UNIT – IV

PROCESSOR

T1 293 R1 421 Instruction Execution – Building a Data Path – Designing a control unit – Hardwired control. R1 425 T1 309 Microprogrammed control – R1 429 Pipelining – Data Hazard – T1 349 T1 316 Control Hazards. T1 361 R1 461

T1

INSTRUCTION EXECUTION

* Sequence of operations required to execute one instruction.

Ex: Add (R3), R1 — R3 – 1000 (pointer) — R1 – 30

→ Adds the contents of a memory location pointed to by R3 to Register R1.

Actions:

1. Fetch the instruction
2. Fetch the first operand
3. Perform the addition
4. Load the result into R1

Control sequence for execution of the instruction

Step	Action
1	PCout, MARin, Read, Select 4, Add, Zin
2.	Zout, PCin, Yin, WMFC
3	MDRout, IRin
4	R3out, IRin
5	R1out, Yin, WMFC
6	MDRout, Select Y, Add, Zin
7	Zout, R1in, End

WMFC – Waiting for Memory Function complete (signal from memory)

Instruction execution proceeds as follows:

Instruction Fetch Phase:

* In Step 1,
  - The instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory
  - The Select signal is set to Select 4, which

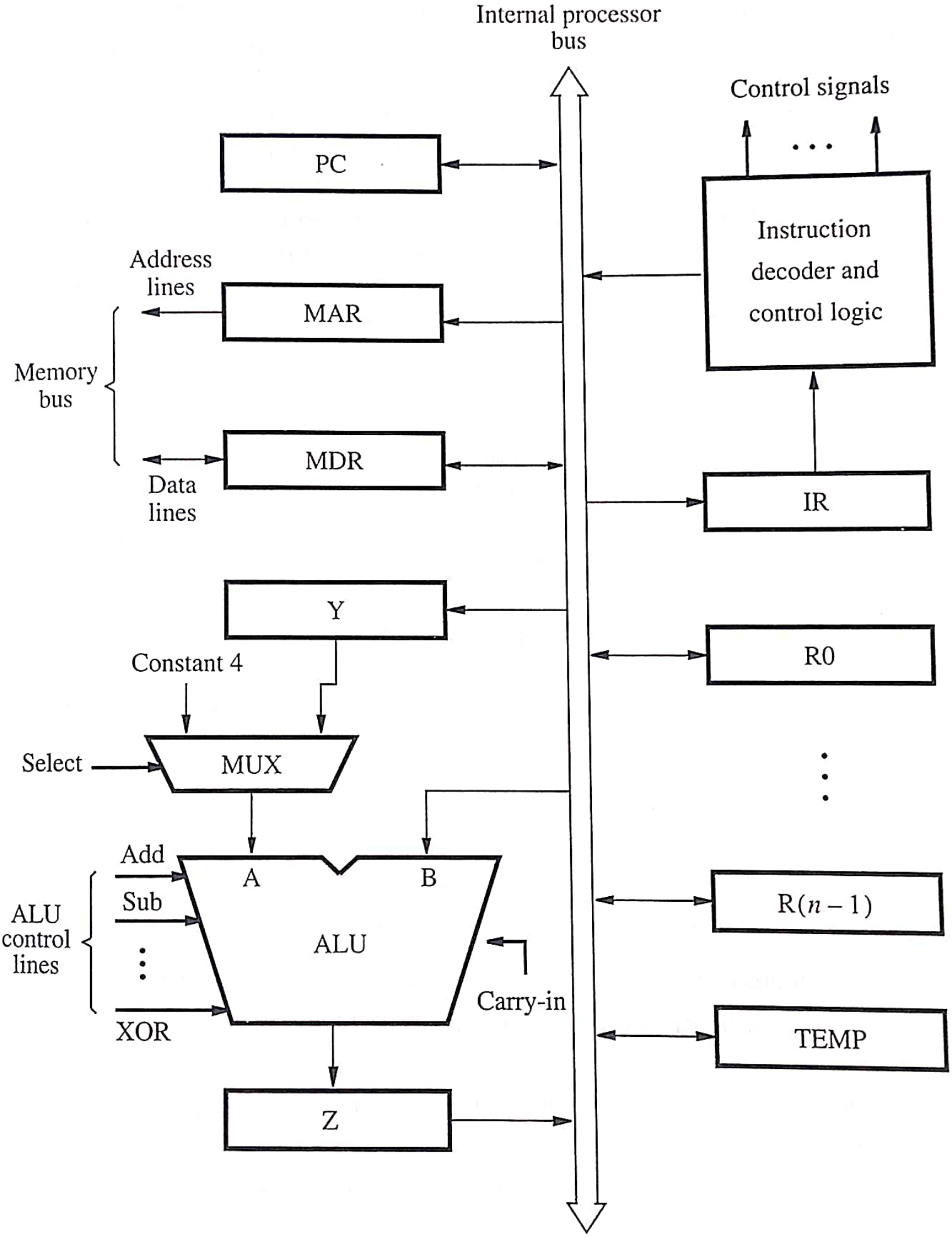


Figure 7.1 Single-bus organization of the datapath inside a processor.

causes the multiplexer MUX to select the constant 4.
- This value is added to the operand at input B, which is the contents of the PC, and the result is stored in register Z.

Step 2:
- The updated value is moved from register Z back into the PC.

Step 3:
- The word fetched from the memory is load into the IR.

Step 4:
- The instruction decoding circuit interprets the contents of the IR.
- This enables the control circuitry to activate the control signals
- The contents of register R3 are transferred to the MAR.
- a memory read operation is initiated.

Step 5:
- Then the contents of R1 are transferred to register Y, to prepare for the addition operation.

Step 6:
- When the Read operation is completed, the memory operand is available in register MDR and the addition operation is performed.
- The contents of MDR are gated to the bus and thus also to the B input of the ALU and register Y is selected as the second input to the ALU by choosing Select Y.

Step 7:
- The Sum is stored in register Z then transferred to R1.

* The End signal causes a new instruction fetch cycle to begin by returning to Step 1.

Branch Instructions:

* A branch instruction replaces the contents of the PC with the branch target address.
- The address is obtained by adding an offset X to the updated value of the PC

Control Sequence for an unconditional Branch instruction.

Step	Action
1	PCout, MARin, Read, Select4, Add, Zin
2	Zout, PCin, Yin, WMFC
3	MDRout, IRin
4	Offset-field-of-IRout, Add, Zin
5	Zout, PCin, End

* Processing starts with the fetch phase.
- The phase ends when the instruction is load in the IR.
* The offset value is extracted from the IR by the instruction decoding circuit

Step4:
- Since the value of the updated PC is already in register Y, the offset X is gated onto the bus. and an addition operation is performed.

Step 5:
- The result, which is the branch target address, is loaded into PC.

* The offset X used is the difference between the branch target address and the address immediately following the branch instruction.

→ The PC is incremented during the fetch phase, before knowing the type of instruction being executed.

Conditional branch:

* To check the status of the condition codes before loading a new value into the PC.

Ex: **Branch <0**

step 4 is replaced by

offset-field-of-IRout, Add, Zin, If N=0, then End.

* if N=0, the processor returns to step 1 after step 4.
* if N=1, step 5 is performed to load a new value into the PC.

BUILDING A DATA PATH

Data Path:

* The data path is the path way that the data takes through the CPU
* The datapath consists of functional units that perform addition, subtraction, Logical AND, OR, Shifting etc.

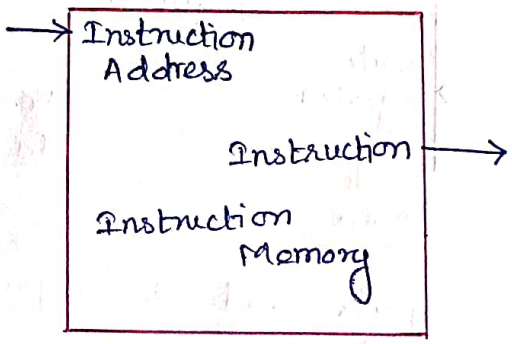
Datapath Elements:

The elements are:

i) Instruction Memory
ii) Program Counter (PC)
iii) Registers
iv) ALU
v) Adder
vi) Data Memory Unit
vii) Sign Extension Unit
viii) MUX / Multiplexer.

i) Instruction Memory:

* It is a memory unit to store the instructions of a program and supply instructions given an address.

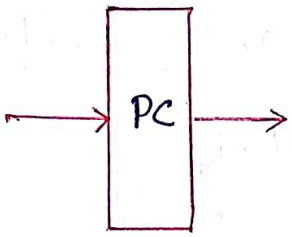


→ It is a combination Logic:
- The output at any time reflects the contents

of the location specified by the address input
– no read control signal is needed.

ii) Program Counter (PC):

* 32-bit register
* The register containing the address of the instruction in the program being executed
– does not need a write control signal.

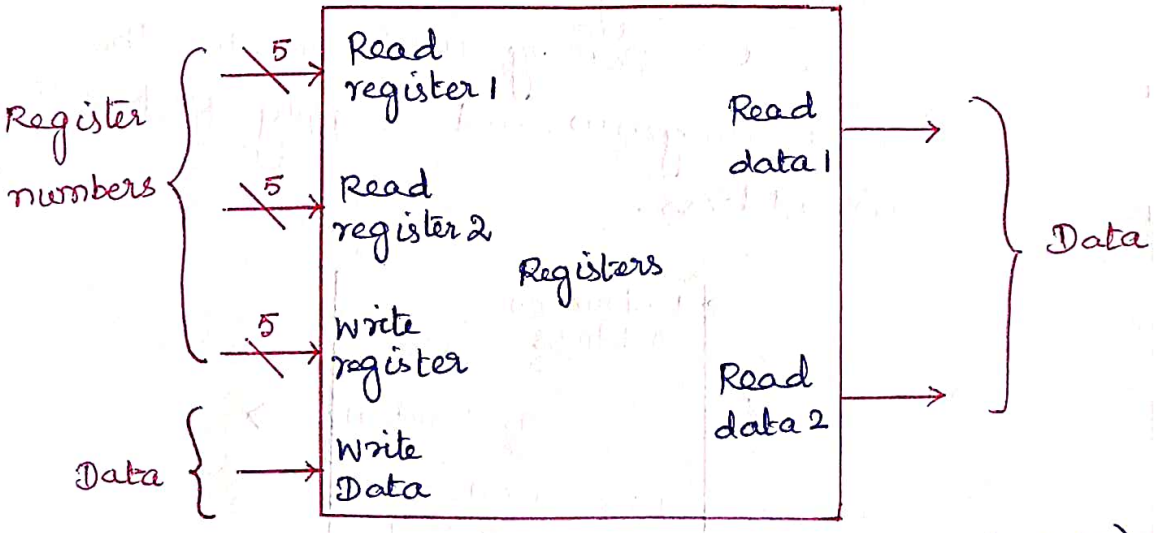


iii) Registers:

* Register file:

→ The processor's 32 general-purpose registers are stored in a structure
→ A collection of registers in which any register can be read or written by specifying the number of the register in the file.

* The register file contains all the registers and has two read ports and one write port.



* 4 inputs and 2 outputs (2 read data)
(2 read port,
1 write port)
1 write data)

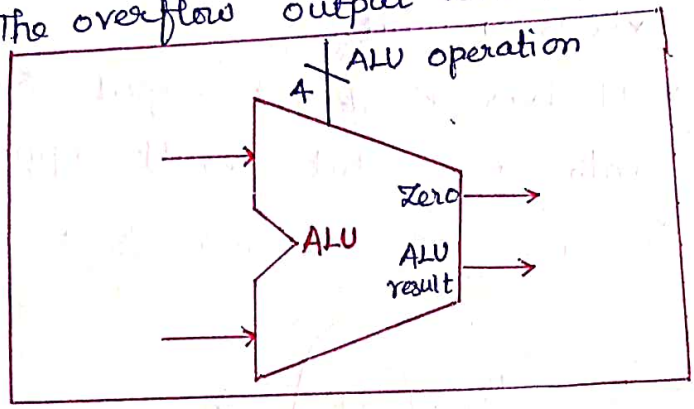
* a register write must be explicitly indicated by asserting the write control signal.

iv) ALU:

* It takes two 32-bit inputs
* produces a 32-bit result, as well as 1-bit signal if the result is 0.

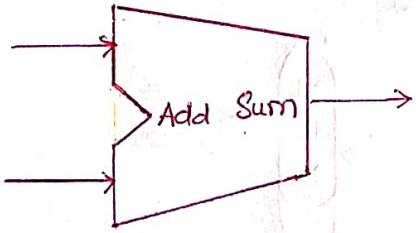
→ The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide.

- The overflow output will not be needed.



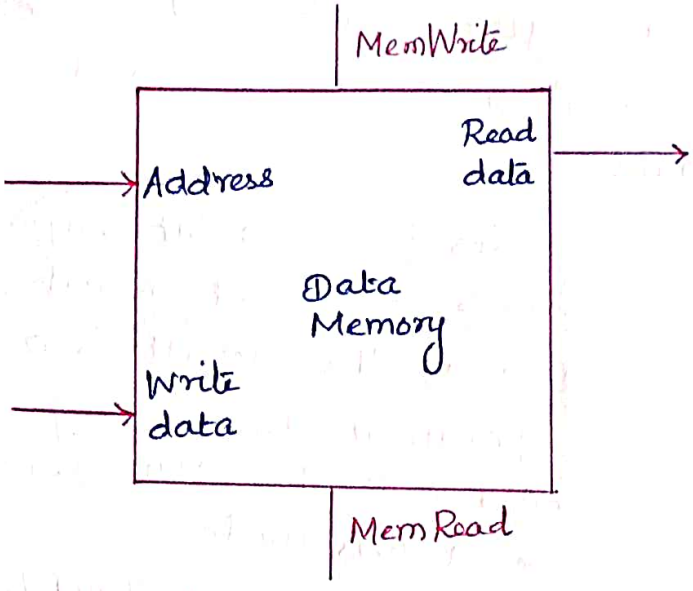
v) Adder:

* The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.
* To increment the PC to the address of the next instruction.
- Cannot perform the other ALU functions



vi) Data Memory Unit:

* The memory unit is a state element with inputs for the
  - i) address
  - ii) write data
- single output for the read result
* The data memory has read and write control signals.

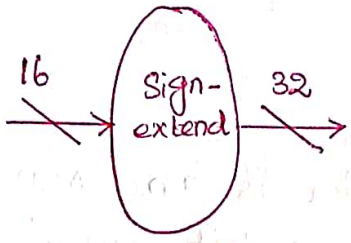


- The memory unit needs a read signal
- reading the value of an invalid address can cause problems.

vii) Sign Extension Unit:

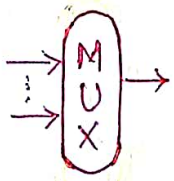
* It has a 16-bit input that a sign-extended into a 32-bit result appearing on the output

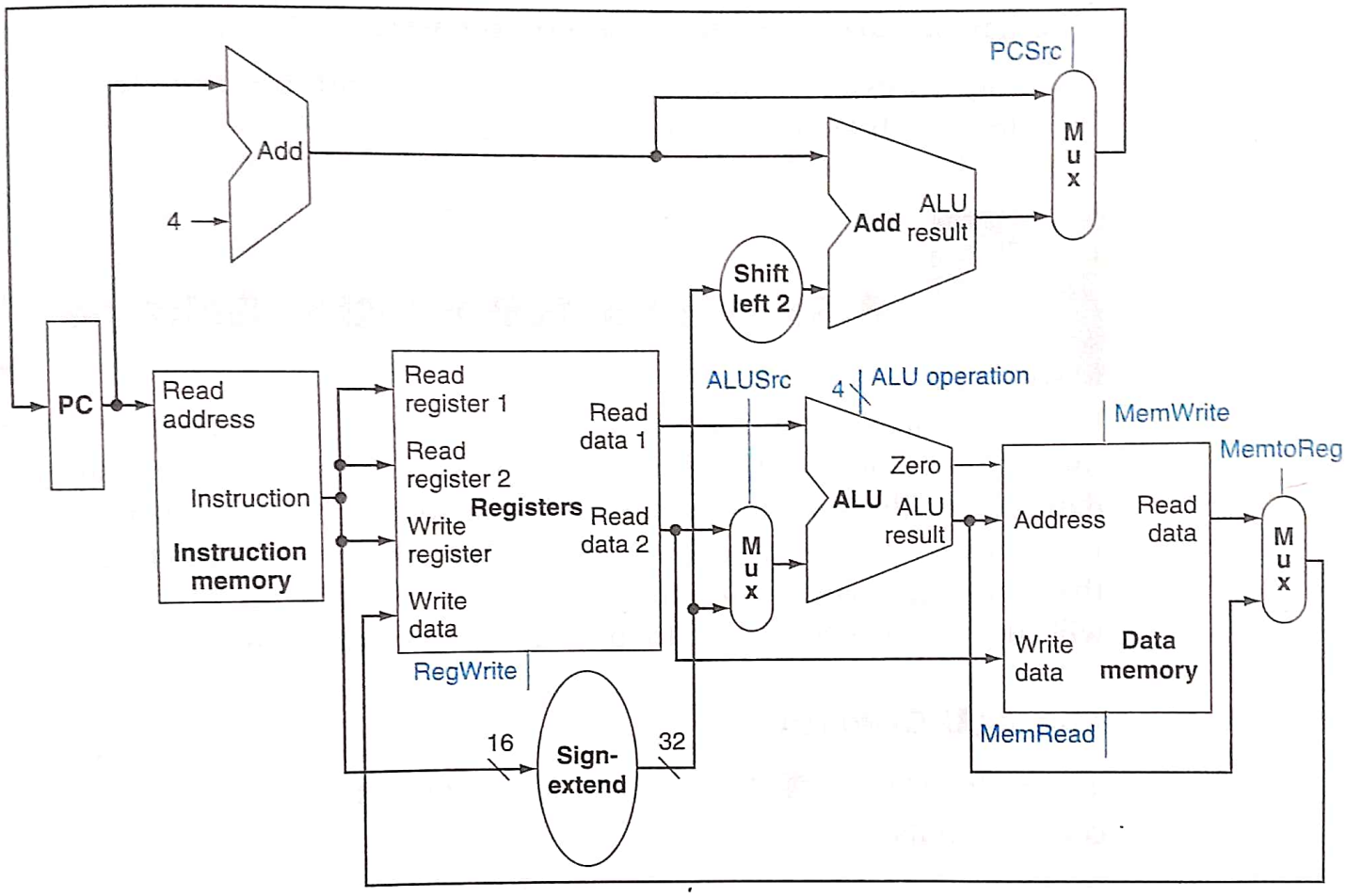
- To increase the size of a data item



viii) MUX / Multiplexer:

* It is also called as data selector
* selects from among several inputs based on the setting of its control lines.





Simple data path for the MIPS architecture.

MIPS – Microprocessor without Interlocked Pipe Stage

* This datapath can execute the basic instructions
  - load-store word,
  - ALU operations &
  - branches.

  in a single clock cycle.

DESIGNING A CONTROL UNIT

* To design the main control unit, three instruction classes are used.

They are:

1. R-type instructions
2. Branch instructions
3. Load-Store instructions

1. R-type Instruction:

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

* opcode → 0 (value - zero)
* Three register operands:
  sources : rs, rt
  destination : rd
* ALU operations → funct
  Is decoded by the ALU control
  Add, sub, or etc.
* Shifting operation → shamt (shift amount)

2. Branch instructions:

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

* opcode → 4
* The registers rs and rt are source registers that are compared for equality.
* 16 bit address field is sign extended, shifted and added to the PC to compute the branch target address

3. Load or Store Instruction:

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

* opcode → 35_{ten} (load)

43_{ten} (store).

* The register rs is the base register
  - It is added to the 16 bit address field to form the memory address

* **Load:**
  * rt is the destination register for the loaded value

* **Store:**
  * rt is the source register whose value should be stored into memory.

Simple Datapath with the control unit:

* The input to the control unit is the 6-bit opcode field from the instruction.
* The output of the control unit consists of 8 signals.

→ Three 1 bit signals → used to control multiplexors.

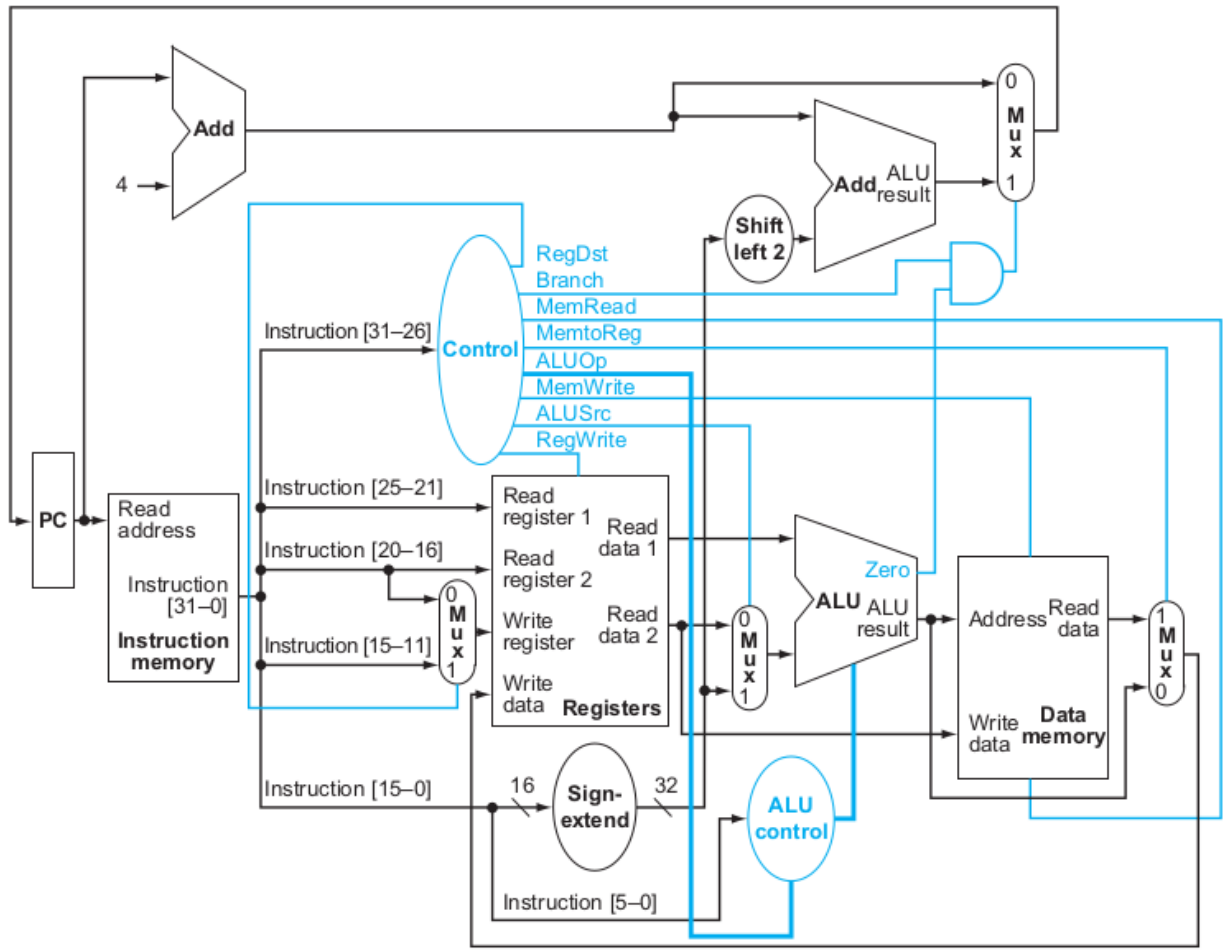
→ Three signals (RegWrite, MemRead and MemWrite)
  - controlling reads and writes in the register and data memory.

→ 1-bit signal (branch) → used to determine whether to possibly branch.

→ 2-bit control signal (ALUOP) → used for ALU.

AND gate:
- used to combine the branch control signal and the zero output from the ALU.
- output controls the selection of the next PC.

The simple datapath with the control unit



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

i) Operation of the datapath for an R-type instruction

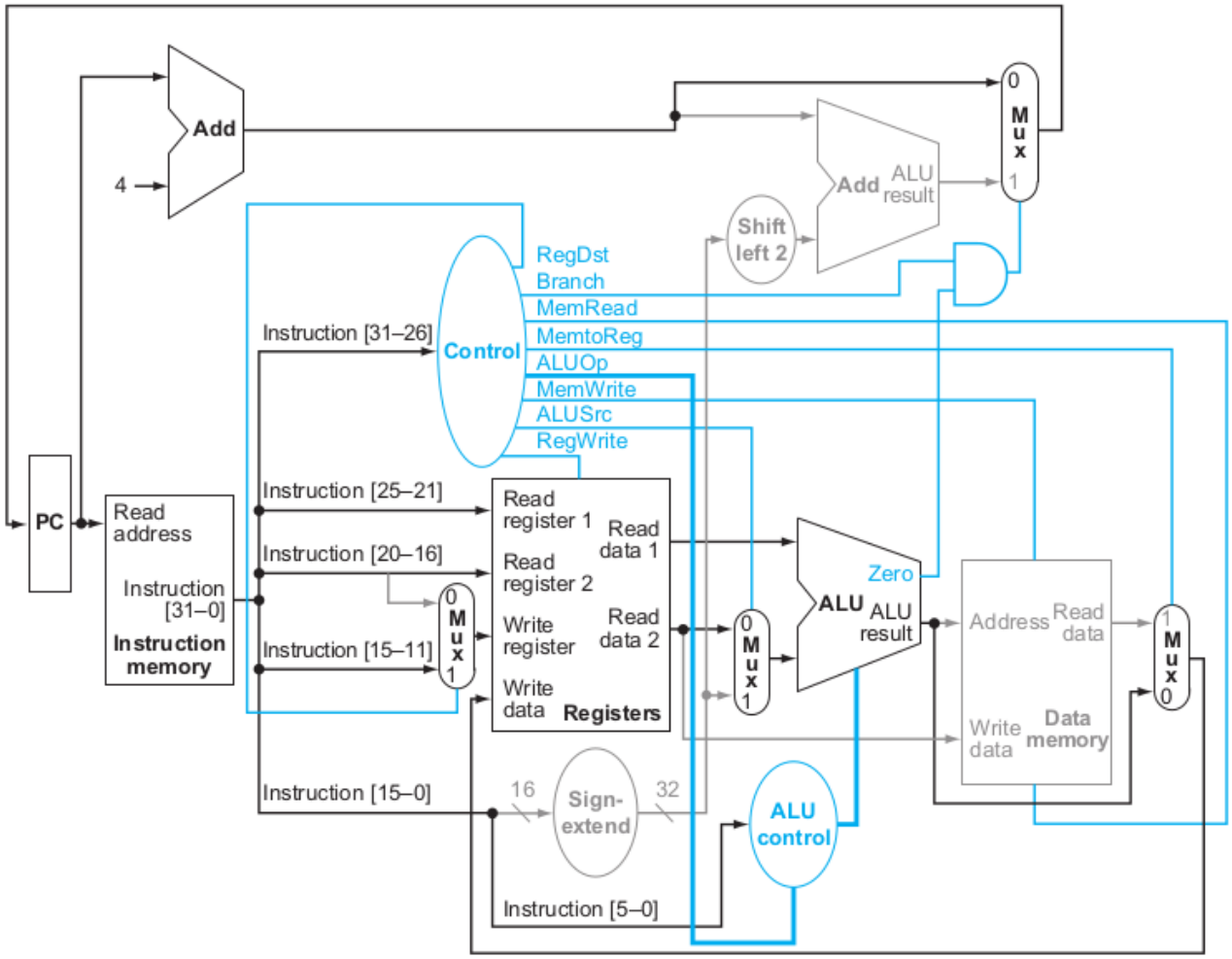
Ex:

add $t1,$t2,$t3

Four steps to execute the instruction

1. The instruction is fetched, and the PC is incremented.
2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct fi eld, of the instruction) to generate the ALU function.
4. Th e result from the ALU is written into the register fi le using bits 15:11 of the instruction to select the destination register ($t1).

The datapath in operation for an R-type instruction



ii) The execution of a load word

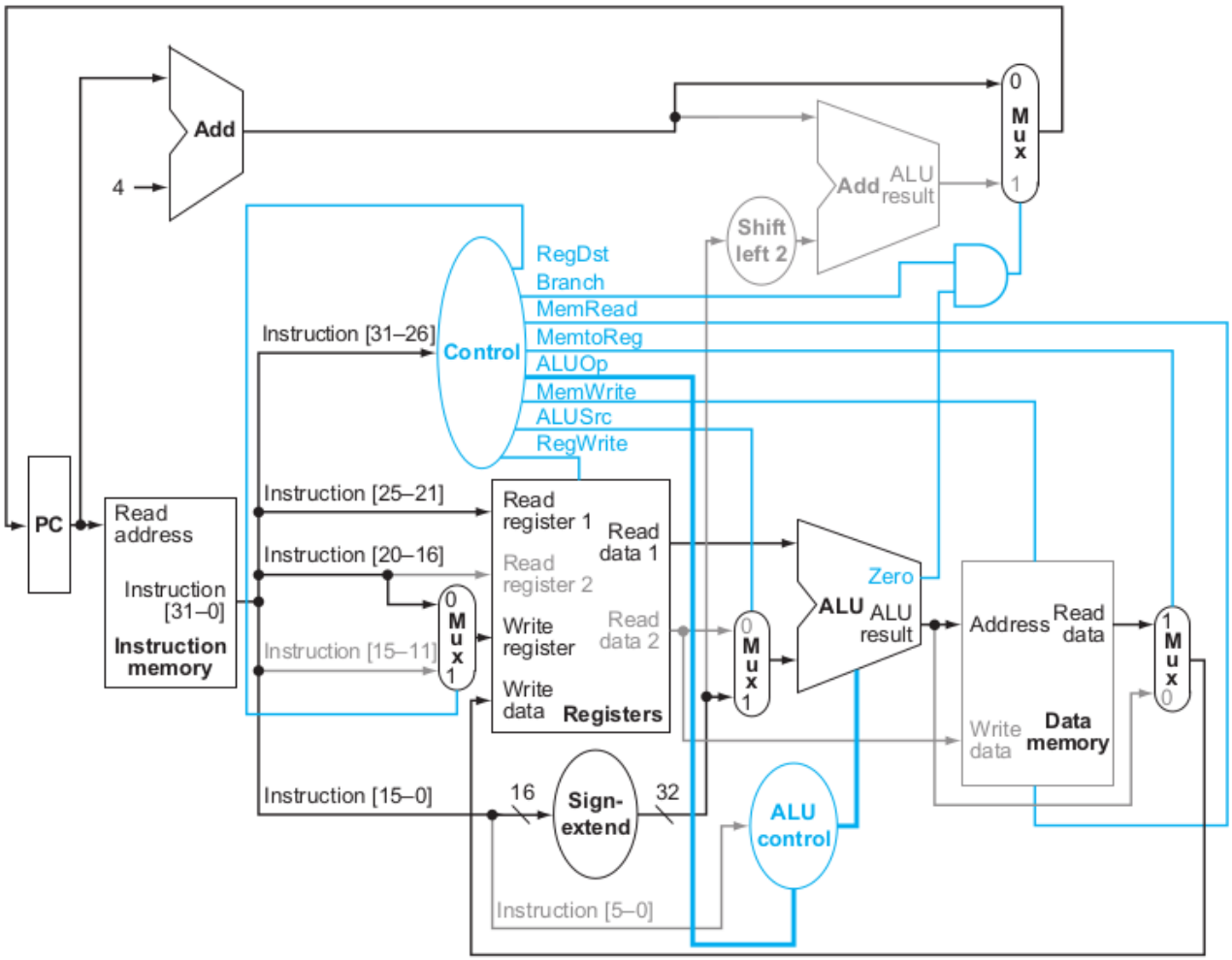
Ex:

lw $t1, offset($t2)

Five steps :

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

The datapath in operation for a load instruction



The datapath in operation for a load instruction

- The control lines, datapath units, and connections that are active are highlighted.
- A store instruction would operate very similarly.
- The main diff erence would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register fi le would not occur.

iii) Operation of the branch-on-equal instruction

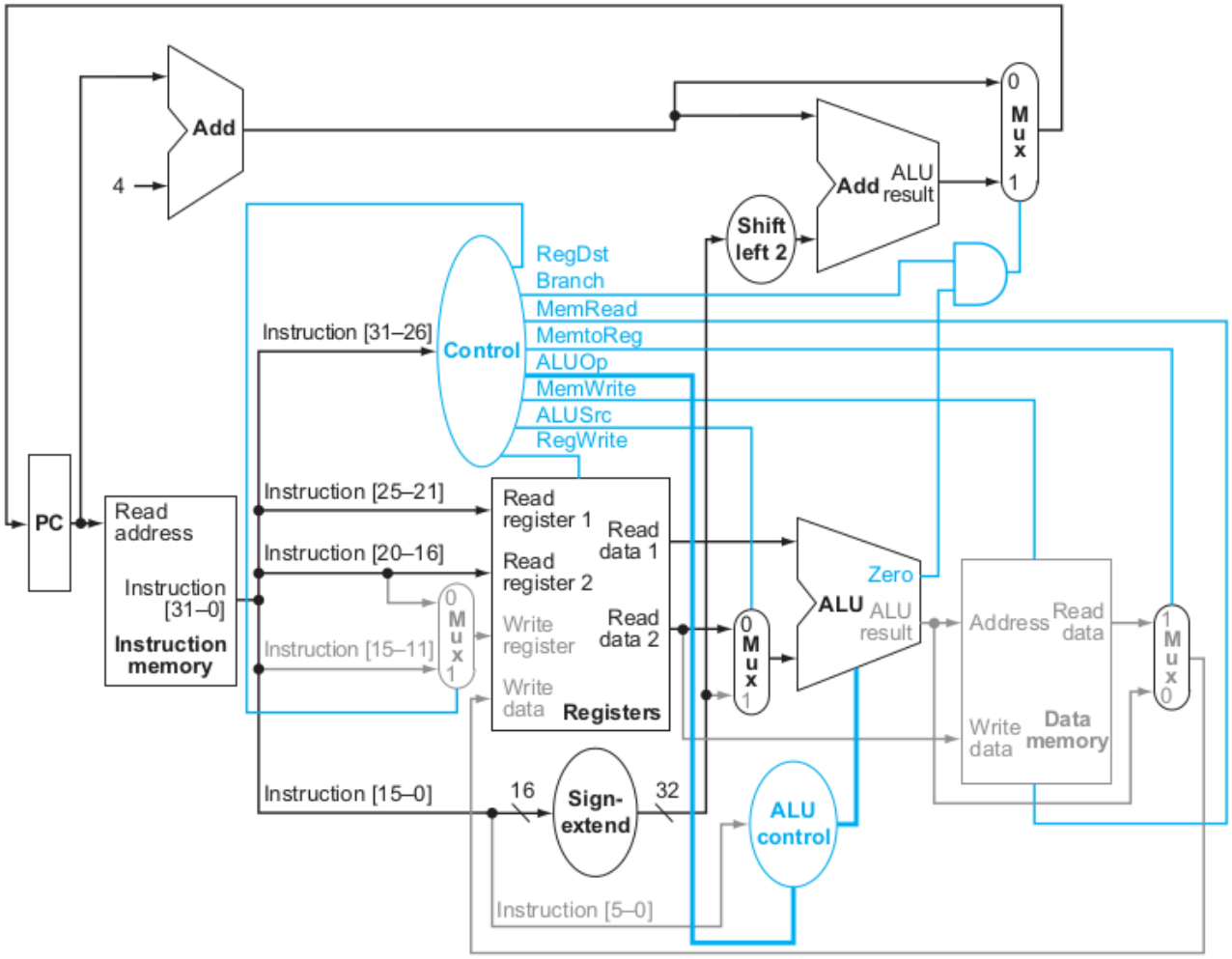
Ex:

beq $t1, $t2, offset

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. Th e 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) shift ed 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.

The datapath in operation for a branch-on-equal instruction



HARDWIRED CONTROL

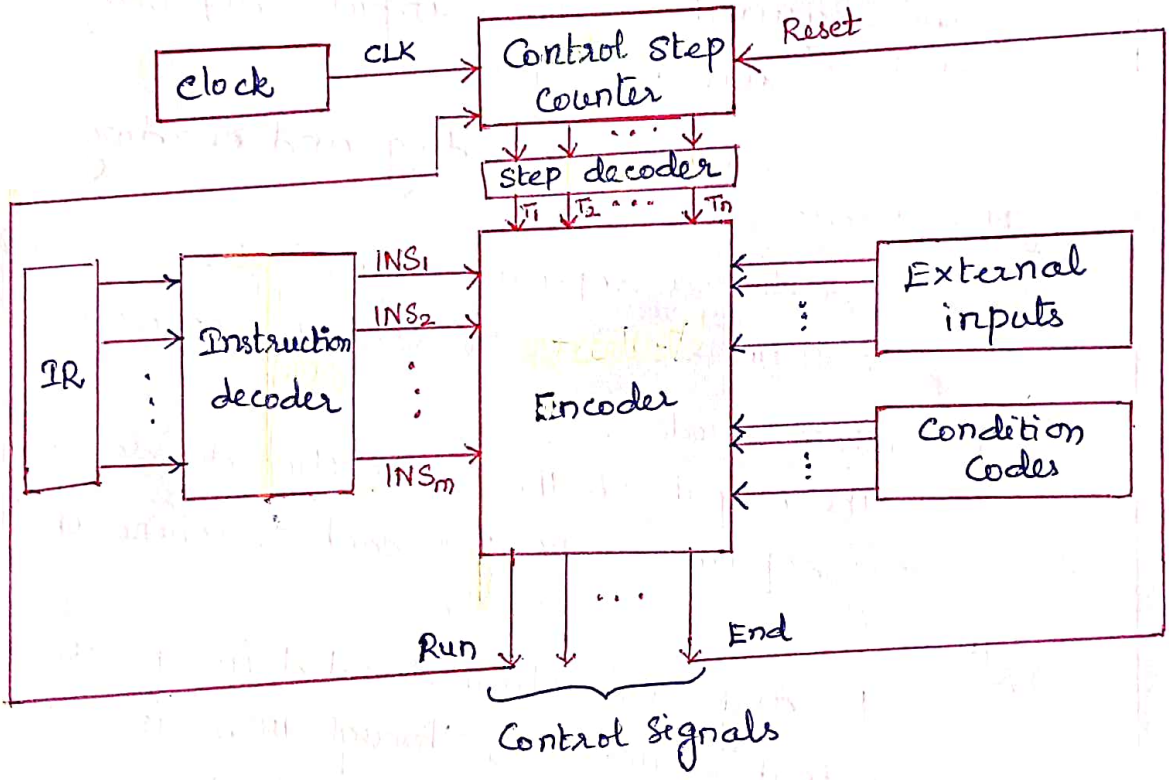
* To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence

Two Approaches:

i) Hardwired Control

ii) Microprogrammed Control

Control Unit Organization



* Each step in the control sequence for the execution of an instruction is completed in one clock period.

Counter:

* Used to keep track of the control steps
* Each state, or count, of this counter corresponds to one control step.

Determination of the required control signals :

information:

* Contents of the control step counter
* Contents of the instruction register
* Contents of the condition code flags
* External input signals
  - MFC
  - interrupt requests

Decoder/encoder :

* It is a combinational circuit that generates the required control inputs, depending on the state of all its inputs.

Step Decoder :

* Provides a separate signal line for each step, or time slot, in the control sequence.

Instruction decoder :

* The output of the instruction decoder consists of a separate line for each machine instruction

IR :

* For any instruction loaded in the IR, one of the output lines INS, through INS_m is set to 1, and all other lines are set to 0.

Encoder:

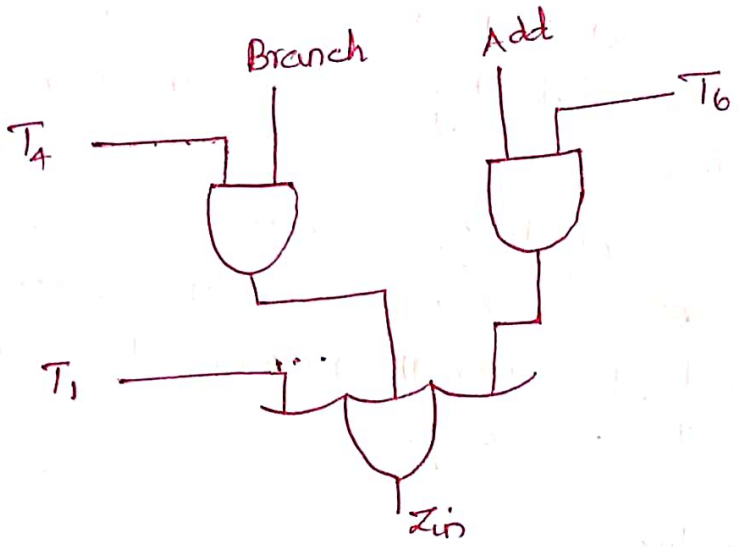
* The input signals to the encoder block are combined to generate the individual control signals Y_{in}, PCout, Add, End and so on.

Ex:

Zin

Generation of the Z_{in} Control Signal for the Processor

$$Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR$$



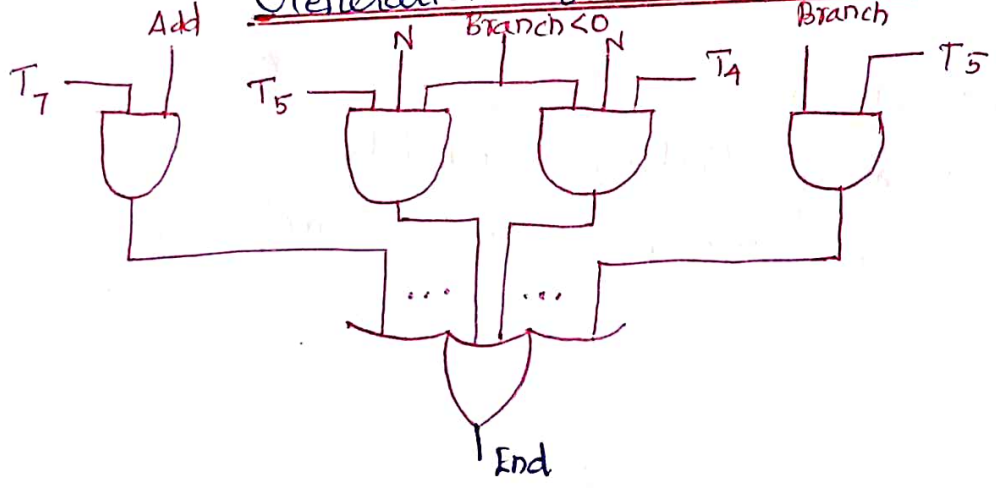
* This signal is asserted during time slot T_1 for all instructions, during T_6 for an Add instruction, during T_4 for an unconditional branch instruction and so on.
- The logic function for Zin is derived from the control sequences.

End:

* A circuit that generates the End control signal from the logic function

$$End = T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \bar{N}) \cdot BRN + \ldots$$

Generation of the End control signal.



End signal:

* starts a new instruction fetch cycle by resetting the control step counter to its starting value.

RUN:

* Control signal
* When set to 1, RUN causes the counter to ~~the~~ incremented by one at the end of every clock cycle.
* When set to 0, the counter stops counting.

– This is needed whenever the WMFC signal is issued, to cause the processor to wait for the reply from the memory.

Control hardware:

* It can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes, and the external inputs.
* The outputs of the state machine are the control signals.

→ The sequence of operations carried out by the machine is determined by the wiring of the logic elements, hence the name "hardwired".

* A controller that uses this approach can operate at high speed.

* It has little flexibility
* The complexity of the instruction set it can implement is limited.

MICROPROGRAMMED CONTROL

* Control signals are generated by a program similar to machine language programs.

Terms:

Control word:

- A word whose individual bits represent the various control signals.
* Each of the control steps in the control sequence of an instruction defines a unique combination of 1s and 0s in the CW.

Micro-routine:

* A sequence of control words corresponding to the control sequence of a machine instruction constitutes the micro-routine for that instruction.

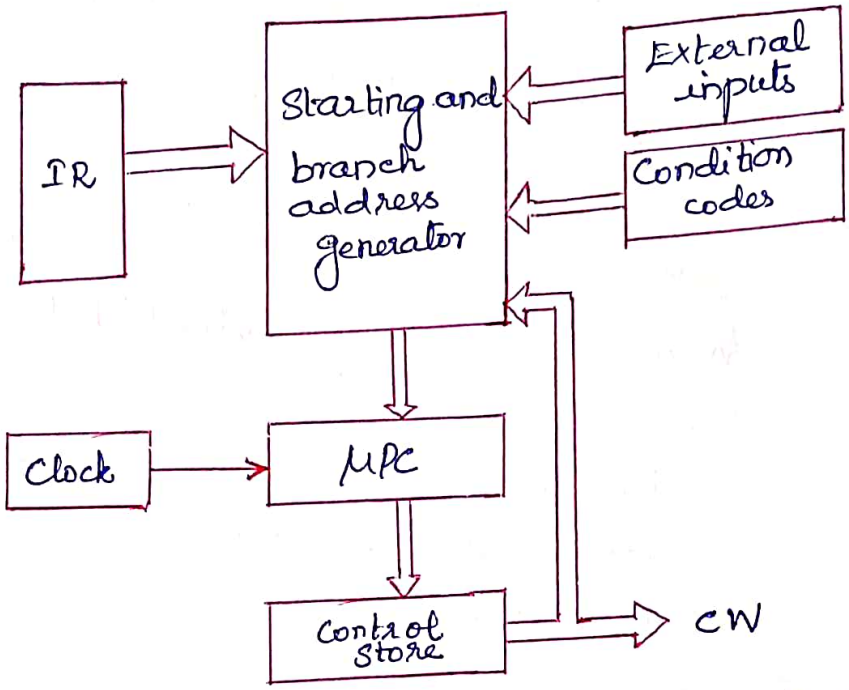
Microinstructions:

* The individual control words in the micro-routine.

Control Store:

* The microroutines for all instructions in the instruction set of a computer are stored in a special memory called the control store.

Organization of the Control Unit



* The control unit can generate the control signals for any instruction by sequentially reading the CWs of the corresponding microroutine from the control store.

MicroProgram counter (μPC):

* To read the control words sequentially from the control store.

IR:

* Every time a new instruction is loaded into the IR, the output of the block labeled "starting address generator" is loaded into the μPC.

clock:

* The μPC is then automatically incremented by the clock, causing successive microinstructions to be read from the control store.

→ Hence, the control signals are delivered to various parts of the processor in the correct sequence.

* Microinstructions:

(i) Straightforward way:

- To assign one bit position to each control signal.

Drawback:

- long microinstructions

* 42 control signals are needed.
* 42 bits would be needed in each microinstruction.

(ii) Binary coding Scheme:

* Signals can be grouped so that all mutually exclusive signals are placed in the same group.
- At most one microoperation per group is specified in any microinstruction.
* Represent the signals within a group.

EX: Partial format for field-encoded microinstructions

F1	F2	F3	F4	F5	...
F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)	
0000: No transfer 0001: PCout 0010: MDRout 0011: Zout 0100: R0out 0101: R1out 0110: R2out 0111: R3out 1010: TEMPout 1011: offsetout	000: No transfer 001: PCin 010: IRin 011: Zin 100: R0in 101: R1in 110: R2in 111: R3in	000: No transfer 001: MARin 010: MDRin 011: TEMPin 100: Yin	0000: Add 0001: Sub ⋮ 1111: XOR 16 ALU functions	00: No action 01: Read 10: Write	

F6	F7	F8	...
F6 (1 bit)	F7 (1 bit)	F8 (1 bit)	
0: Select Y 1: Select 4	0: No action 1: WMFC	0: Continue 1: End	

* 20 bits are needed to store the patterns for the 42 signals

Encoded Scheme

Highly encoded scheme
↓
Vertical organization
→ Use compact code to specify only a small number of control functions in each microinstruction
→ slower operating speed
→ fewer bits are required
– less hardware is needed for execution.

Minimally encoded scheme
↓
Horizontal organization
→ Many resources can be controlled with a single microinstruction
→ higher operating speed
→ allows parallel use of resources.

MicroProgram Sequencing:

* The simple microprogram requires only straight forward sequential execution of microinstructions, except for the branch at the end of the fetch phase.
  – μpc governs the sequencing.
* Some branching capability within the microprogram can be introduced through special branch microinstruction
  – Standard software techniques can be used for writing microprograms.

Disadvantages

i) Large total number of microinstructions and a large control store
ii) Execution time is longer

Bypass the Branching:

i)* Branch Address Modification Using Bit-ORing
  – Use an OR gate to change LSB of address to 1
ii)* Use 2 conditional branch microinstructions
iii)* To include 2 next address fields within a branch microinstn

Wide - Branch Addressing :

* The instruction decoder generates the starting address of the microroutine that implements the instruction that has just been loaded into the IR.
  - IR contains the instruction, for which the instruction decoder generates the microinstruction.
  - The address cannot be loaded as is into the microprogram counter.
  - The Source operand of the instruction can be specified in any of several addressing modes.
  - The bit-ORing technique can be used to modify the starting address generated by the instruction decoder to reach the appropriate path.

Use of WMFC :

* The WMFC signal means that the microinstruction may take several clock cycles to complete.
  - If the branch is allowed to happen in the first clock cycle, the microinstruction would be fetched and executed prematurely.
  - → To avoid this problem, the branch must not take place until the memory transfer in progress is completed, that is, the WMFC signal must inhibit any change in the contents of the microprogram counter during the waiting period.

Microinstructions with Next-Address Field.

Branch microinstructions:

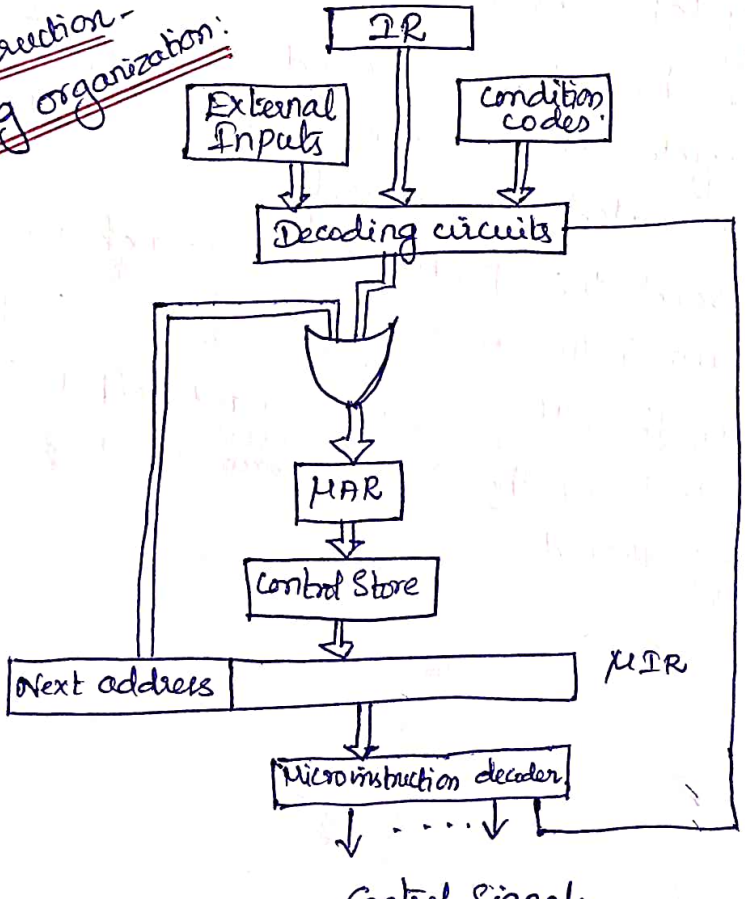
- perform no useful operation in the datapath
- they are needed only to determine the address of the next microinstruction.

* To include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.
- Address field of 12 bits is required.
  1/6 of the control store capacity.
* Each instruction contains the address of the next instruction
- no need for a counter to keep track of sequential addresses.

MAR:

* The μPC is replaced with a microinstruction address register (MAR), which is loaded from the next-address field in each microinstruction.

Microinstruction-Sequencing organization:



OR gate:

* The next address bits are fed through the OR gates to the μAR, so that the address can be modified on the basis of the data in the IR, external inputs and condition codes.

Decoding circuits:

* generate the starting address of a given microroutine on the basis of the op code in the IR.

Fig: a) Format for microinstructions.
b) Ex: Implementation of the microroutine using a next-microinstruction address field.

Prefetching Microinstructions:

* Faster operation is achieved if the next microinstructions is prefetched while the current one is being executed.
  - the execution time can be overlapped with the fetch time.

Difficulty:

- fetch must be repeated with correct address, which requires more complex hardware.

Emulation:

Main function of Microprogrammed Control
- simple
- flexible
- relatively inexpensive execution of machine instruction.

* Programs written in the machine language of M_2 can run on computer M_1.
  - M_1 emulates M_2.
* Emulation allows to replace obsolete equipment with more up-to-date machines.

x ——— x

PIPELINING

→ exploits parallelism among the instrs in a sequential instruction stream.

→ increases the no. of simultaneous executing instrs

* Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
  - The pipelined approach takes much less time
  - The stages in pipelining are operating concurrently.

Pipeline instruction execution:

MIPS instructions – Five steps:

1. Fetch instruction from memory
2. Read registers while decoding the instruction. Reading and decoding occur simultaneously.
3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

EX: Total Time for each instruction calculation

Instruction class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
Load word(lw)	200ps	100ps	200 ps	200ps	100ps	800ps
Store word(sw)	200ps	100ps	200 ps	200ps		700ps
R-format(add, sub, AND, OR, slt	200ps	100ps	200ps		100ps	600ps
Branch(beq)	200ps	100ps	200ps			500ps

Time between instructions on the pipelined processor:

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

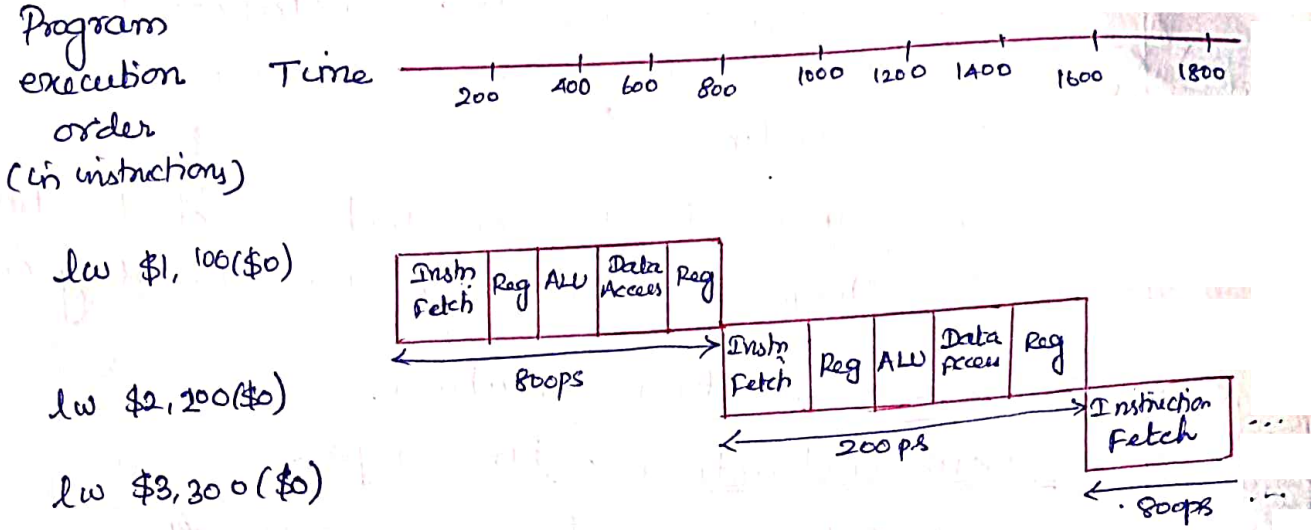
Latency / Pipeline

- The number of stages in a pipeline
  or
  number of stages b/w 2 instrs during execution.

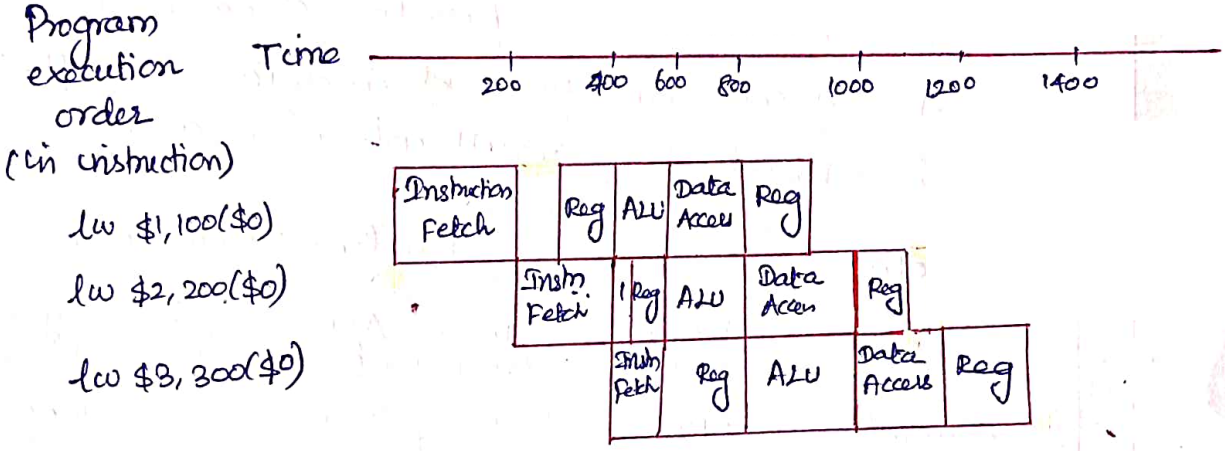
MIPS Instruction

lw $t1, offset($t2)
sw $t1, offset($t2)
add $t1, $t2, $t3

Nonpipelined Execution



Pipelined Execution



* Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction.

Designing Instruction sets for Pipelining

* 1. All MIPS instructions are the same length
2. MIPS has only a few instruction formats.
3. Memory operands only appear in loads or stores in MIPS
4. Operands must be aligned in memory.

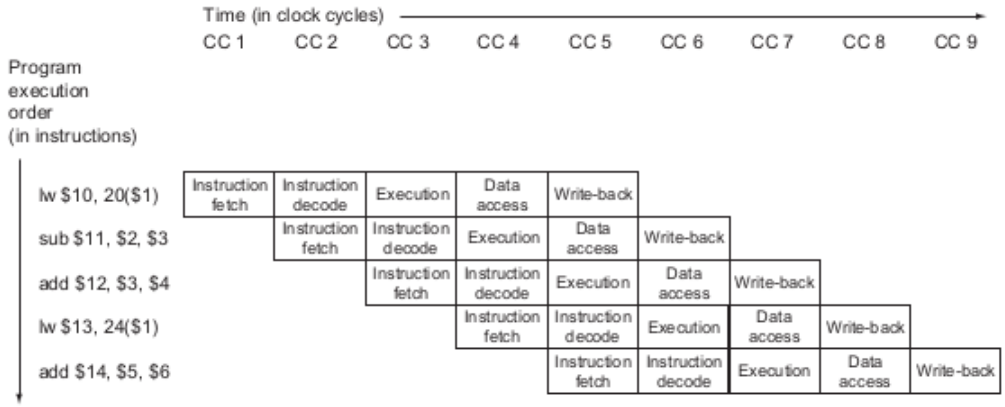


FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.43.

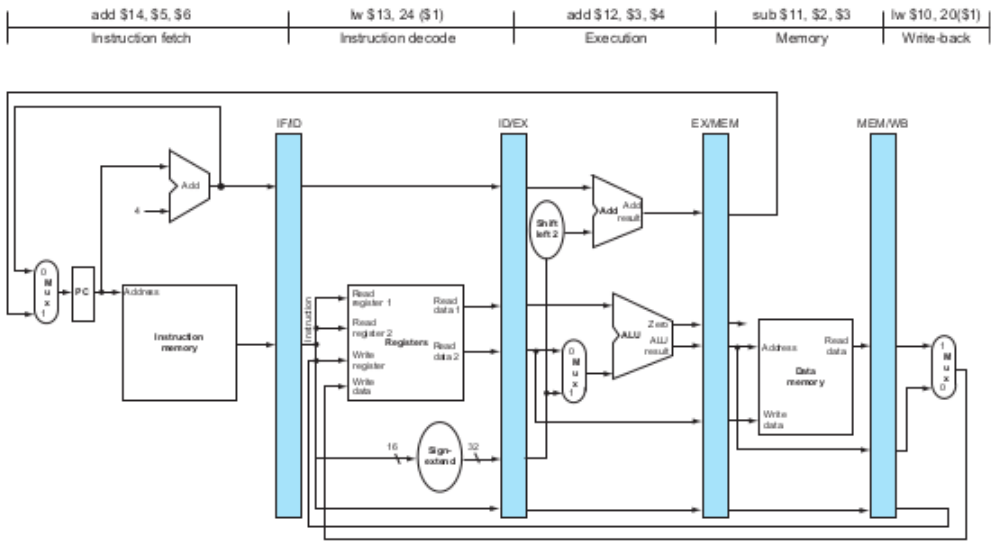


FIGURE 4.45 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.43 and 4.44. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

* Pipeline Hazards:

* Situations in pipelining when the next instruction cannot execute in the following clock cycle.
  - These events are called hazards.

Three types:

1. Structural Hazards
2. Data Hazards
3. Control Hazards

① Structural Hazards:

* 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.

Ex:

* Instruction is accessing data from memory.
* At the same time, another instruction is fetching from the same memory
  - structural hazard occur.

→ Need for two memories.

② Data Hazards:

* When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available

- occur when the pipeline must be stalled because one step must wait for another to complete.

* In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.

Ex:

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

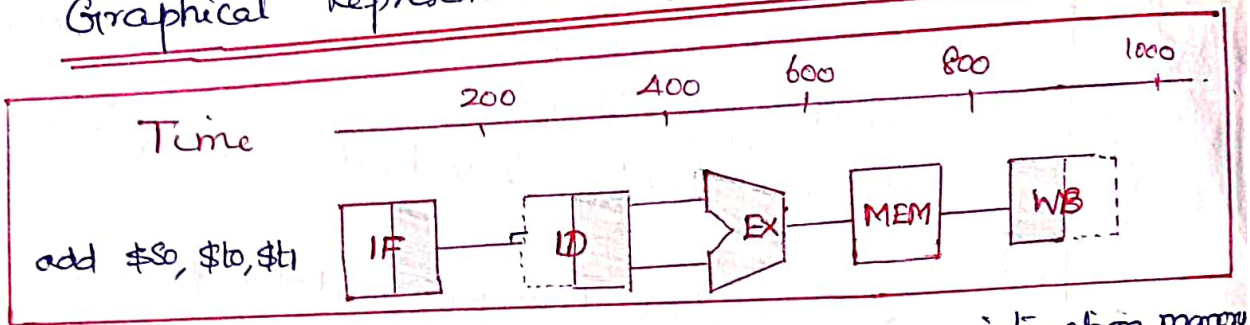
Solution:

① Forwarding/ bypassing:

* Adding extra hardware to retrieve the missing item early from the internal resources.

→ 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.

Graphical Representation of the instruction pipeline



Five stages:

IF – Instruction Fetch – box representing instruction memory

ID – Instruction Decode/ Register file Read – register file being read.

EX – Execution stage – representing ALU

MEM – Memory Access stage – box representing data memory

WB – Write back stage – register file being written

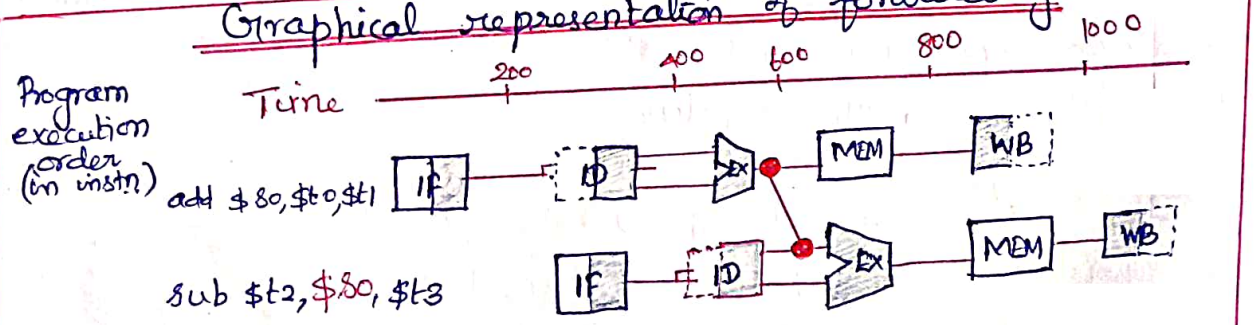
* shading indicates the element is used by the instruction

MEM – white background

Shading on the right half – The element is read.

left half – written in that stage

Graphical representation of forwarding



* Connection to forward the value in $s0 after the execution stage of the add instruction as input to the execution stage of the sub instruction.

→ Forwarding works very well.
- It cannot prevent all pipeline stalls.

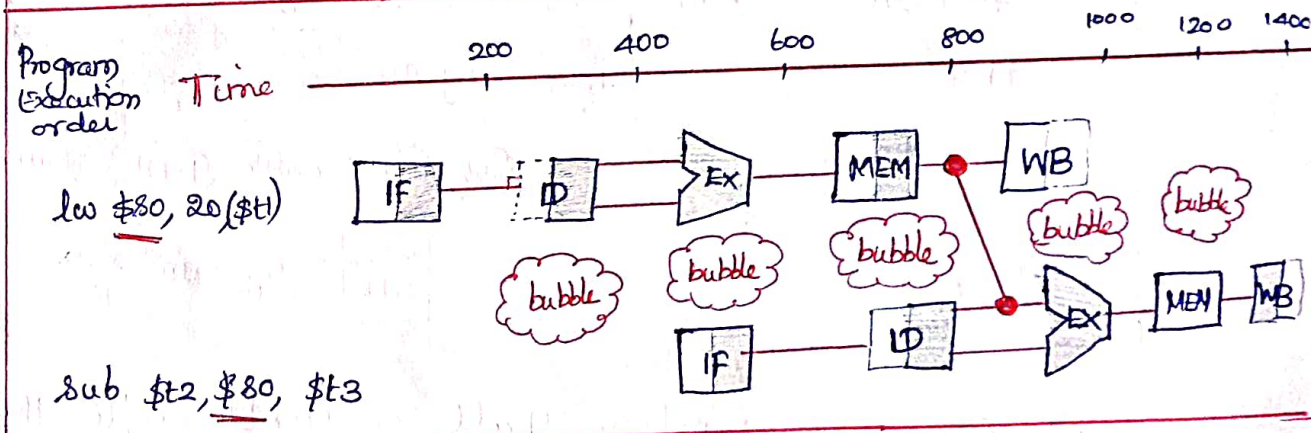
Load-use data hazard:

* Specific form of data hazard
  - data being loaded by a load instruction has not yet become available when it is needed by another instruction.

Ex:
load of $s0 instead of add
→ the desired data would be available only after the fourth stage of the first instruction.
- too late for the input of the 3rd stage of sub.

② Pipeline Stall / Bubble:

* Needs when an R-format instruction following a load tries to use the data.



* A stall initiated in order to resolve a hazard.

→ Reorders code to try to avoid load-use pipeline stalls

Disadvantages of Forwarding:

* Forwarding is harder if there are multiple results to forward per instruction or they need to write a result early on in instruction execution.

③ Reordering the code

Control Hazards: / branch hazard

* Arising from the need to make a decision based on the results of one instruction while others are executing.

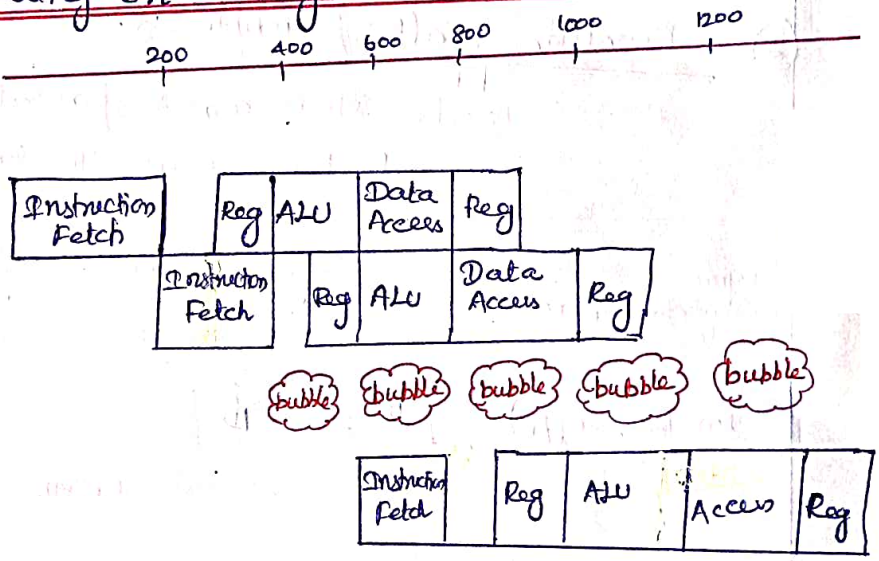
→ When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed;
- the flow of instruction address is not what the pipeline expected.

Soln: Decision task – branch instruction.

* Fetching the instruction following the branch.

Solution:
- To stall immediately after fetching a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

Stalling on every conditional branch



→ One-stage pipeline stall, or bubble, after the branch.

Prediction:

* Use prediction to handle branches.
- to predict always that branches will be untaken
* only when branches are taken does the pipeline stall.

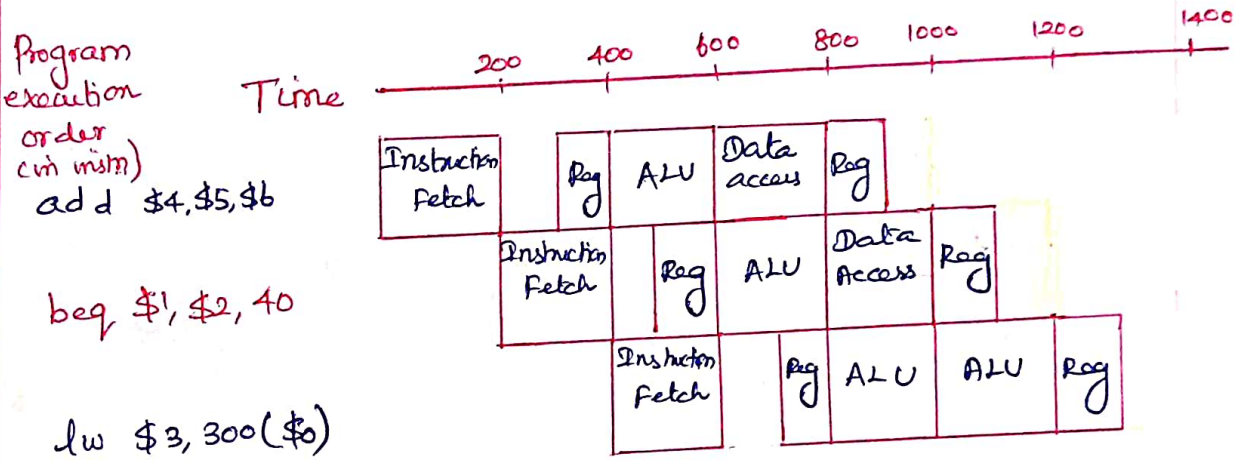
Solution 2:

Branch prediction:

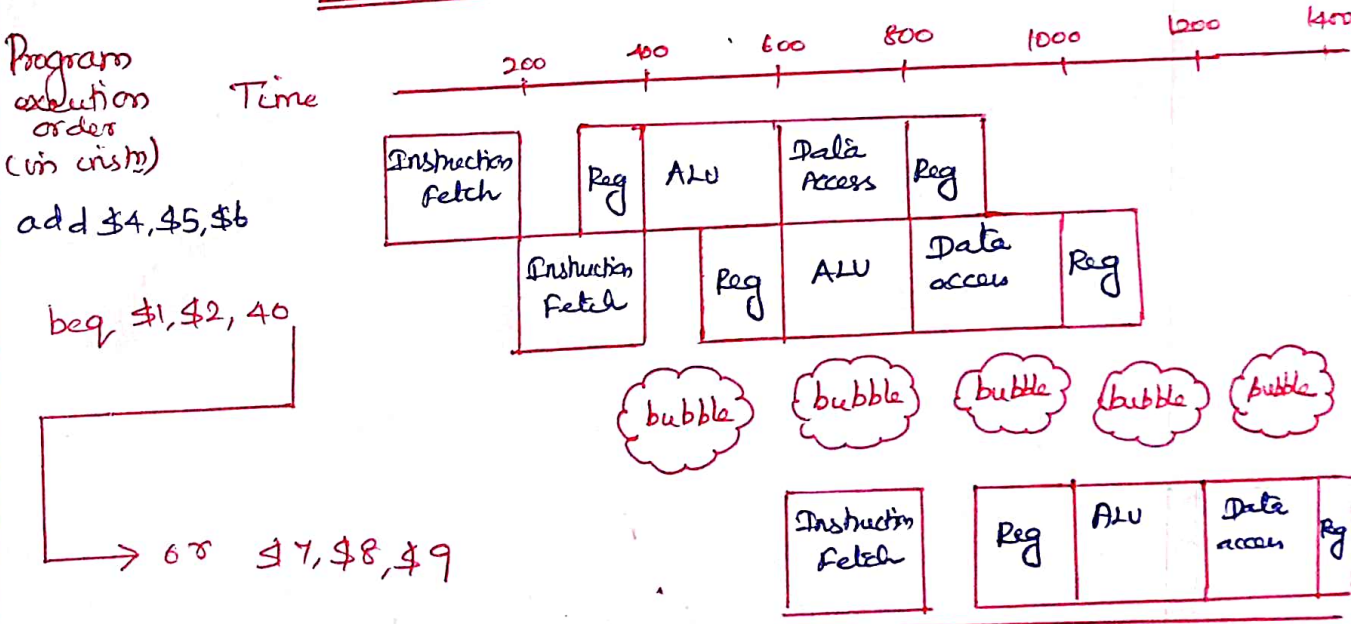
* A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

Ex:

The branch is not taken



The branch is taken



Solution 3:

- Delayed decision / delayed branch.
- used by MIPS architecture.

* The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay.
  - used branches are short
  long → h/w based branch prediction is used

DATA HAZARDS: Forwarding Versus Stalling

Data Hazards:

Also called a pipeline data hazard. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instructions is not yet available.

Data Hazards are obstacles to pipelined execution

```
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. If register $2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following register that refer to register $2.

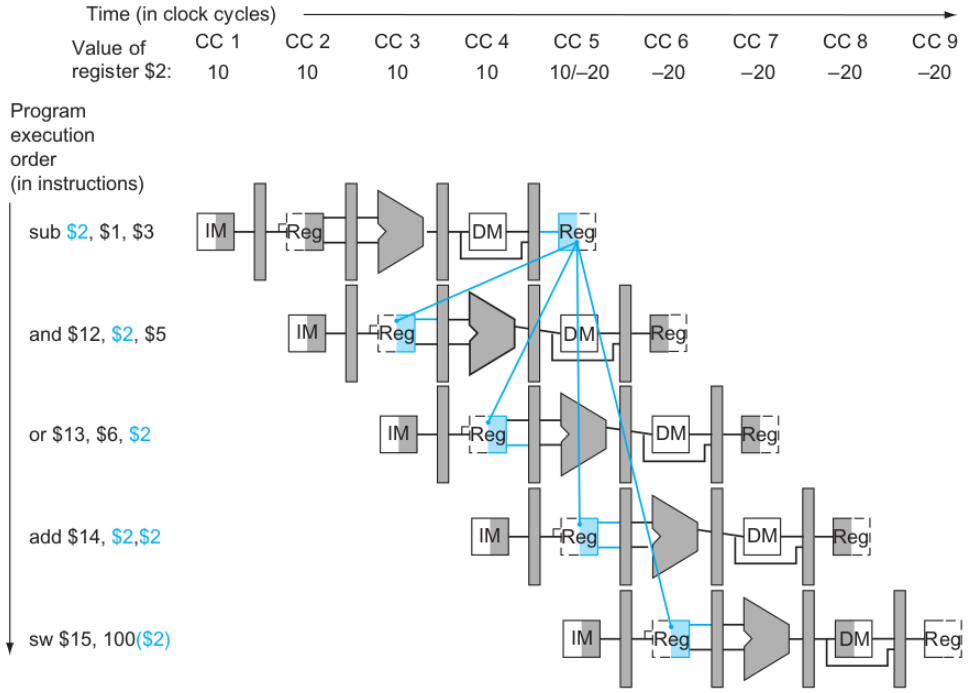


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

Fig 1:- Pipep Pipelined dependencies in a five-instruction sequence using simplified datapaths to show the dependencies.

* The values read for register $2 would not be the result of the sub instruction unless the read occured during clock cycle 5 or later.

* The lines from the top datapath to the lower ones show the dependencies.

* Fig 2 shows the dependencies between the pipeline registers and the inputs to the ALU for the same code sequence as in Fig 1.

* The change is that the dependencies begin from a pipeline register, rather than waiting for the WB stage to write the register file.

* Thus, the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded.

The conditions for detecting hazards and the control signals to resolve them.

1] EX Hazard:-

if (EX/MEM.RegWrite
and (EX/MEM. Register Rd ≠0)
and (EX/MEM. Register Rd = ID/EX. Register Rs)) Forward A=10

if (EX/MEM. RegWrite
and (EX/MEM. Register Rd ≠0)
and (EX/MEM. Register Rd = ID/EX. Register Rt)) Forward B =10

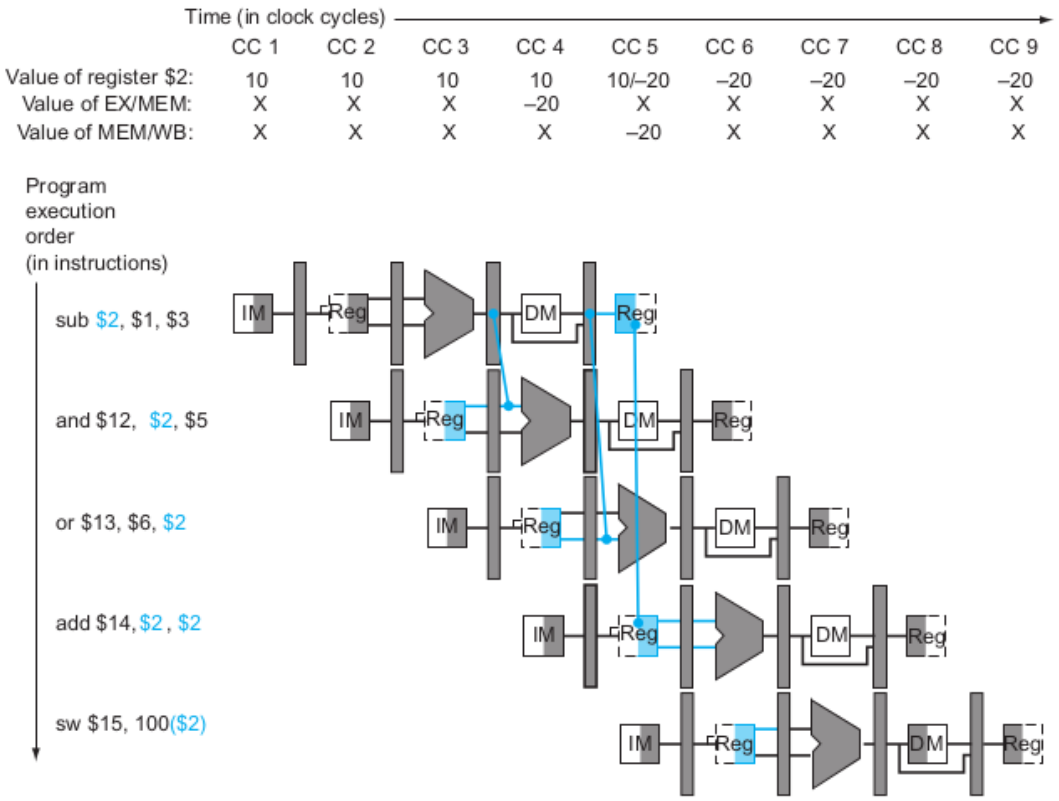


FIGURE 4.53 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction and OR instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register $2 having the value 10 at the beginning and –20 at the end of the clock cycle. As in the rest of this section, we handle all forwarding except for the value to be stored by a store instruction.

2] MEM hazard:

if (MEM/WB. RegWrite
and (MEM/WB. Register Rd ≠ 0)
and (MEM/WB. Register Rd = ID/Ex. Register Rs)) ForwardA = 01

if (MEM/WB. RegWrite
and (MEM/WB. Register Rd ≠ 0)
and (MEM/WB. Register Rd = ID/EX. Register Rt))

ForwardB = 01

Data Hazards and Stalls:-

if (ID/Ex . MemRead and

((ID/Ex . RegisterRt = IF/ID . RegisterRs) or

(ID/Ex . RegisterRt = IF/ID . RegisterRt)))

Stall the pipeline.

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

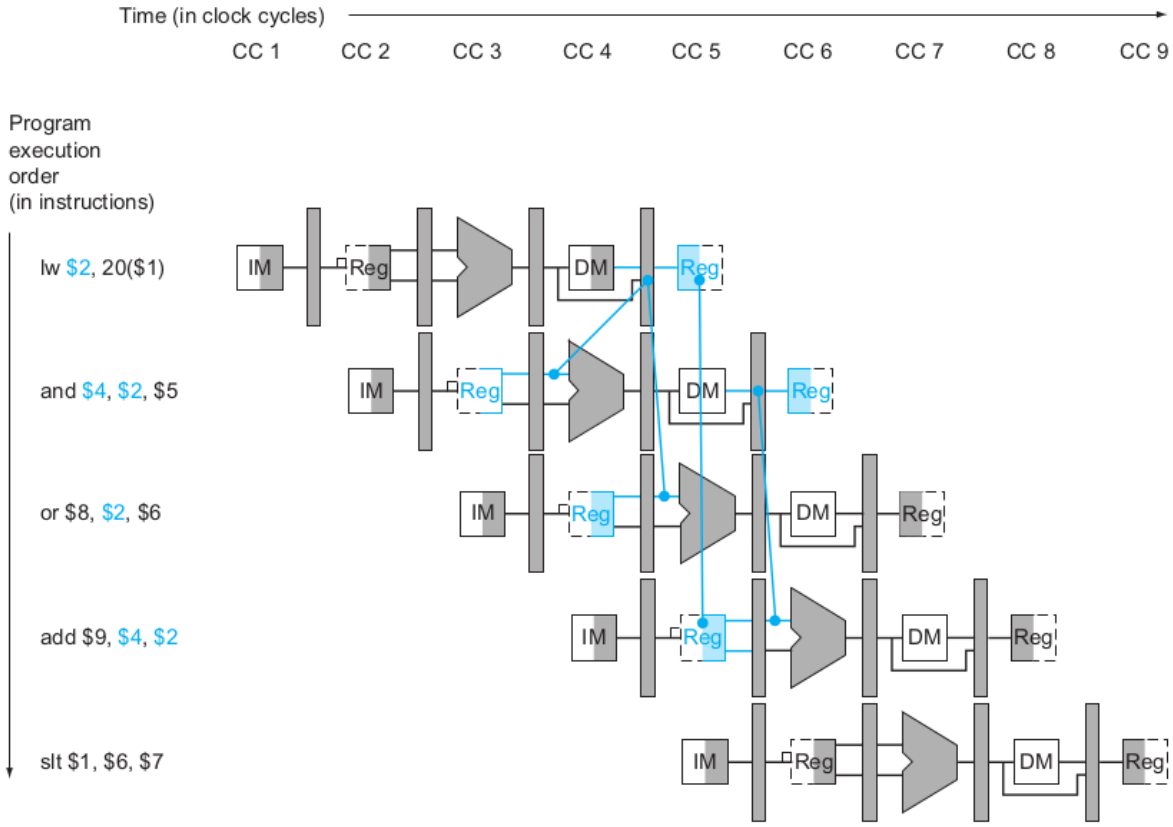


FIGURE 4.58 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Data Hazards and Stalls:

When an instruction tries to read a register following a load instruction that writes the same register.

Nop: An instruction that does no operation to change state.

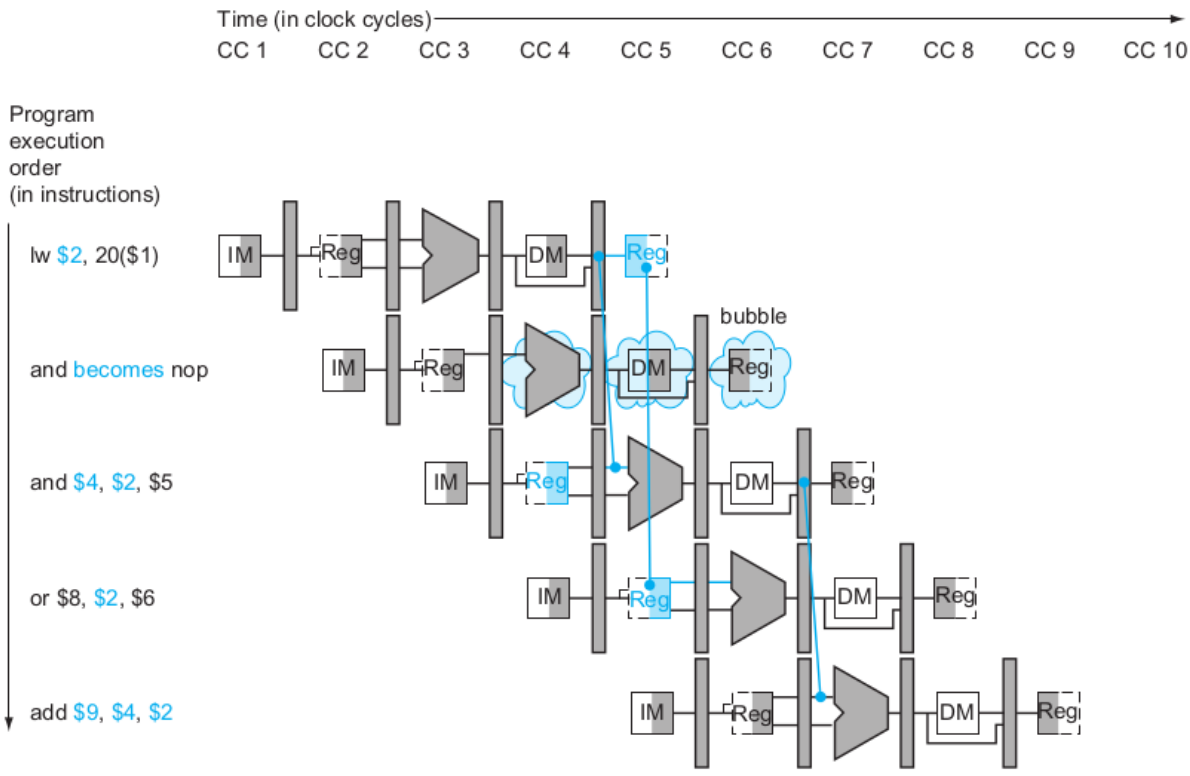


FIGURE 4.59 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the `OR` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

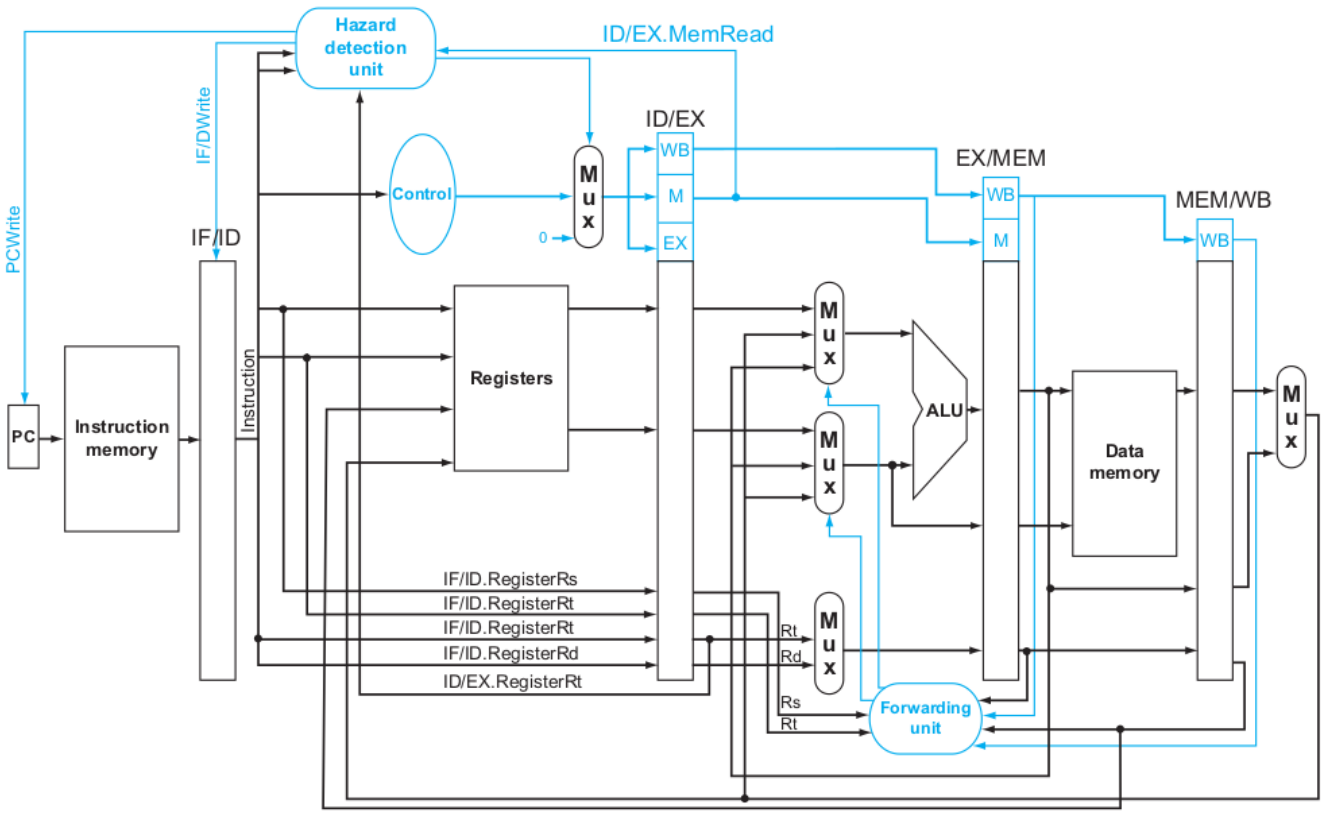


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Control Hazards:-

* Control hazards occurs when we execute the branch instruction in pipeline process.

* Control hazards also called branch hazards when the proper instruction cannot execute in the proper clock cycle

* An instruction must be fetched at every clock cycle to sustain the pipeline.

* The delay in determining the proper instruction to fetch is called control hazard or branch hazard.

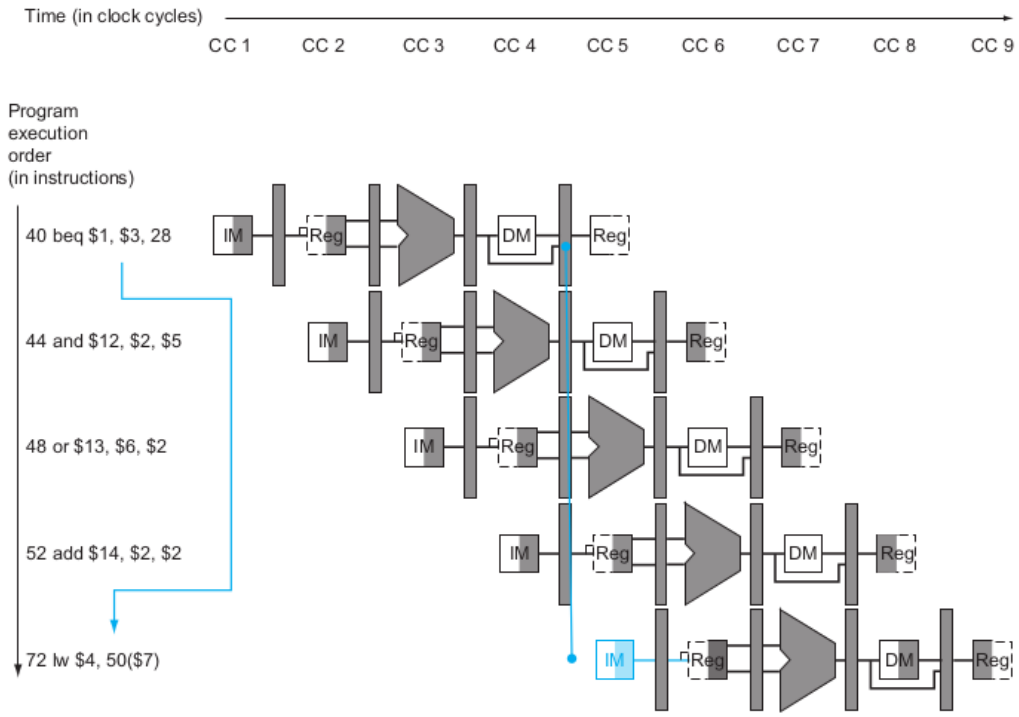


FIGURE 4.61 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, ...) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before `beq` branches to `lw` at location 72. (Figure 4.31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

Methods to resolve control hazards.

The methods to resolve control hazards are

1. * Branch Not Taken
2. * Reducing the delay of branches
3. * Dynamic Branch Prediction.

1. Branch Not Taken:.

* If the branch is taken then the instructions that are being fetched and decoded must be discarded
* After discarding the 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 error.
* If branch not taken means no need to discard any instructions and pipelining will execute the instructions 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.

(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 only but if we move the branch execution earlier in the pipeline then fewer instructions need to be flushed.

* Moving branch execution earlier in the pipeline will increase the speed of performance.

* When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required - a situation that is similar to the use of condition codes for branches.

* Moving the branch decision up requires two actions to occur earlier. They are

→ computing the branch target address

→ Evaluating the branch decision.

* The easy part of the change is to move up the branch address calculation.

* Now, just move the branch adder from the EX stage to ID stage.

* The harder part is the branch decision itself. For branch equal, we would compare two registers read during ID stage to see if they are equal.

* Equality is tested by using exclusive ORing.

* For example, to implement branch on equal, we will need to forward results to the equality test logic that operates during ID.

There are two complicating factors.

1) During ID, we must decode the instruction, decide whether bypass to the equality unit is needed.

Forwarding the operands of branches are handled by ALU forwarding logic unit.

Note that 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 stage but may be produced later in time, it is possible to occur data hazard and stall will be needed.

To overcome these difficulties we can move the branch EX to the ID stage, because it reduces the penalty of a branch to only one instruction if the branch is taken.

(3) Dynamic Branch Prediction.

* Dynamic branch prediction is the prediction of branches at runtime using runtime information.

* One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the

same place as the last time. This technique is called dynamic branch prediction.

* One implementation of that approach is a branch prediction buffer or branch history table

* A branch prediction buffer is a small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not

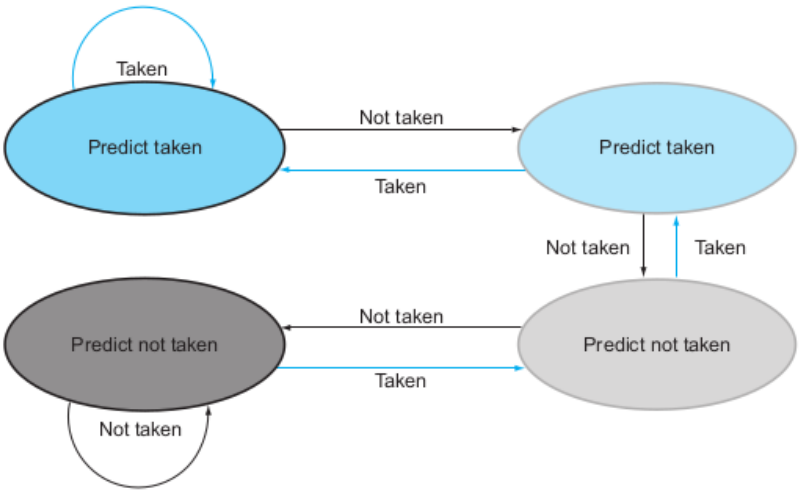


FIGURE 4.63 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

The final datapath and control