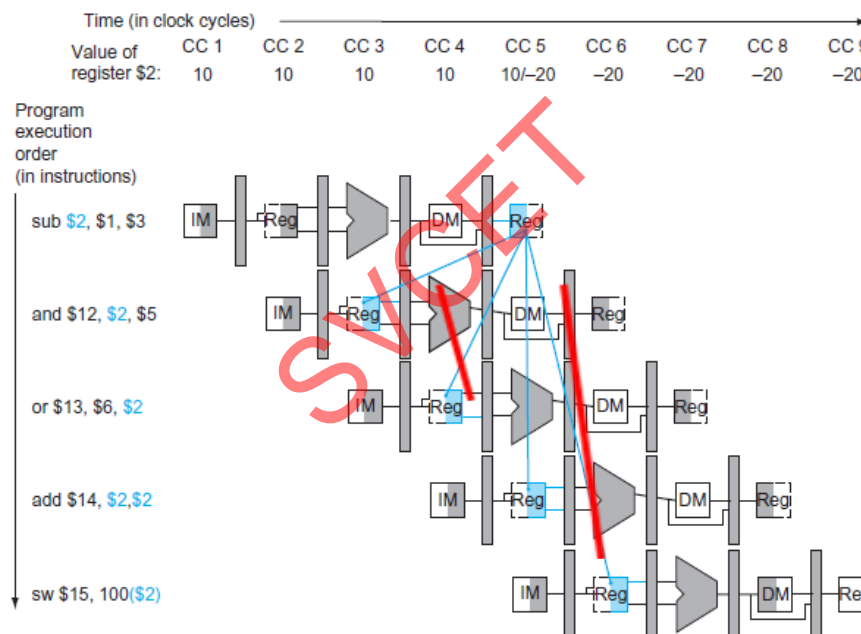
6. Explain data hazard and the ways to detect and handle the data hazards?

Consider this sequence:

```

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
  
```

The last four instructions are all dependent on the result in register \$2 of the first instruction. Register \$2 had the value 10 before the subtract instruction and -20 after subtraction. The programmer intends that -20 will be used in the following instructions that refer to register \$2.



Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences

The above diagram shows the dependence of each instruction with respect to the first instruction SUB and the result stored in \$2 register.

- As above, the **and & or** instructions would get the incorrect value 10 (assumed value of \$2 before execution of SUB instruction).
- Instructions that would get the correct value of -20 are **add & sw** (since both the instruction will need the value from / after CC5 (clock cycle)).

To detect and forward without stall:

To avoid stall, the result can have forwarded to and & or instruction from CC3 where the result is available at the end of EX stage. (As shown in above diagram).

To detect the data hazard, the source register dependence on any destination register of the previous instruction can be found by checking the following condition.

$$1a. \quad EX/MEM.RegisterRd = ID/EX.RegisterRs$$

$$1b. \quad EX/MEM.RegisterRd = ID/EX.RegisterRt$$

When the result is available in CC3, and the above condition is TRUE then the hazard is detected and forwarding process must be initiated to the next instruction, at the end of EX stage (CC3) of the current instruction. Here in this example condition 1a is TRUE.

$$1a. \quad EX/MEM.RegisterRd = ID/EX.RegisterRs = \$2 \text{ is TRUE.}$$

Similarly, When the result is available in CC4, and the below condition is TRUE then the hazard is detected and forwarding process must be initiated to the next instruction, at the end of MEM stage (CC4) of the current instruction.

$$2a. \quad MEM/WB.RegisterRd = ID/EX.RegisterRs$$

$$2b. \quad MEM/WB.RegisterRd = ID/EX.RegisterRt$$

Here in the given example Condition 2b is TRUE.

$$2b. \quad MEM/WB.RegisterRd = ID/EX.RegisterRt = \$2 \text{ is TRUE.}$$

Since the \$2 is the second operand of the OR instruction, 2b is TRUE. Only the above all 4 conditions will work when the read operation is Active, i.e., when the rd register does hold the result as ZERO.

- To forward/Bypass the result in CC3 or end of EX stage

Either condition 1a and EX/MEM.RegisterRd \neq 0 must be TRUE,
or

condition 1b and EX/MEM.RegisterRd \neq 0 must be TRUE.

- To forward/Bypass the result in CC4 or end of MEM stage

Either condition 2a and MEM/WB.RegisterRd \neq 0 must be TRUE,
or

condition 2b and MEM/WB.RegisterRd \neq 0 must be TRUE.

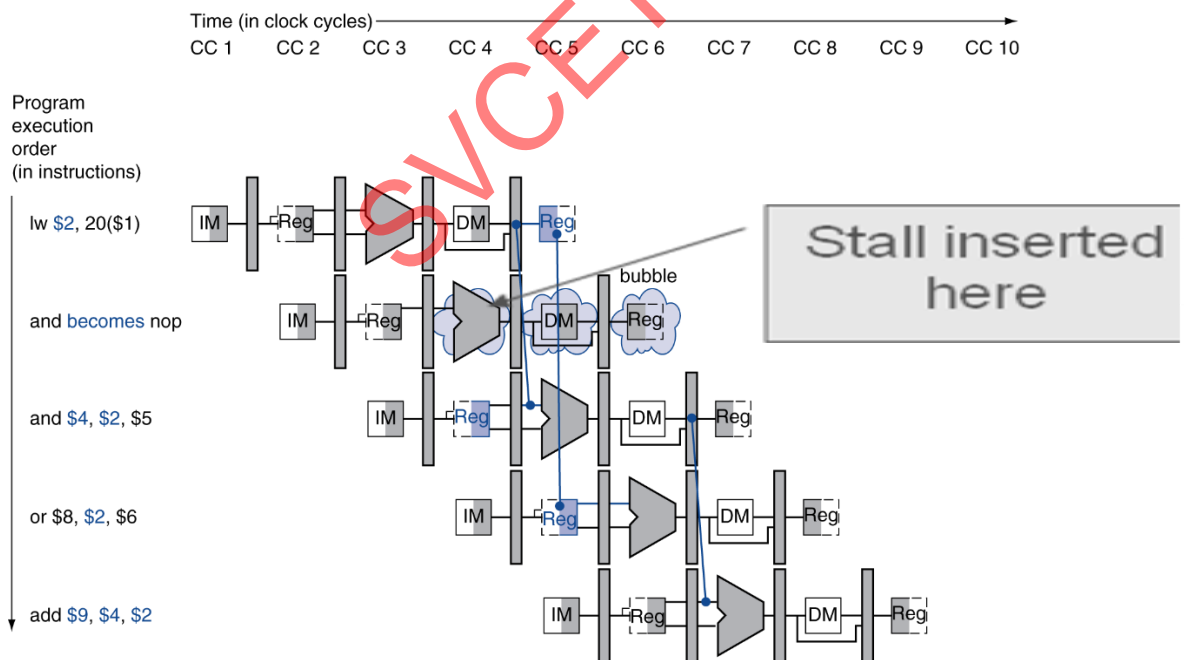
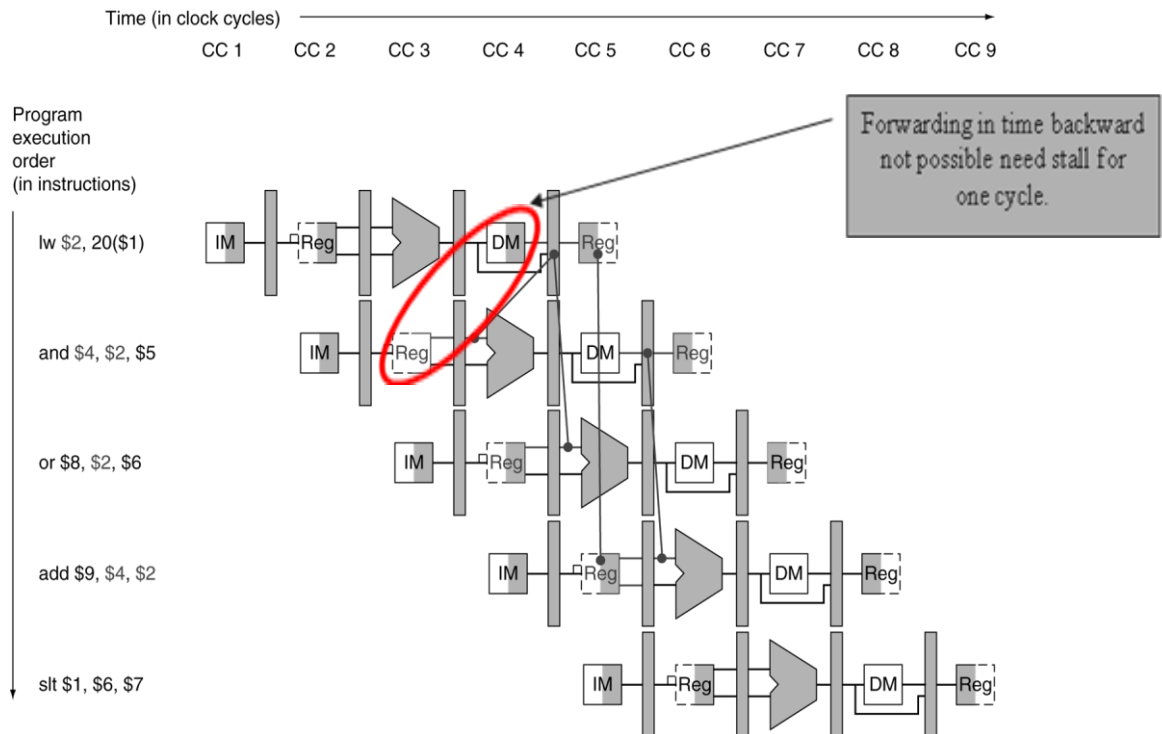
All the above conditions are checked by a special hardware called forwarding unit shown in figure below.



The following diagram will show the need for stall operation. Similarly as done before to detect the data hazard, the conditions for source and destination register in IF/ID and ID/EX registers are checked respectively, if the register number is same then stall is introduced.

Steps to introduce the stall

- Force control values in ID/EX register to 0.
- Introduction of NOP, stages EX, MEM and WB do NOP (no-operation) , in this case stall is introduced from the second instruction.
- Prevent update of PC and IF/ID register (in this case 3rd instruction will not be loaded immediately)
- Same instruction is decoded again. (second instruction is decoded ,as per given example) and the following instruction is fetched again. (3rd instruction is fetched as per given example).
- Now, the first instruction is moved to MEM cycle, so the result can be forwarded to the second instruction (given example).



7. Explain control hazard and the ways to detect and handle the control hazards

The delay in determining the proper instruction to fetch is called a *control hazard* or *branch hazard*. This delay arises as a result of executing a branch instruction. Figure below shows the impact of pipeline on the branch instruction.

There are two schemes to resolve control hazards.

- Assume branch not taken
- Reducing the delay of branches

Assume branch not taken

Predict that the branch will not be taken and thus continue execution down the sequential instruction stream.

If the branch is taken, the instructions that are being fetched and decoded must be flushed (Discarding instructions in a pipeline due to some unexpected event). Execution continues at the branch target.

Reducing the delay of branches

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.

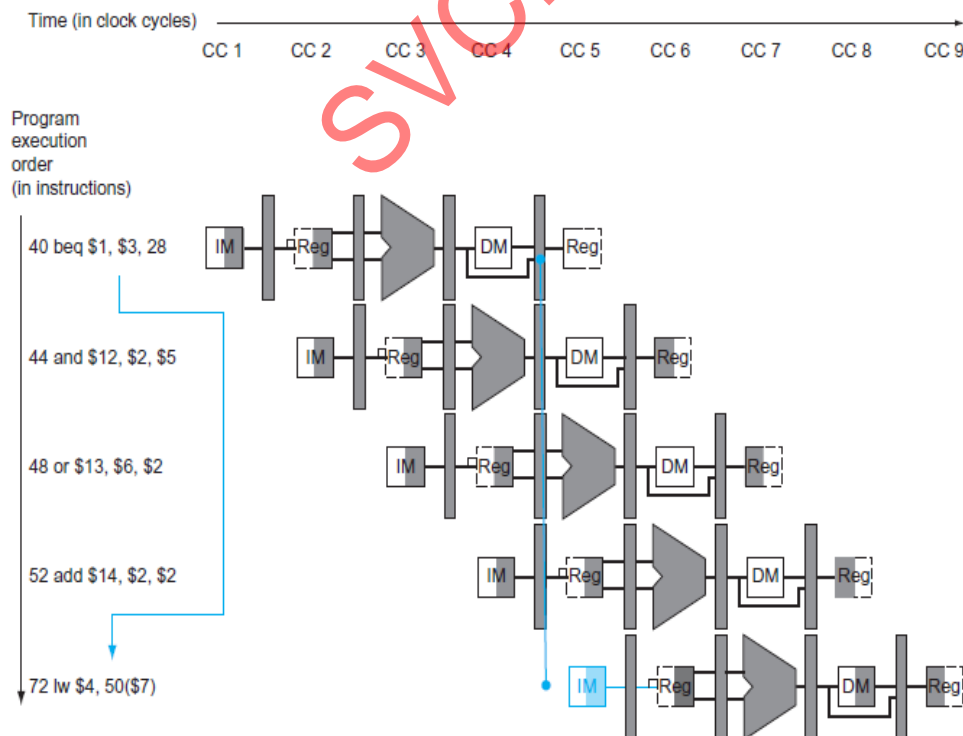


Fig. A The impact of the pipeline on the branch instruction

Moving the branch decision up requires two actions to occur earlier: computing the branch target address and evaluating the branch decision. The easy part of this change is to move up the branch address calculation.

The PC value and the immediate field are already present in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage. The branch target address calculation will be performed for all instructions, but only used when needed.

Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware. But it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched.

Dynamic branch prediction

Prediction of branches at runtime using runtime information. This approach involves looking up the address of the instruction to see if a branch was taken the last time this instruction was executed. If so, then, the instructions are fetched from the same place as last time.

This approach uses a **branch prediction buffer or branch history table**. A branch prediction buffer is a small memory which is identified by the lower portion of the address of the branch instruction. The memory contains a bit which says whether the branch was recently taken or not.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice. To overcome this 2-bit prediction schemes are used.

Figure B shows the finite-state machine for a 2-bit prediction scheme.

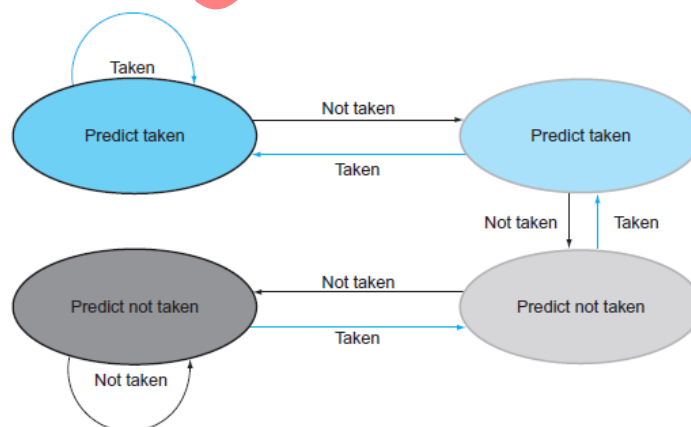


Figure B Finite-state machine for a 2-bit prediction scheme

A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage. If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known in the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Fig B.

8. What is meant by exception? How exceptions are handled in MIPS?

Exception is an unscheduled event that disrupts program execution. An interrupt is an exception that comes out of the processor. These are created to handle unexpected events from within the processor like arithmetic overflow.

How exceptions are handled in the MIPS architecture

The basic action that the processor must perform when an exception occurs is to 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.

Interrupt & 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. (Some architectures use the term interrupt for all exceptions.)

The Intel x86 uses interrupt. We follow the MIPS convention, using the term exception. External term interrupt only when the event is externally caused. Some examples are given below in table

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

How Exceptions Are Handled in the MIPS Architecture

The basic action that the processor must perform when an exception occurs is to 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 performing the necessary action the operating system can either terminate the program or continue with its execution, using the EPC to determine where to restart the execution of the program.

To handle an exception, an operating system should know the reason for the exception and the instruction that caused it. Two methods are used in MIPS architecture to communicate the reason for interrupt.

1. MIPS uses a Status register or cause register which has a field that holds the reason for exception.
2. Vectored interrupts – The operating system knows the reason for the exception by the address at which it is addresses are separated by 32 bytes or eight instructions, and the initiated. The operating system must record the reason for the exception.

Exceptions in a pipelined implementation

A pipelined implementation treats exceptions as a form of control hazard. For example, suppose there is an arithmetic overflow in an add instruction. We must flush the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address.

The easiest way to do this is to flush the instruction and restart it from the beginning after the exception is handled. The final step is to save the address of the offending instruction in the *exception program counter* (EPC). In reality, we save the address +4, so the exception handling the software routine must first subtract 4 from the saved value.

SVCET