UNIT III
PROCESSOR AND CONTROL UNIT

Basic MIPS implementation – Building a Datapath – Control Implementation Scheme – Pipelining – Pipelined datapath and control – Handling Data Hazards & Control Hazards Exceptions

3.1 Basic MIPS Implementation

In this chapter we will see the implementation of a subset of the core MIPS instruction set. These instructions are divided into three classes :

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

The subset considered here does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions. The key principles used in creating a datapath and designing the control are discussed here. The implementation of the remaining instructions is somewhat similar.

For implementing every instruction, the first two steps are same:

1. Fetch the instruction: Send the Program Counter (PC) contents (address of instruction) to the memory that contains the opcode and fetch the instruction from that memory.

2. Fetch operand(s) : 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 we require to read two registers.

The remaining actions required to complete the instruction depend on the instruction class. For each of the three instruction classes (memory-reference, arithmetic-logical and branches), the actions are mostly the same, independent of the exact instruction. This shows that the simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes, except jump, use the Arithmetic-Logical Unit (ALU) after reading the registers.

- Memory-reference instructions use the ALU for an address calculation
- Arithmetic-logical instructions use the ALU for the operation execution and
- Branches use the ALU for comparison.
- After using the ALU, the actions required to complete various instruction classes are not same.

A memory-reference instruction needs 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.

A branch instruction 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.

Fig. 3. 1 shows the block diagram of a MIPS implementation, showing the functional units and interconnection between them.

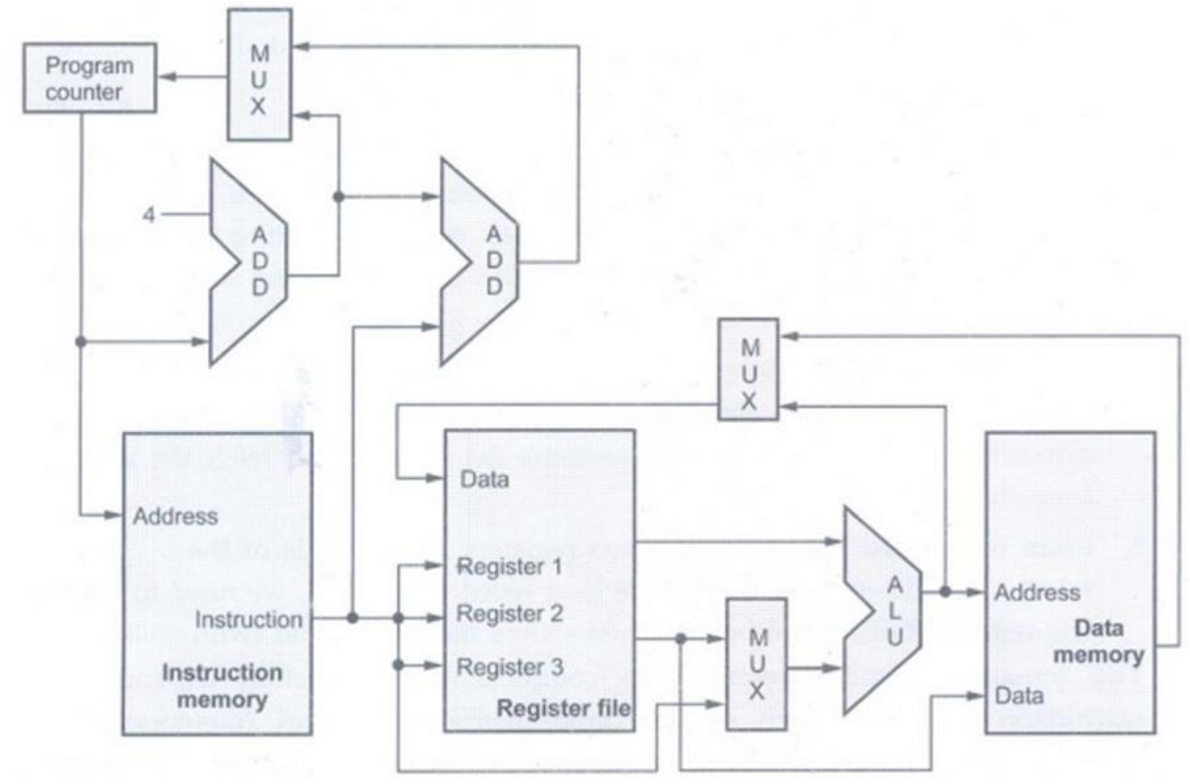


Fig 3.1 Major functional units and interconnections between them for implementation of MIPS subset

3.1.1 Operation

The program counter gives the instruction address to the instruction memory.

After the instruction is fetched, the register operands required by an instruction are specified by fields of that instruction.

Once the register operands have been fetched, they can be used 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. 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, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4.

Fig. 3.1 shows that data going to a particular unit is coming from two different sources. For example, the value written into the PC can come from one of two adders, the data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate field of the instruction. The selection of appropriate source is done using multiplexer (data selector). The multiplexer selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed. This is illustrated in Fig. 3.2.

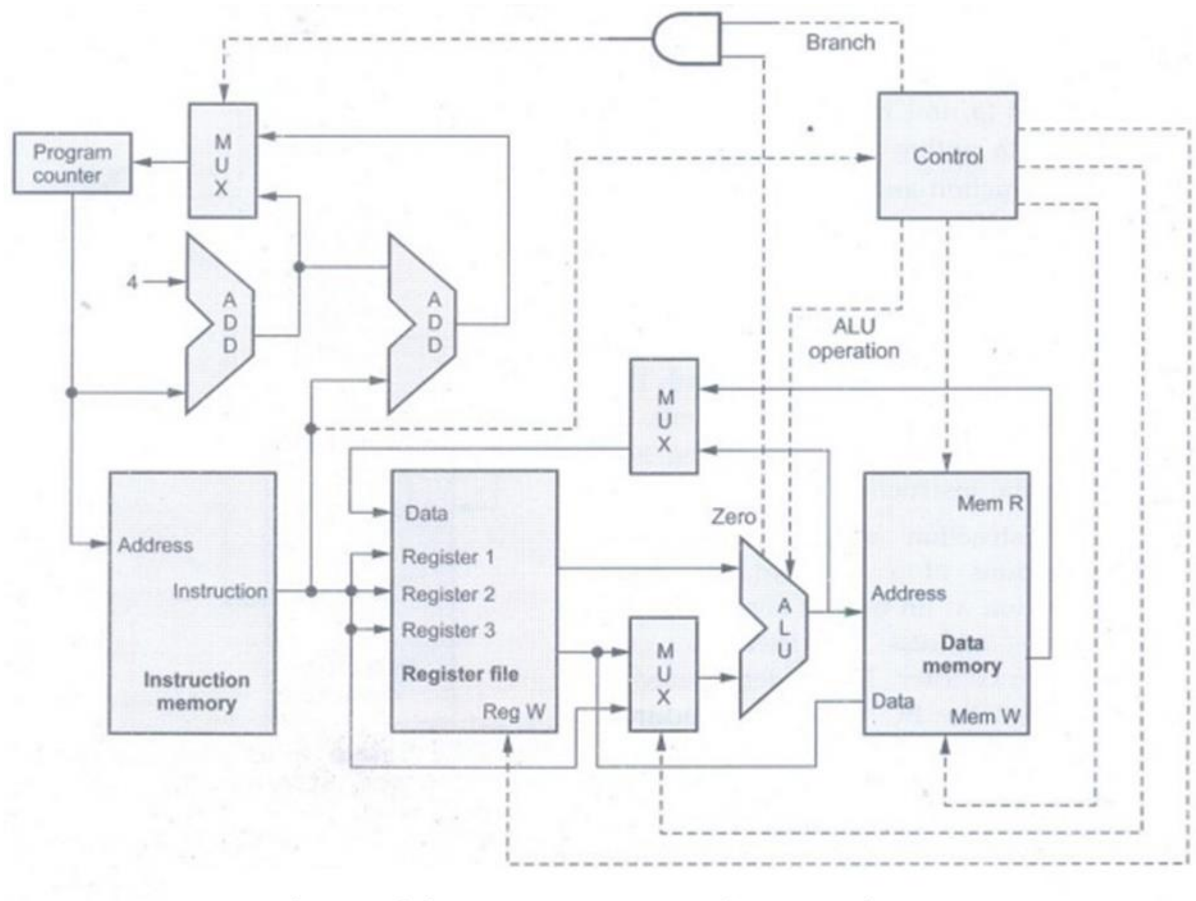


Fig 3.2 Basic implementation of MIPS with Control signals

Fig. 3.2 also shows the control unit, which has the instruction as an input, is used to determine the control signals for the functional units and two of the multiplexers.

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.

3.2. Building a Data Path

As shown in Fig. 3.4, the MIPS implementation includes, the datapath elements (a unit used to operate on or hold data within a processor) such as the instruction and data memories, the register file, the ALU, and adders.

Fig. 3.3 shows the combination of the three elements (instruction memory, program counter and adder) from Fig. 3.4 to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

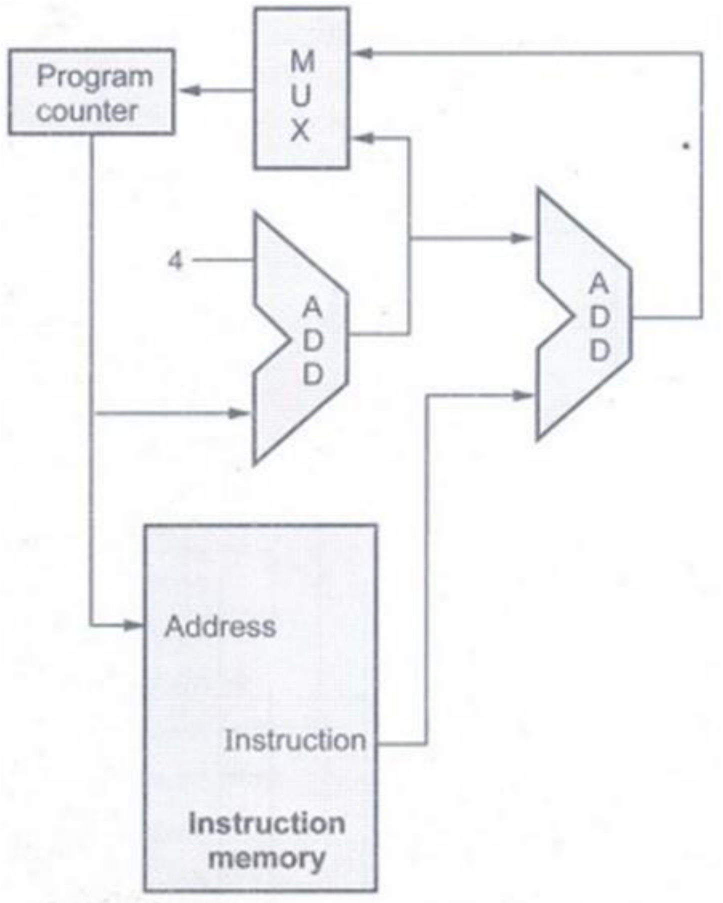


Fig 3.3 Data path to fetch instruction and increment PC

The instruction memory stores the instructions of a program and gives instruction as an output corresponding to the address specified by the program counter. The adder is used to increment the PC by 4 to the address of the next instruction.

Since the instruction memory only reads, the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed.

The program counter is a 32-bits register that is written at the end of every clock cycle and thus does not need a write control signal.

The adder always adds its two 32-bits inputs and place the sum on its output.

3.2.1. Datapath Segment for Arithmetic - Logic Instructions

The arithmetic-logic instructions read operands from two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions as R-type instructions. This instruction class includes add, sub, AND, OR, and slt. For example, OR $t1, $t2, $t3 reads $t2 and $t3, performs logical OR operation and saves the result in $t1.

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

Fig. 3.4 shows multiport register file (two read ports and one write port) and the ALU section of Fig. 3.4 We know that, the R-format instructions have three register operands: numbers Two source operands and one destination operand.

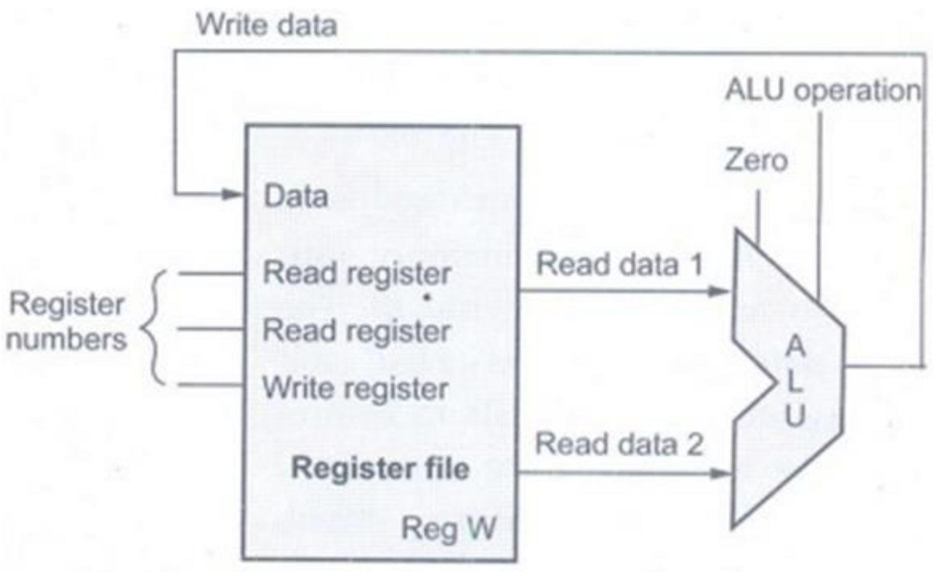


Fig 3.4 Multiport register file and the ALU

For each data word to be read from the register file, we need to specify the register number to the register file. On the other hand, to write a data word, we need two inputs: One to specify the register number to be written and one to supply the data to be written into the register.

The register file always outputs the contents of whatever register numbers are on the Read register inputs. Write operations, however, are controlled by the write control (Reg W) signal. This signal is asserted for a write operation at the clock edge.

Since writes to the register file are edge-triggered, it is possible to perform read and write operation for the same register within a clock cycle: The read operation gives the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle.

As shown in Fig. 3.4, the register number inputs are 5 bits wide to specify one of 32 registers, whereas the data input and two data output buses are each 32 bits wide.

3.2.2. Datapath Segment for Load Word and Store Word Instructions

Now, consider the MIPS load word and store word instructions, which have the general form lw $t1, offset_value($t2) or sw $t1, offset_value ($t2).

In these instructions $t1 is a data register and $t2 is a base register. The memory address is computed by adding the base register ($t2), to the 16-bits signed offset value specified in the instruction.

In case of store instruction, the value from the data register ($t1) must be read and in case of load instruction, the value read from memory must be written into the data register ($t1). Thus, we will need both the register file and the ALU from Fig. 3.4.

We know that, the offset value is 16-bits and base register contents are 32-bits. Thus, we need a sign-extend unit to convert the 16-bits offset field in the instruction to a 32-bits signed value so that it can be added to base register.

In addition to sign extend unit, we need a data memory unit to read from or write to. The data memory has read and write control signals to control the read and write operations. It also has an address input, and an input for the data to be written into memory. Fig. 3.5 shows these two elements.

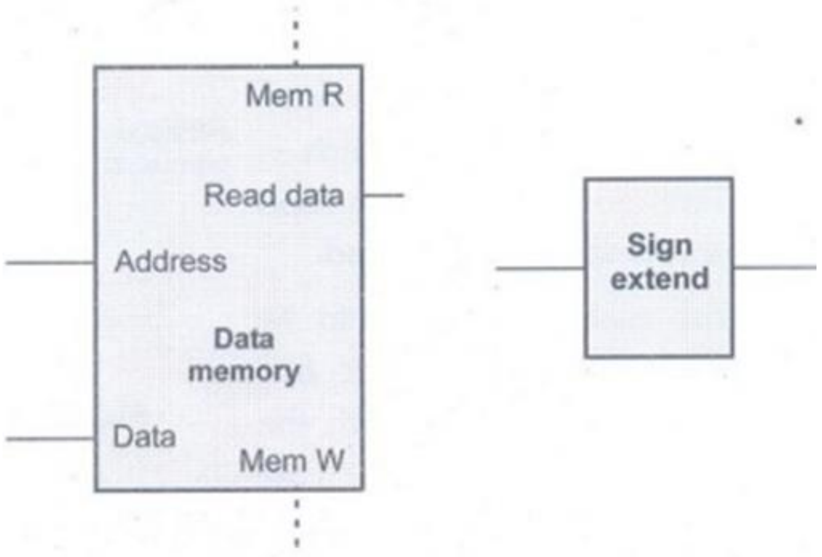


Fig 3.5 Data memory unit and the sign extension unit

Sign extension is implemented by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

Therefore, two units needed to implement loads and stores, in addition to the register file and ALU of Fig. 3.4, are the data memory unit and the sign extension unit.

3.2.3. Datapath Segment for Branch Instruction

The beq instruction has three operands, two registers that are compared for equality, and a 16-bits offset which is used to compute the branch target address relative to the branch instruction address. It has a general form beq $t1, $t2, offset.

To implement this instruction, it is necessary to compute the branch target address by adding the sign-extended offset field of the instruction to the PC. The two important things in the definition of branch instructions which need careful attention are:

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

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

Therefore, the branch target address is given by

Branch target address = PC+4 + offset (shifted left 2 bits)

In addition to computing the branch target address, we must also see whether the two operands are equal or not. If two operands are not equal the next instruction is the instruction that follows sequentially (PC= PC+4); in this case, we say that the branch is not taken. On the other hand, if two operands are equal (i.e., condition is true), the branch target address becomes the new PC, and we say that the branch is taken.

Thus, the branch datapath must perform two operations : Compute the branch target address and compare the registercontents.

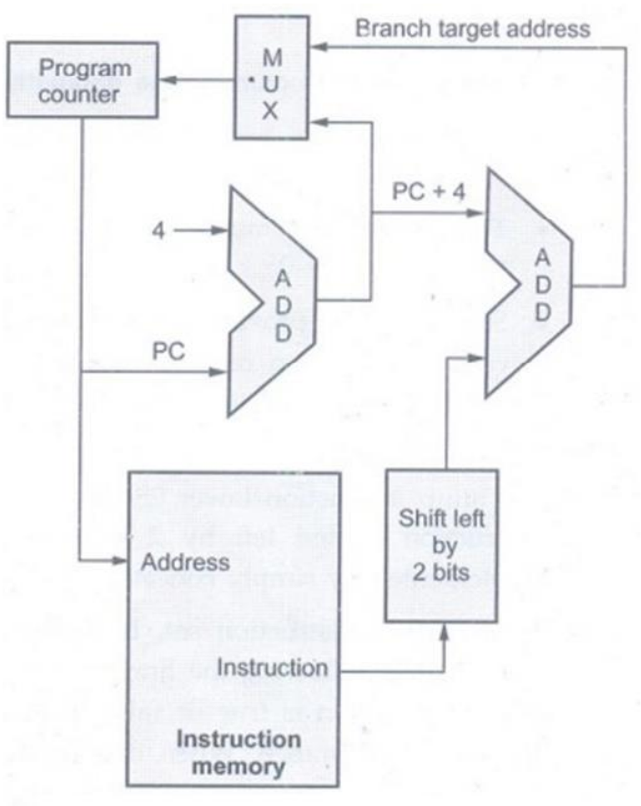


Fig 3.6 Computation of branch target address

Fig. 3.7 shows the structure of the datapath segment that handles branches.

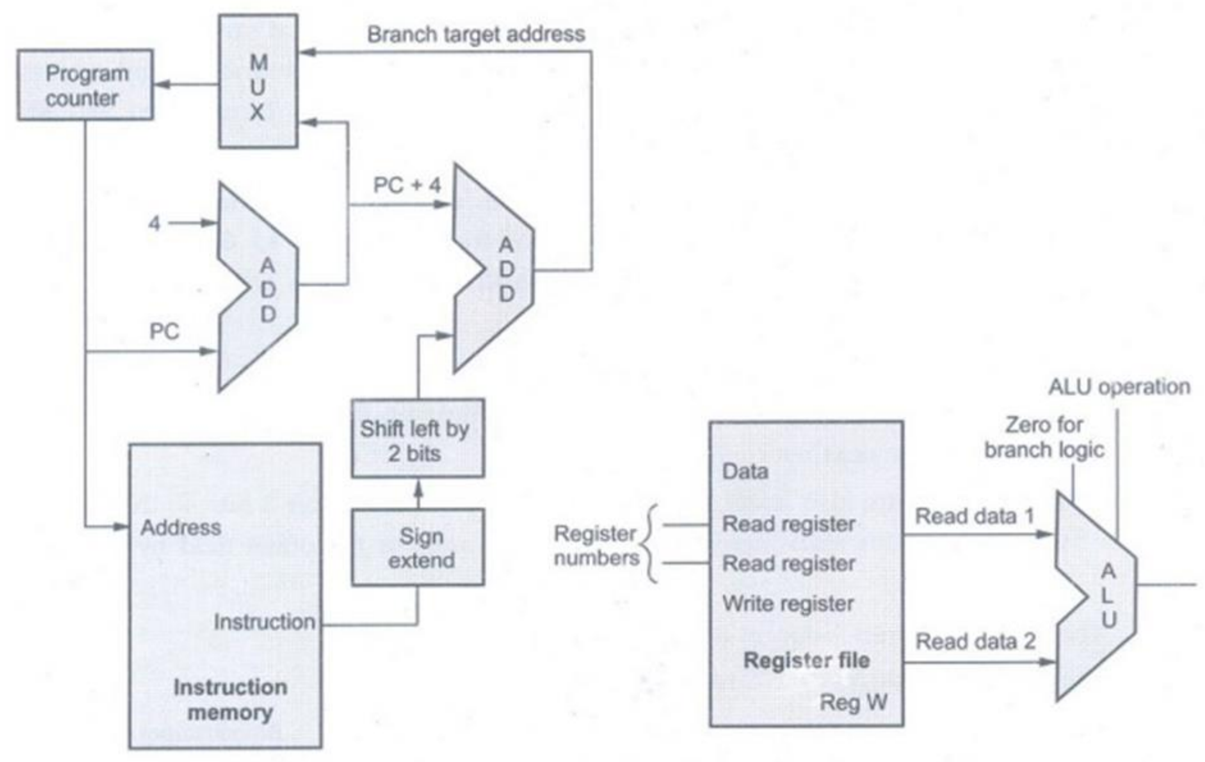


Fig 3.7 Structure of the datapath segment that handles branches

To compute the branch target address, the branch datapath includes a sign extension unit, shifter and an adder.

To perform the compare, we need to use the register file and the ALU shown in Fig. 3.4.

Since the ALU provides an Zero signal that indicates whether the result is 0, we I can send the two register operands to the ALU with the control set to do a subtract operation. If the Zero signal is asserted, we know that the two values are equal.

For jump instruction lower 28 bits of the PC are replaced by lower 26 bits of the instruction shifted left by 2 bits and making two LSB bits 0. This can be implemented by simply concatenating 00 to the jump.

In the MIPS instruction set, branches are delayed, meaning that the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false. When the condition is false, the execution looks like a normal branch. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address.

3.2.4. Creating a Single Datapath

We can combine the datapath components needed for the individual instruction classes into a single datapath and add the control to complete the implementation.

This simplest datapath will attempt to 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. We therefore need a memory for instructions separate from one for data. We need the functional units to be duplicated and many of the elements can be shared by different instruction flows.

To share a datapath element between two different instruction classes, we have connected multiple connections to the input of an element and used a multiplexer and control signal to select among the multiple inputs.

Example 3.1 Show how to build a datapath for the operational portion of the memory-reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexers.

Solution : To create a datapath 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. Fig. 3.8 shows the operational portion of the combined datapath.

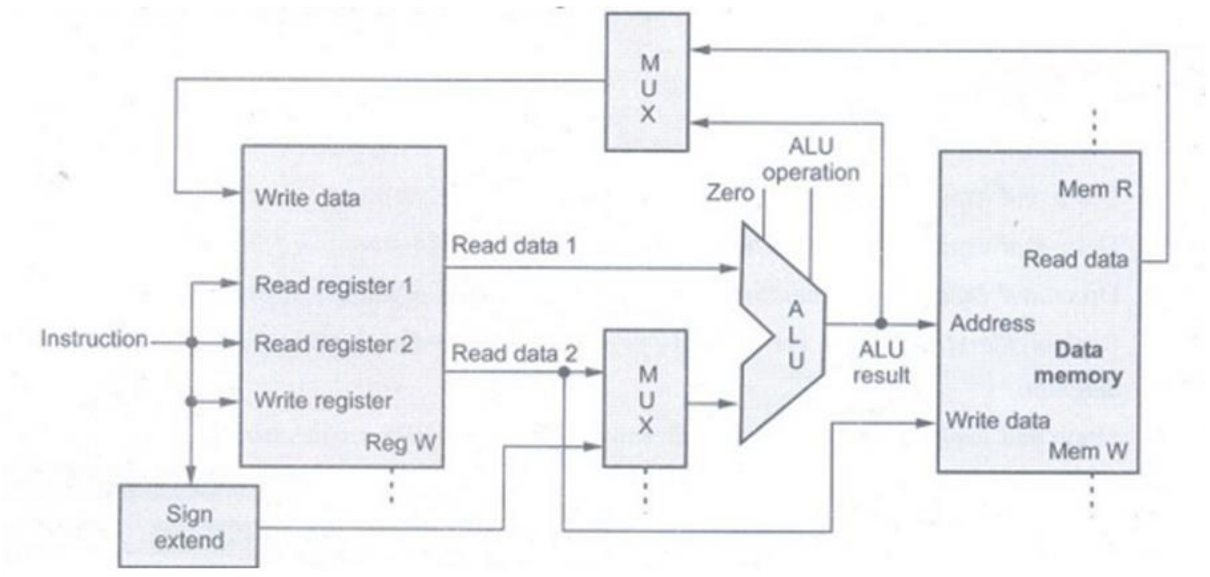


Fig 3.8 Combine datapath for the memory instruction and the R-type instruction

We can make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch, the datapath from R-type and memory instructions, and the datapath for branches as shown in the Fig. 3.9.

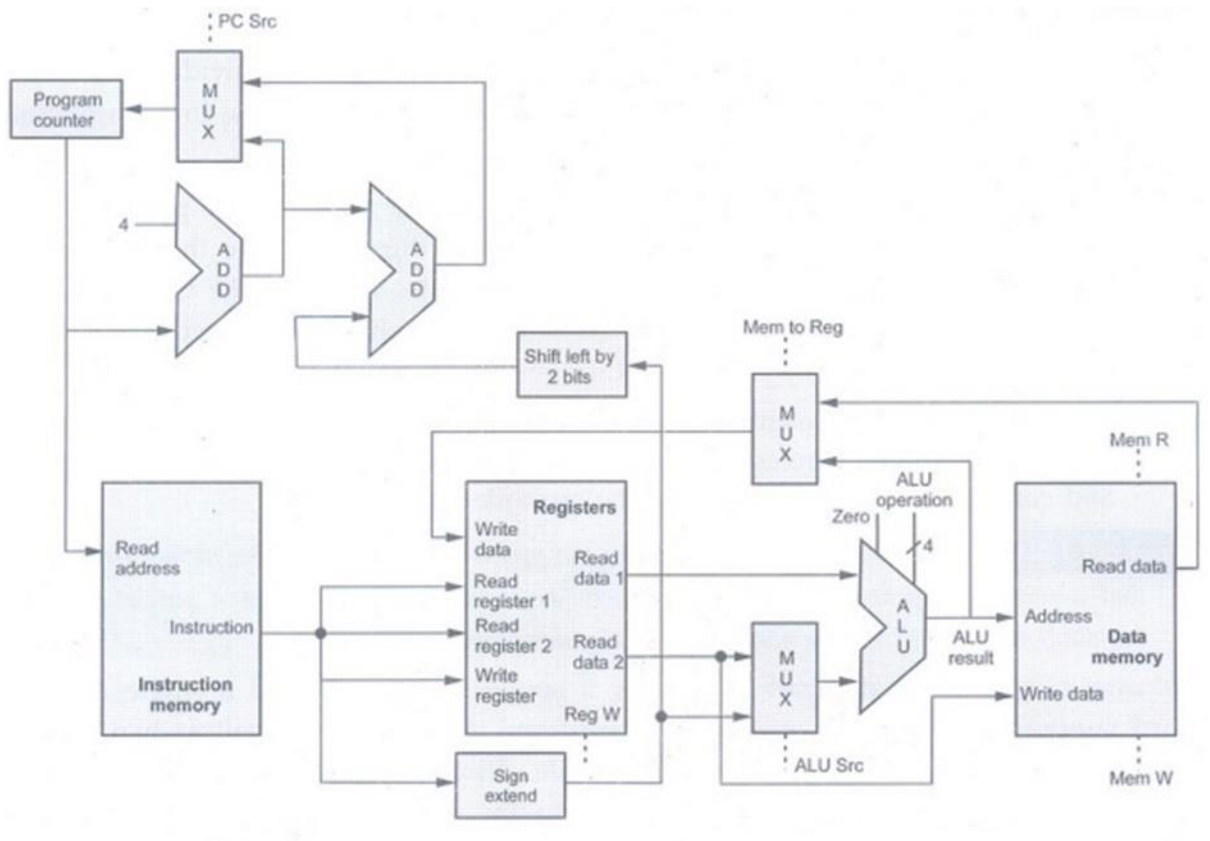


Fig 3.9 The simple combine datapath for the MIPS architecture

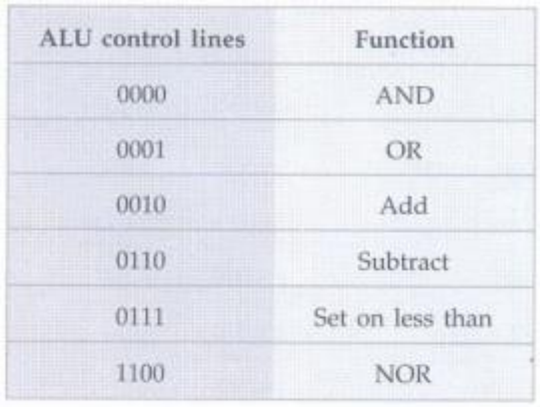
3.3. Designing a Control Unit

Here, we restrict ourselves to implement load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than. We will also the design to include a jump instruction (j).

3.3.1. The ALU Control

The MIPS ALU defines the six following combinations of four control inputs shown in the Table 3.1:

Table 3.1 Combination of Four control inputs



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

Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR function is needed for other parts of the MIPS instruction set. It is not included in the subset we are implementing.)

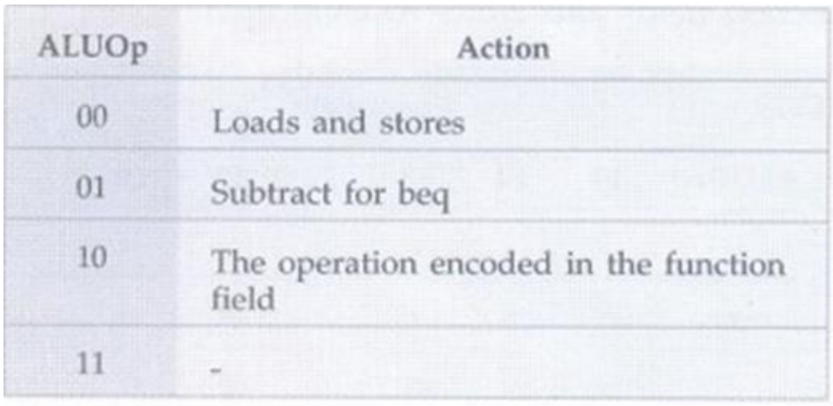
In case of load word and store word instructions, we use the ALU to compute the memory address by addition.

In case of the R-type instructions, the ALU needs to perform one of the five actions - AND, OR, subtract, add, or set on less than.

In case of branch equal, the ALU must perform a subtraction.

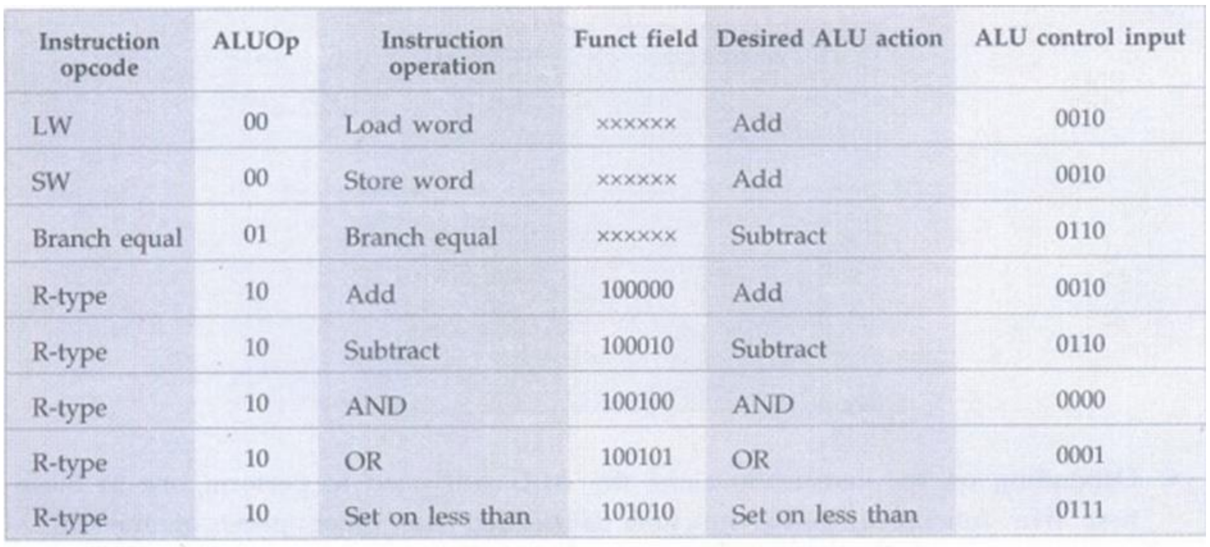
We can control the operation of ALU by the 4-bits ALU control input and 2-bits ALUOP. The 2-bits ALUOP is interpreted as shown in Table 3.2.

Table 3.2 ALUOp field



ALUOp	Action
00	Loads and stores
01	Subtract for beq
10	The operation encoded in the function field
11	-

Table 3.3 shows how to set the ALU control inputs based on the 2-bits ALUOP control and the 6-bits function code.



Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	Load word	xxxxxx	Add	0010
SW	00	Store word	xxxxxx	Add	0010
Branch equal	01	Branch equal	xxxxxx	Subtract	0110
R-type	10	Add	100000	Add	0010
R-type	10	Subtract	100010	Subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	Set on less than	101010	Set on less than	0111

Here, multiple levels of decoding technique is used.

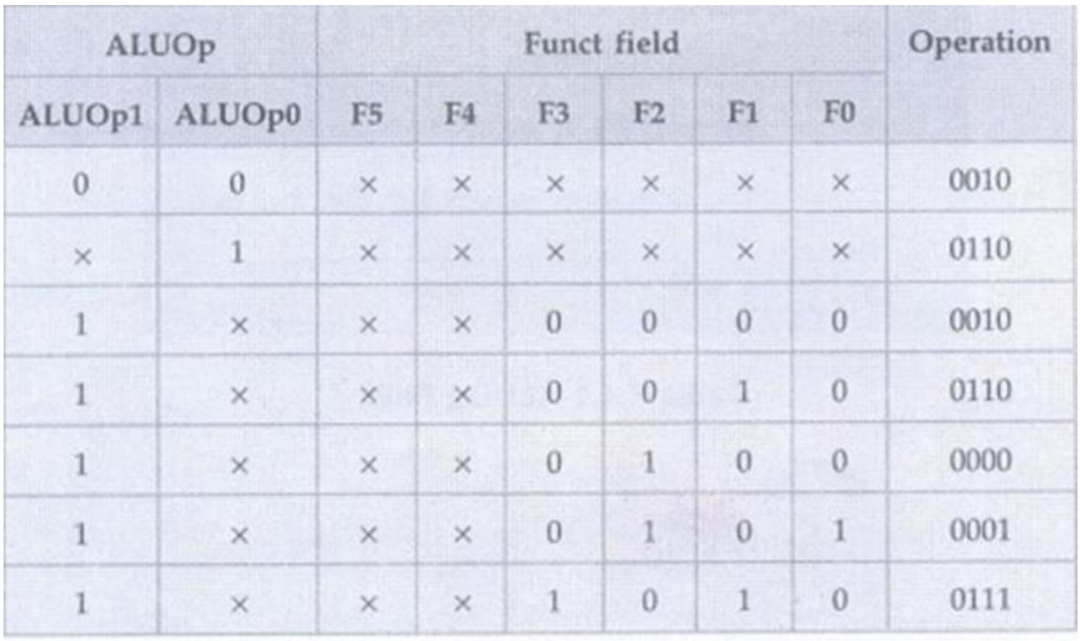
3.3.2. Advantages of using multiple levels of decoding

It reduces the size of the main control unit.

Use of several smaller control units may also potentially increase the speed of the control unit.

Table 3.4 shows how the 4-bits ALU control is set depending on these two input fields: 6-bits function fields and 2-bits ALUOp field.

Table 3.4



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

Once the truth table has been constructed, it can be optimized and can be implemented using logic gates.

3.4. Designing the Main Control Unit

Before looking at the rest of the control design, it is useful to review the formats

of the three instruction classes: The R-type, branch and load-store instructions. Fig. 3.10 shows these formats.

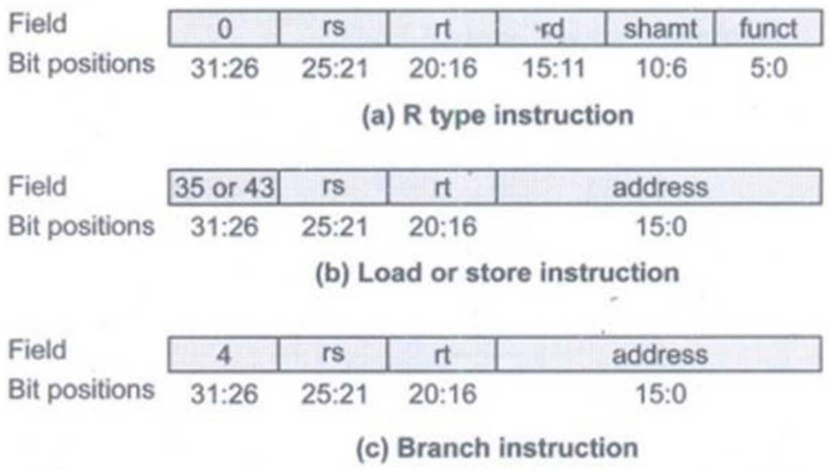


Fig 3.10 Formats of the three instruction classes

Format for R-format instructions: Opcode is 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The funct (Function) field is an ALU function discussed in the previous section. The shamt field is used only for shifts.

Format for load and store instructions: Load (opcode) or store (opcode). The register rs is 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.

Format for branch equal: Opcode is 4. The registers rs and rt are the source registers that are compared for equality. The 16-bits address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address.

3.4.1. Important observations about this instruction format

Bits 31: 26 in the instruction format is op field and gives opcode (operation code). We will refer to this field as Op[5: 0].

Bits 25:21 and 20:16 in the instruction format always specify the rs and rt fields, respectively.

Bits 25: 21 always give the base register (rs) for load and store instructions.

Bits 15: 0 give the 16-bits offset for branch equal, load, and store.

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 multiplexer to select which field of the instruction is used.

From the above information, we can add the instruction labels and extra multiplexer (for the Write register number input of the register file) to the simple datapath. Fig. 3.11 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. Since all the multiplexers have two inputs, they each require a single control line.

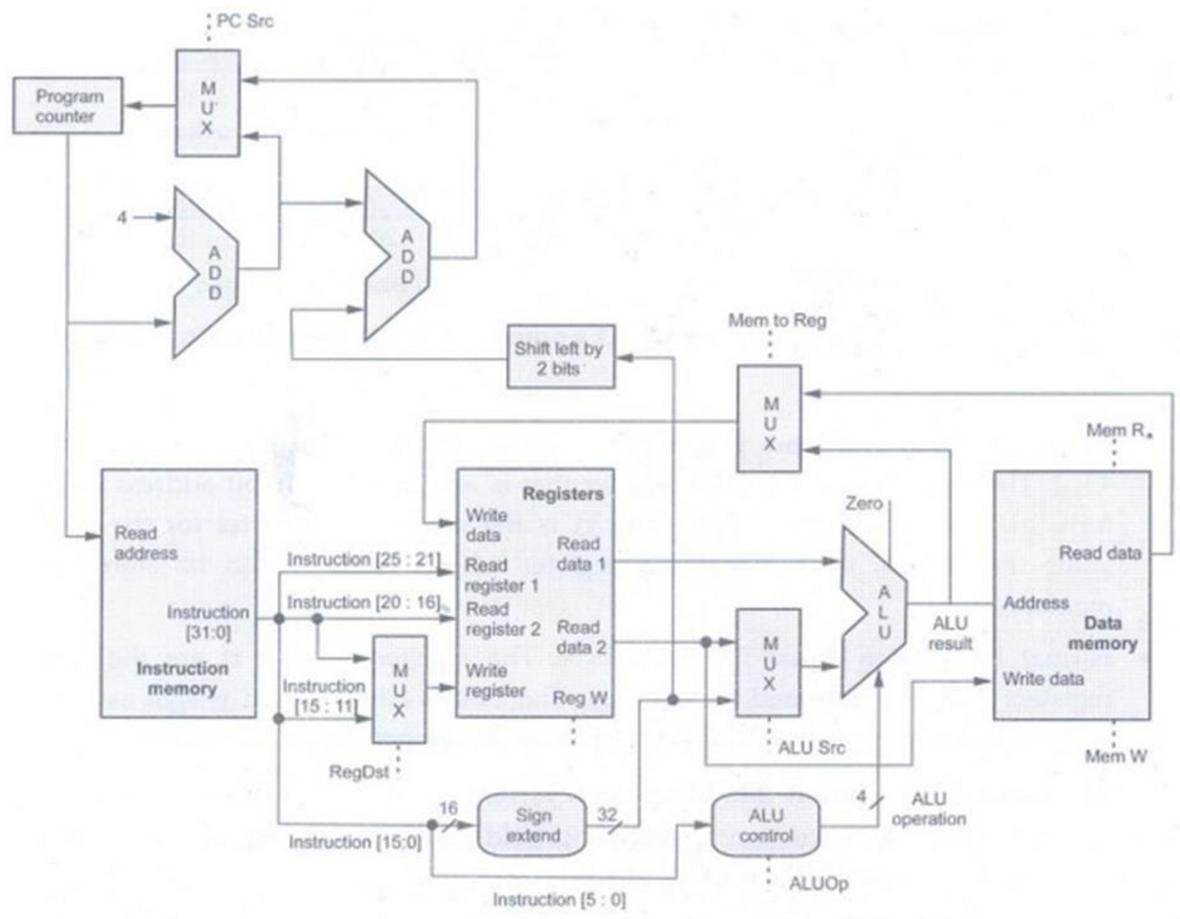
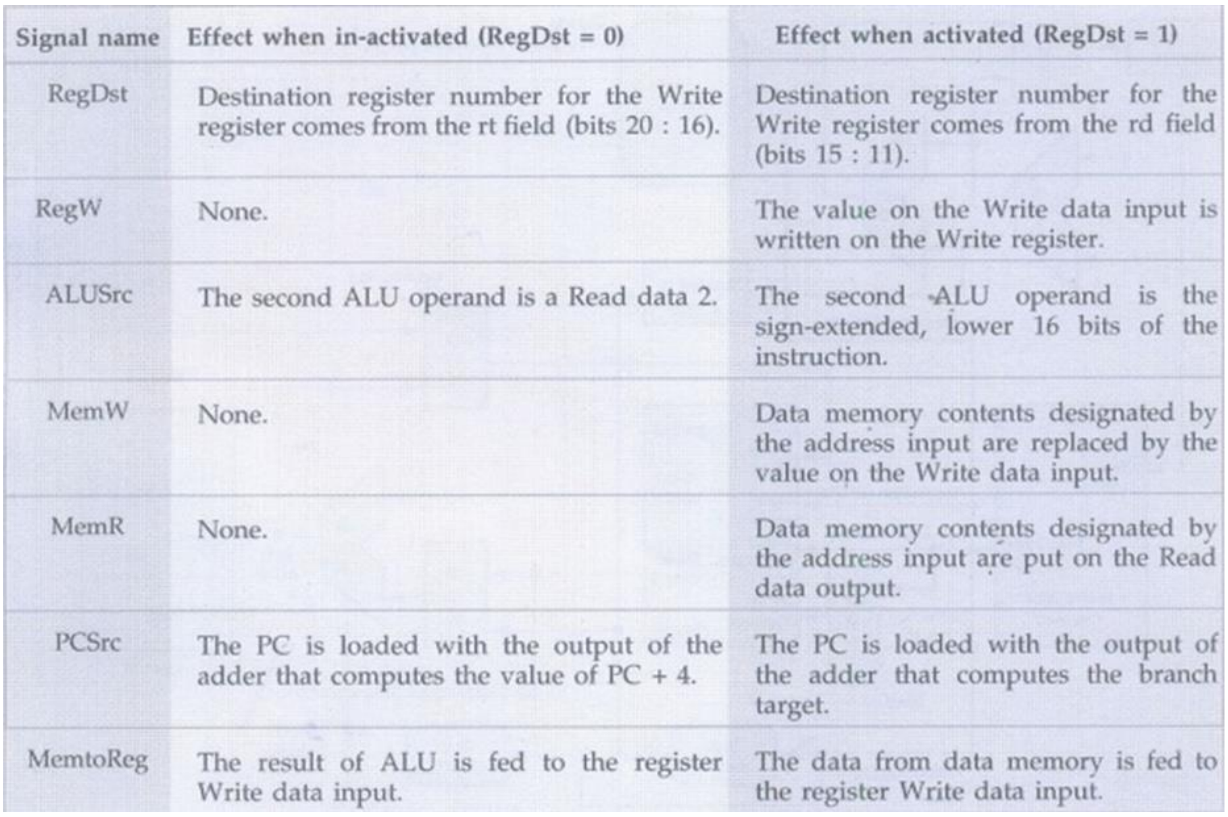


Fig 3.11 The datapath with all necessary multiplexers and all control lines

Fig. 3.11 shows seven single-bit control lines (RegDst, RegW, ALUSrc, MemW, MemR, PCSrc and MemtoReg) plus the 2-bits ALUOP control signal.

Table 3.5. describes the function of single-bit control lines.

Table 3.5 Functions if seven single-bit control lines



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

These nine control signals (seven single-bit control lines and the 2-bits ALUOP control signals) can be set according six input signals to the control unit, which are the opcode bits 31 to 26. Fig. 3.12 shows the datapath with the control unit and the control signals. [Refer Fig. 3.12 on next page]

As shown in the Fig. 3.12, the input to the control unit is the 6-bits 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-bits 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.

Table 3.6 defines whether each control signal should be 0, 1, or don't care (X) for each of the opcode values.

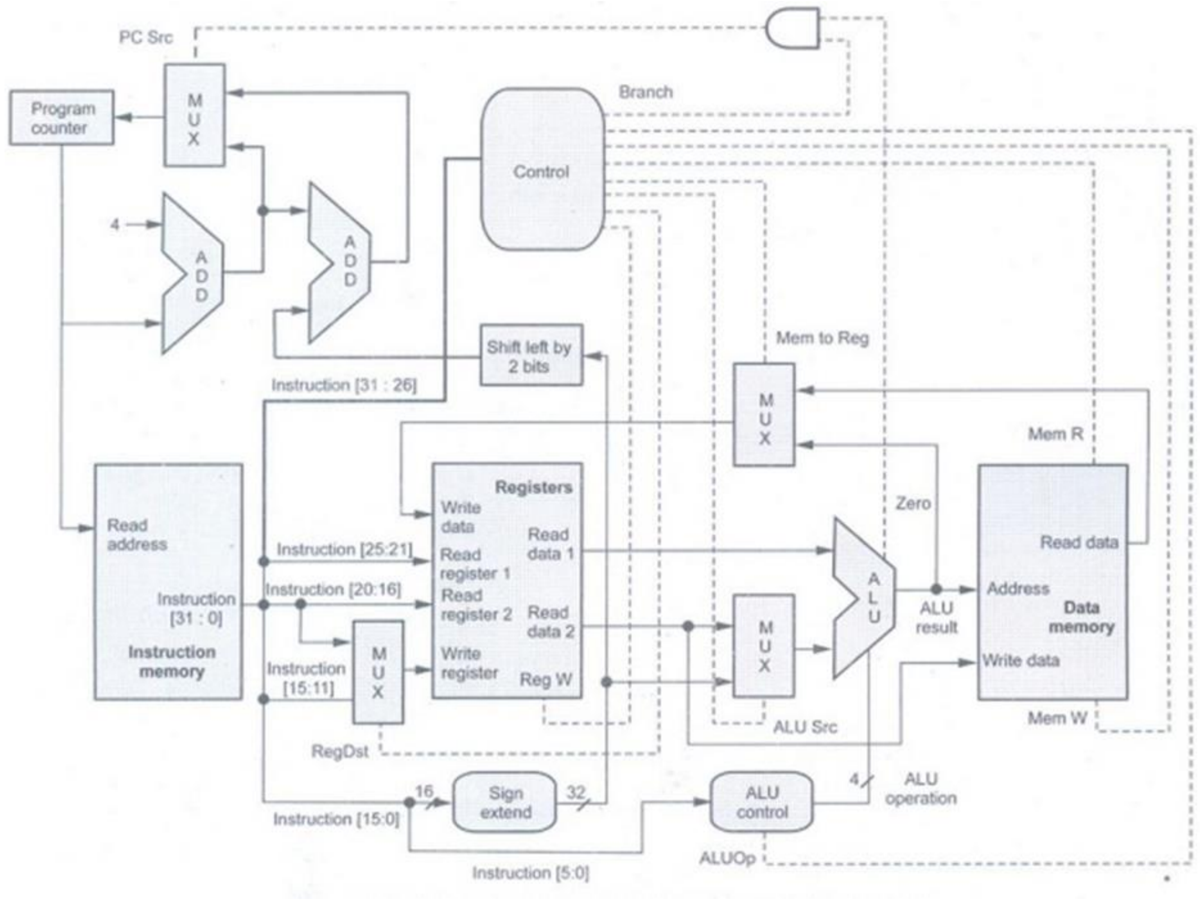
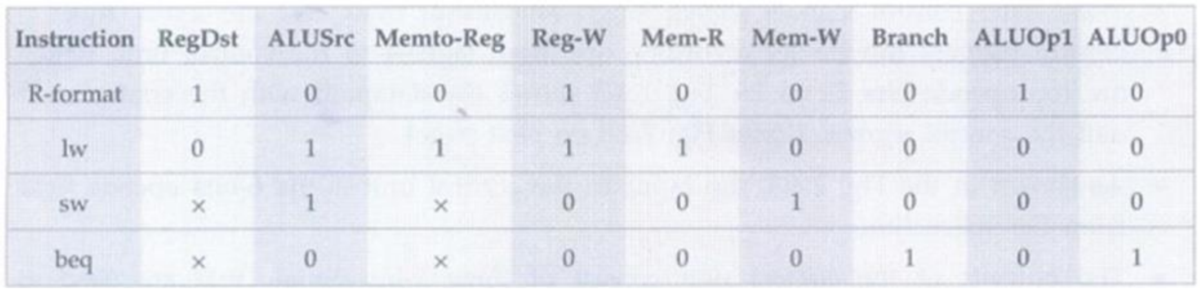


Fig 3.12 Simple datapath with control unit

Table 3.6



Instruction	RegDst	ALUSrc	Memto-Reg	Reg-W	Mem-R	Mem-W	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	×	1	×	0	0	1	0	0	0
beq	×	0	×	0	0	0	1	0	1

For all R-format instructions (add, sub, AND, OR, and slt) 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 (See first row of Table 3.6).

R-type instruction also writes a register (Reg W=1), but neither reads nor writes data memory.

For all R-format instructions, the PC should be unconditionally replaced with PC 4. Thus the Branch control signal is 0; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.

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

The second and third rows of Table 3.6 give the control signal settings for lw and sw. These ALU Src and ALUOP fields are set to perform the address calculation.

The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and Reg W are set for a load to cause the result to be stored into the rt register.

The branch instruction sends the rs and rt registers to the ALU. The ALUOP field for branch is set for a subtract ALU control 01), which is used to test for equality.

It is important to note that the Mem to Reg field is irrelevant when the Reg W signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry Mem to Reg in the last two rows of the Table 3.6 is replaced with X for don't care. Don't cares can also be added to RegDst when Reg W is 0.

3.4.2. Operation of the Datapath

3.4.2.1 Datapath for an R-type Instruction

Fig. 7.4.4 shows the operation of the datapath for an R-type instruction. Here as an example we have considered instruction: add $t1, $t2, $t3. The asserted control signals and active datapath elements are highlighted. Although everything occurs in one clock cycle, the execution of the instruction can be divided into sequential steps according to flow of information as given below:

1. Fetch the instruction and increment the PC.

2.Read data from two registers, $t2 and $t3 and compute the setting of the control lines using main control unit.

3. Generate the ALU function using the function code for addition operation (bits 50, which is the funct field, of the instruction) and perform addition on the data read from the register file.

4. Write the result from the ALU into the register file using bits 15: 11 of the instruction that select the destination register ($t1).

Note The control lines, datapath units, and connections that are active are highlighted.

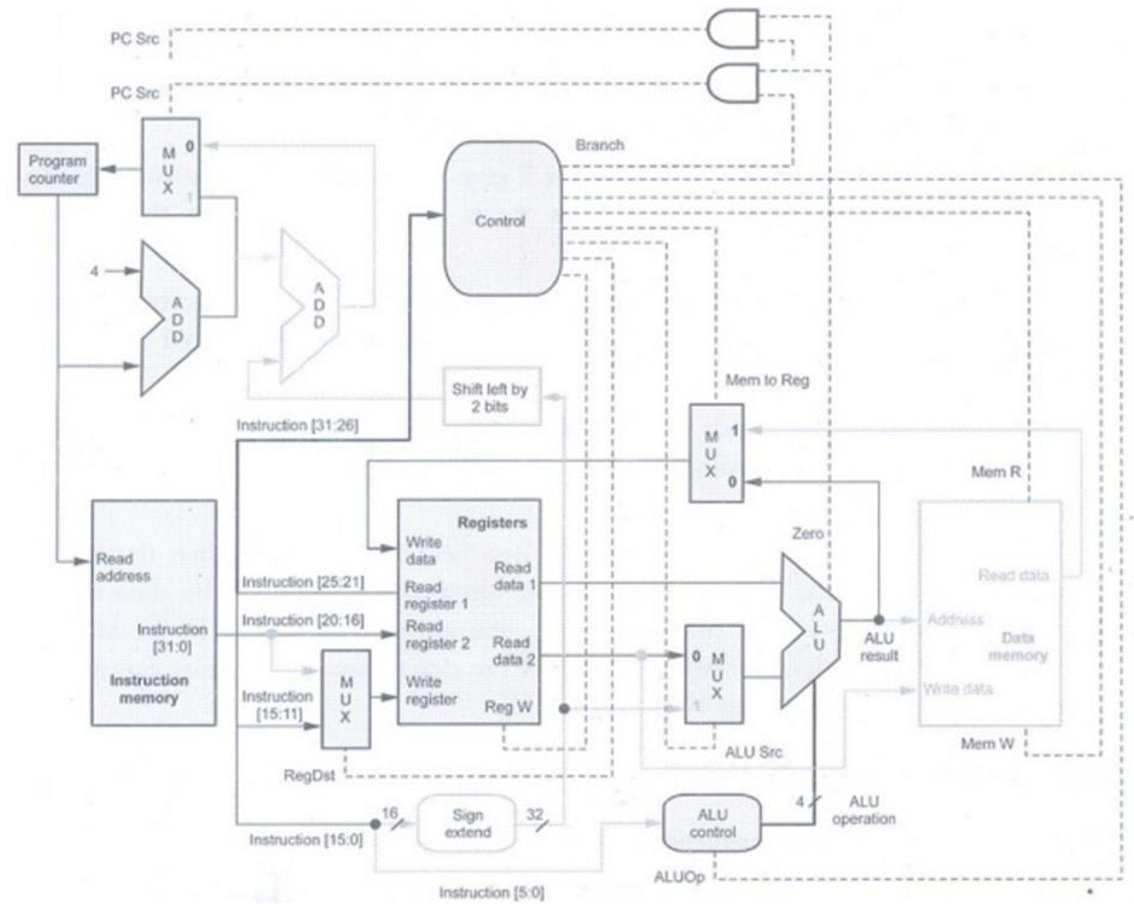


Fig 3.13 Datapath in operation for an R-type instruction(Example : add $t1,$t2,$t3)

Fig. 3.13 shows the operation of the datapath for load word instruction. Here as an example we have considered instruction: Iw $t1, offset($t2). The execution of the load instruction can be divided into sequential steps according to flow of information as given below:

1. Fetch the instruction and increment the PC.

2. Read data from register $t2 of register file.

3. Compute the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset) using ALU.

4. Use the sum from the ALU as the address for the data memory.

5. Write the data from the memory unit into the register file; the register destination is given by bits 20: 16 of the instruction ($t1).

Note The control lines, datapath units, and connections that are active are highlighted.

A load and store instructions operate very similarly. The main differences are that the memory control indicate a write rather than a read, the value of the second

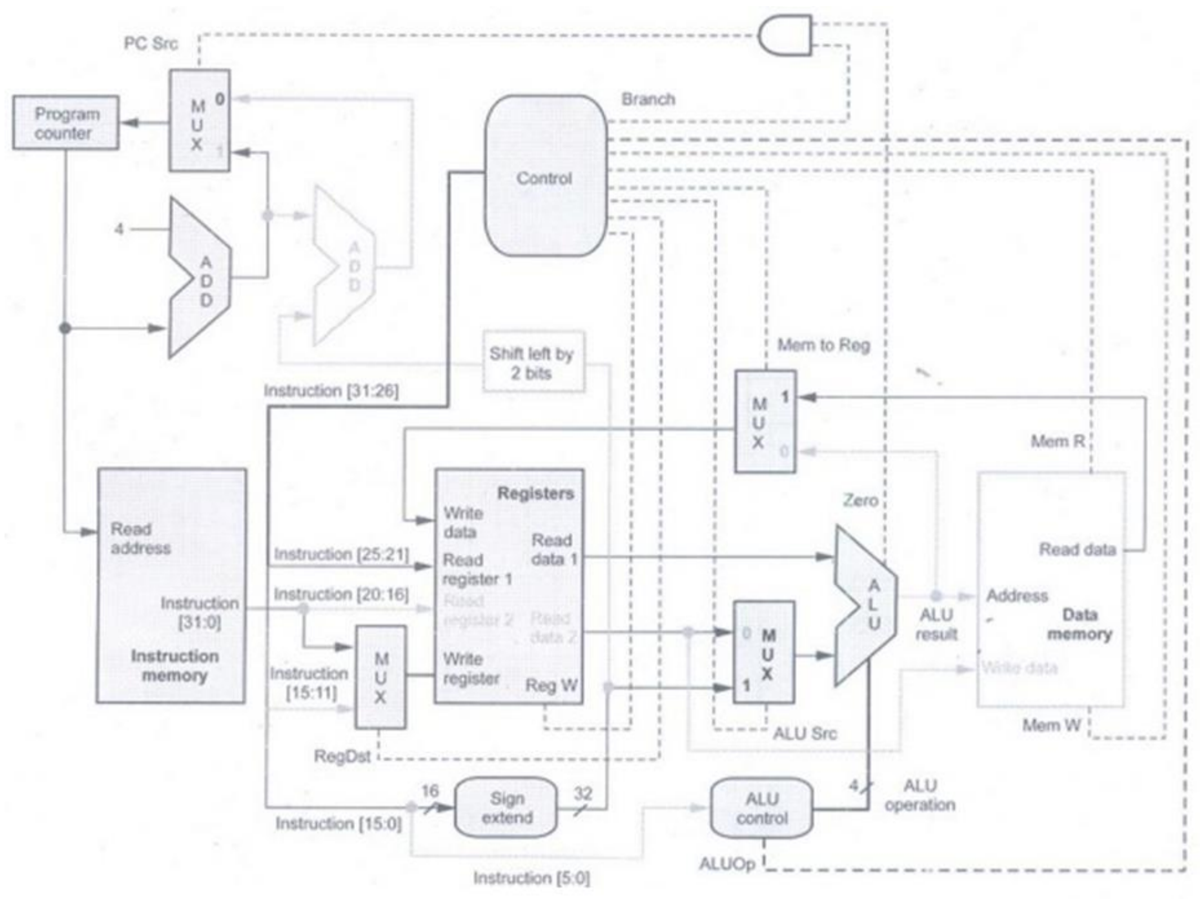


Fig 3.14 Datapath in operation for a load instruction

register is used for the data to store, and the operation of writing the data memory value to the register file is not necessary.

3.4.3 Datapath for Branch - On - Equal Instruction

Fig. 3.14 shows the operation of the datapath for branch-on-equal instruction. Here as an example we have considered instruction: 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. The execution of the branch-on-equal instruction can be divided into sequential steps according to flow of information as given below :

3.4.3.1. Fetch the instruction and increment the PC.

3.4.3.2. Read data from two registers, $t2 and $t3.

3.4.3.3. Generate the ALU function using the function code for subtract operation (bits 5 0, which is the funct field, of the instruction) and perform subtraction on the data read from the register file. Add the value of PC + 4 to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two to get the branch target address.

3.4.3.4.Use the Zero result from the ALU to decide which adder result to store into the PC.

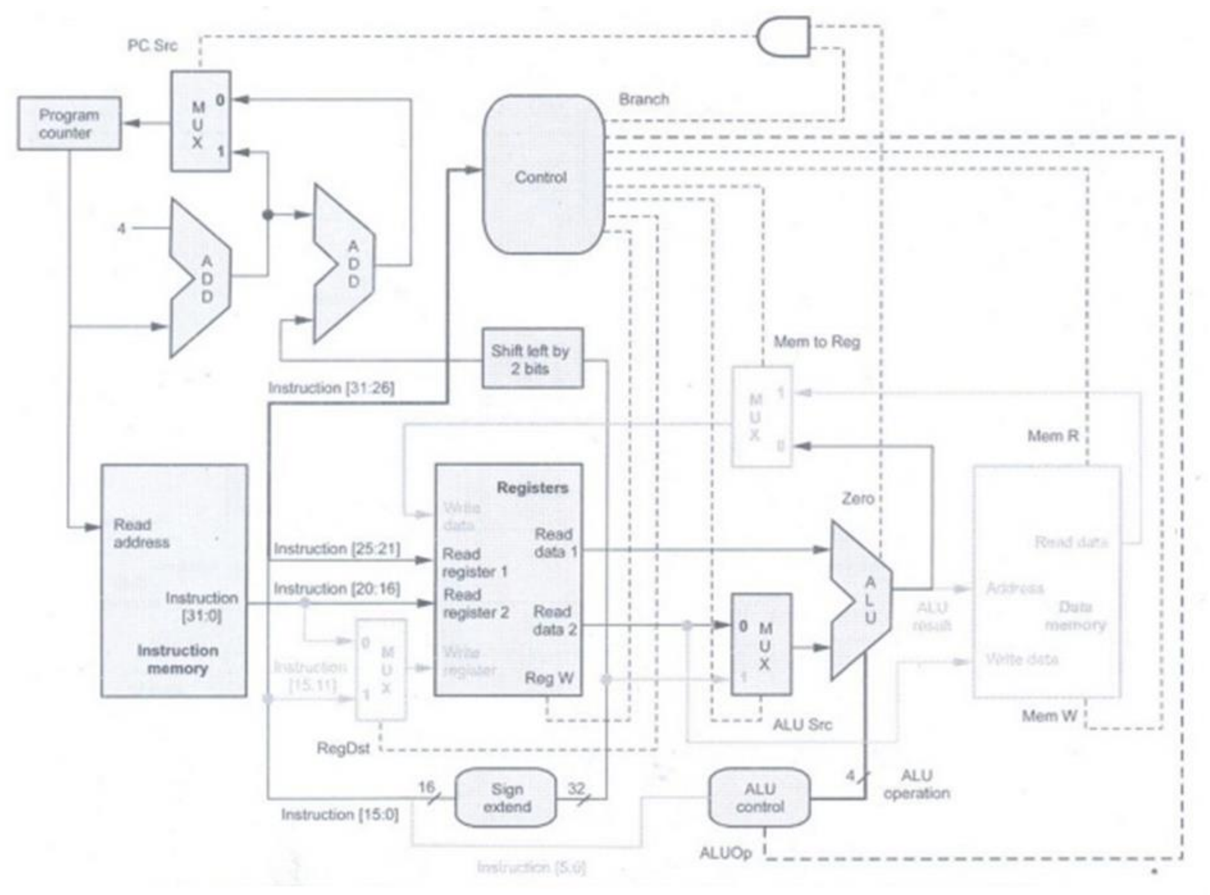


Fig 3.15 Datapath in operation for a branch-on-equal instruction

Note The control lines, datapath units and connections that are active are highlighted.

3.4.4. Finalizing Control

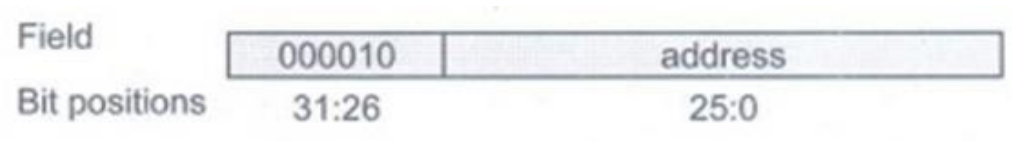
The control function can be precisely defined using the contents of Table 3.7.. The outputs are the control lines, and the input is the 6-bits opcode field, Op [5: 0], i.e. bits 31 26 of the instruction. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes as shown in the Table 3.7.

Table 3.7. Truth table for output control lines and ALUOp

Instruction	Input Op [5 : 0]						Reg Dst	ALU scr	Mem toReg	Reg W	Mem R	Mem W	Branch	ALU Op1	ALU Op2
	5	4	3	2	1	0									
R-format	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0
lw	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
sw	1	0	1	0	1	1	x	1	x	0	0	1	0	0	0
beq	0	0	0	1	0	0	x	0	x	0	0	0	1	0	1

3.4.5. Implementing Jumps

The jump instruction is not conditional. Fig 3.16 shows the format of jump instruction.



Field	000010	address
Bit positions	31:26	25:0

Fig 3.16 Format of jump instruction

Like a branch, the low-order 2 bits of a jump address are always 00_2. The next lower 26 bits of this 32-bits address come from the 26-bits immediate (address) 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-bits immediate field of the jump instruction

The bits 00_{two}

Fig. 3.17 shows the datapath operation for jump instruction. Here, additional control is added to Fig. 3.12 for jump instruction. An additional multiplexer is used to select the source for the new PC value, which is either the incremented PC (PC + 4), the branch target PC, or the jump target PC. One additional control signal is added for the additional multiplexer. This control signal, called Jump, is asserted only when the instruction is a jump-that is, when the opcode is 2.

3.4.6. Reasons for not using Single - Cycle Implementation

3.4.6.1. It is inefficient. Because the longest possible path in the processor determines the clock cycle. Remember that the clock cycle must have the same length for every instruction in single-cycle design.

3.4.6.2. The overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

3.4.6.3. The penalty for using the single-cycle design with a fixed clock cycle is significant. Single-cycle designs for floating-point unit or an instruction set with more complex instructions do not work well at all.

3.4.6.4. It do not improve the worst-case cycle time. Thus it violates the great idea of making the common case fast.

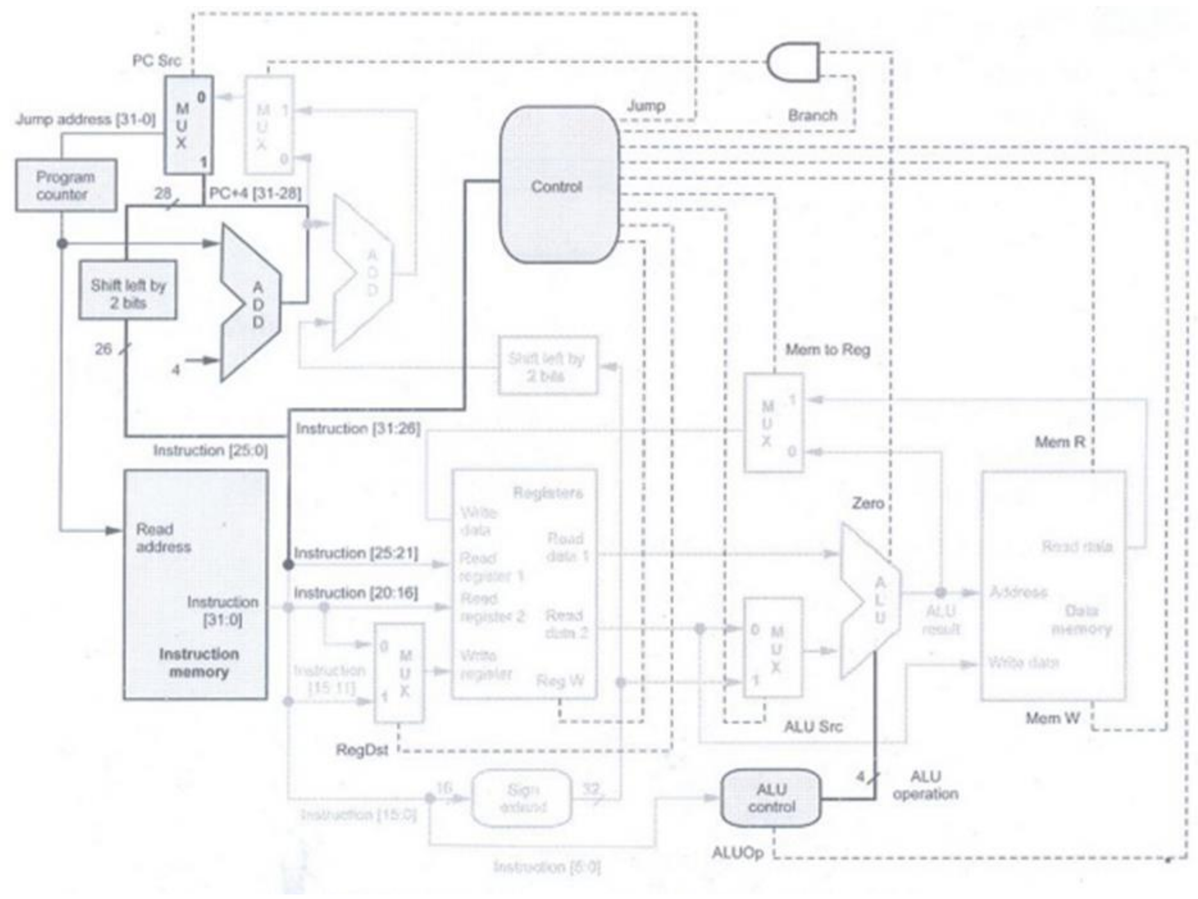


Fig 3.17 Datapath operation for jump instruction

3.5. Pipelining

We have seen various cycles involved in the instruction cycle. These fetch, decode and execute cycles for several instructions are performed simultaneously to reduce overall processing time. This process is referred to as instruction pipelining.

To apply the concept of instruction pipelining, we must subdivide instruction processing in number of stages as given below.

S_1 - Fetch (F): Read instruction from the memory.

S_2 - Decode (D): Decode the opcode and fetch source operand (s) if necessary.

S_3 - Execute (E): Perform the operation specified by the instruction.

S_4 - Store (S): Store the result in the destination.

Here, instruction processing is divided into four stages hence it is known as four-stage instruction pipeline. With this subdivision and assuming equal duration for each stage we can reduce the execution time for 4 instructions from 16 time units to 7 time units. This is illustrated in Fig. 3.18.

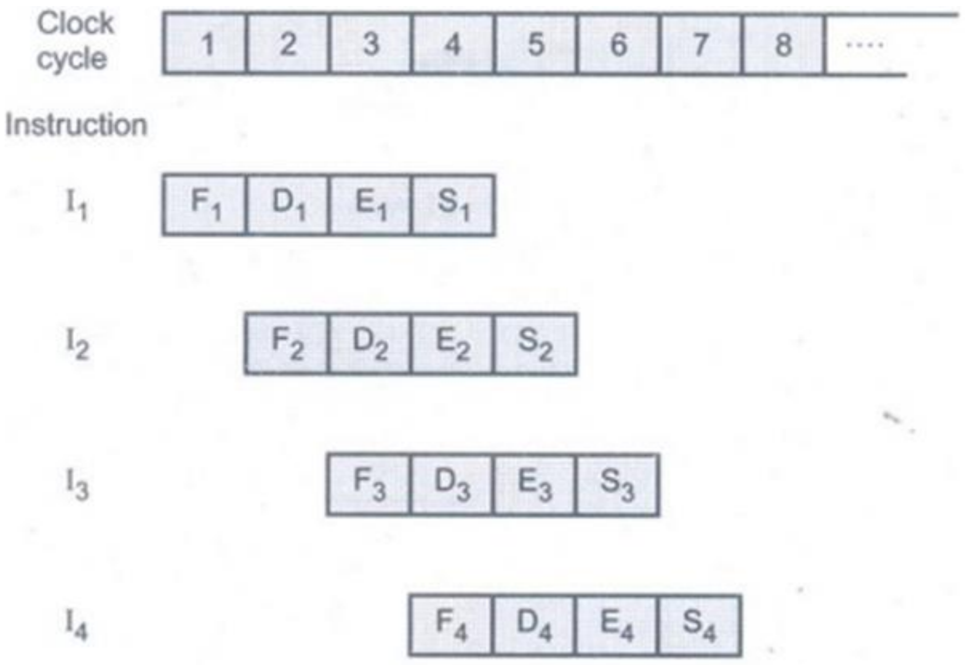

Clock cycle
1
2
3
4
5
6
7
8
....
Instruction
I_1 F_1 D_1 E_1 S_1
I_2 F_2 D_2 E_2 S_2
I_3 F_3 D_3 E_3 S_3
I_4 F_4 D_4 E_4 S_4

Fig 3.18 Four stage instruction pipelining

In this instruction pipelining four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Fig. 3.19. These units are implemented such that they are capable of performing their tasks simultaneously and without interfering with one another. Information from the stage is passed to the next stage with the help of buffers.

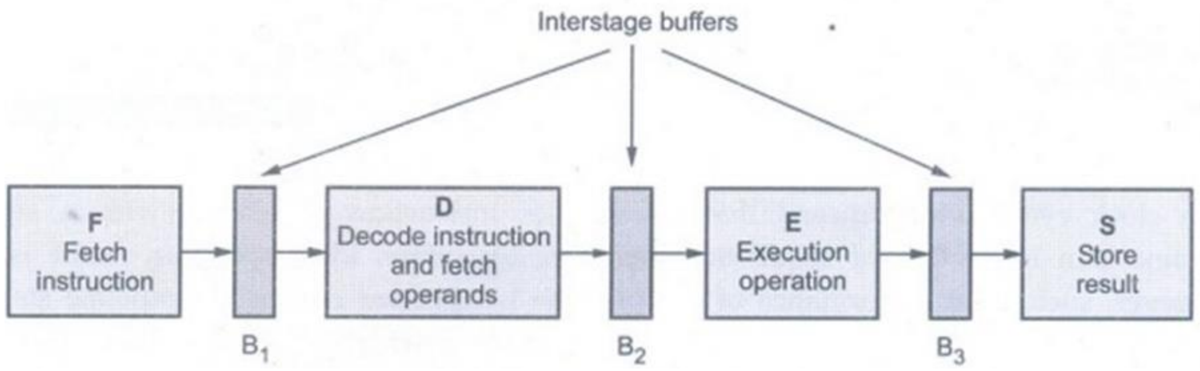


Fig 3.19 Hardware organization for four-stage instruction pipeline

Example 3.2.1 Explain the function of a six segment pipeline and draw a space diagram for a six segment pipeline showing the time it takes to process eight tasks.

Solution: Six stages in the pipeline :

1) Fetch Instruction (FI): Read the next expected instruction into a buffer.

2) Decode Instruction (DI): Determine the opcode and the operand specifiers.

3) Calculate Operands (CO): Calculate the effective address of each source operand.

4) Fetch Operands (FO): Fetch each operand from memory.

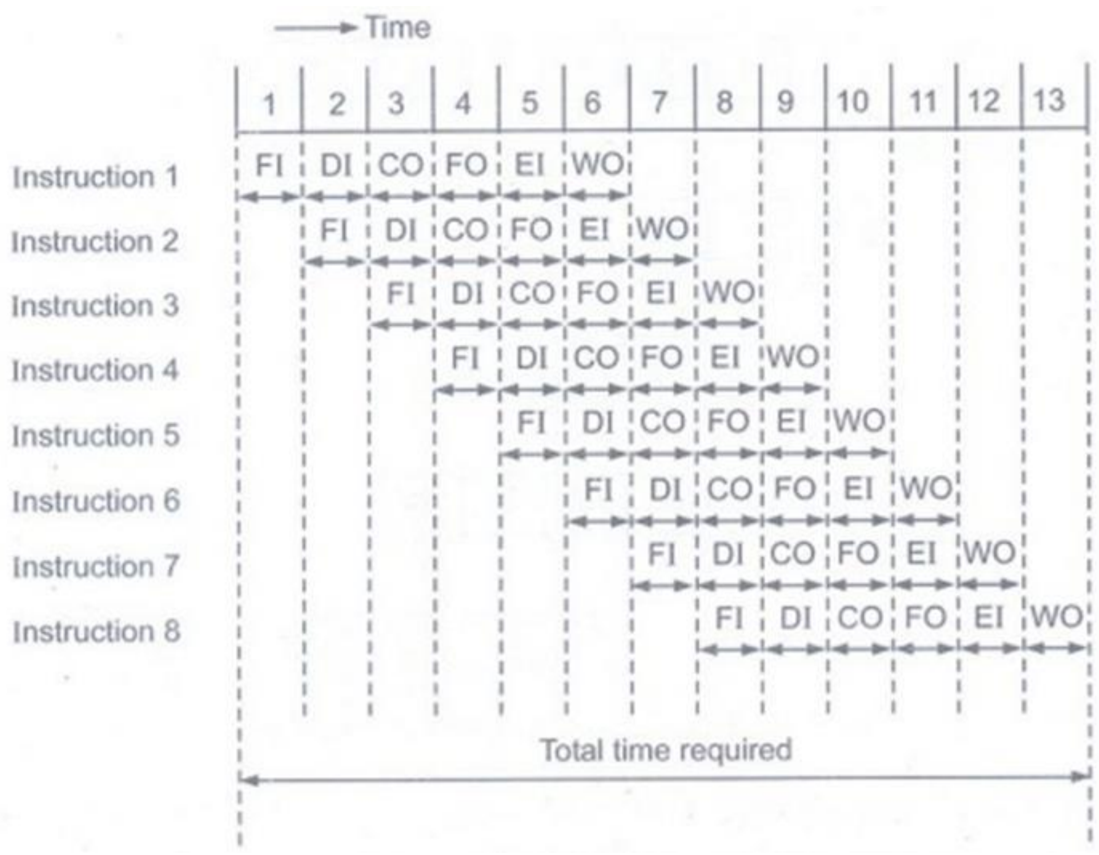


Fig 3.20 Total time required

5) Execute Instruction (EI): Perform the indicated operation and store the result, if any in the specified destination operand location.

6) Write Operand (WO): Store the result in memory.

Example 3.2.2 What is the ideal speed-up expected in a pipelined architecture with 'n' stages? Justify your answer.

Solution: The pipelined processor ideally completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing with n stage pipeline is n times that of sequential operation. Therefore, ideal speed-up factor is n. However, such ideal performance of the pipeline is achieved only when pipeline stages must complete their processing tasks for a given instruction in the time allotted. Unfortunately, this is not the case; pipeline operations could not sustained without interruption throughout the program execution.

Pipeline Stages in the MIPS Instructions

1. Fetch instruction from memory.

2. Read registers while decoding the instruction. The regular 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.

3.5.1. Designing Instruction Sets for Pipelining

From the following points we can realized that the MIPS instruction set is designed for pipelined execution.

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 being located in the same place in each instruction. Due to this symmetry the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched.

3. Memory operands only appear in loads or stores in MIPS. Due to this restriction we can use the execute stage to calculate the memory address and then access memory in the following stage.

4. Since operands are aligned in memory, data can be transferred between processor and memory in a single pipeline stage.

3.5.2 Pipeline Hazards

The timing diagram for instruction pipeline operation shown in Fig. 3.20 completes the processing of one instruction in each clock cycle. This means that the rate of instruction processing is four times that of sequential operation.

The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation shown in Fig. 3.20 could be performed without any interruption throughout program execution. Unfortunately, this is not the case.

For many of reasons, one of the pipeline stages may not be able to complete its operation in the allotted time.

Fig. 7.8.4 shows an example in which the operation specified in instruction 2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the information in buffer B_2 must remain intact until the instruction execution stage has completed its operation. This means that stage 2 and in turn, stage 1 are blocked from accepting new instructions because the information in B_1 cannot be overwritten. Thus decode step for instruction and fetch step for instruction 5 must be postponed as shown in the Fig. 7.8.4.

The instruction pipeline shown in Fig. 7.8.4 is said to have been stalled for two clock cycles (clock cycles 5 and 6) and normal pipeline operation resumes in clock cycle 7.

Any reason that causes the pipeline to stall is called a hazard.

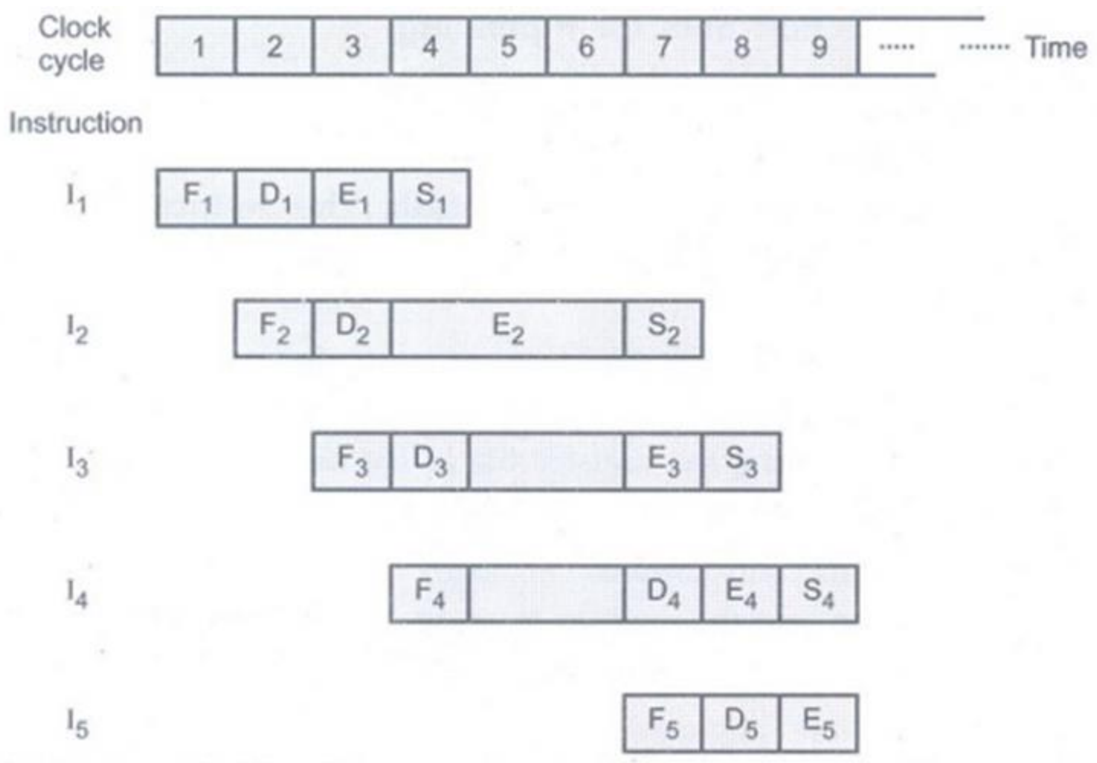


Fig 3.21 Effect on pipeline of an execution operation taking more than one clock cycle

3.5.3. Types of Hazards

3.5.3.1. Structural hazards: These hazards are because of conflicts due to insufficient resources when even with all possible combination, it may not be possible to overlap the operation.

3.5.3.2. Data or data dependent hazards: These result when instruction in the pipeline depends on the result of previous instructions which are still in pipeline and not completed.

3.5.3.3. Instruction or control hazards: They arise while pipelining branch and other instructions that change the contents of program counter. The simplest way to handle these hazards is to stall the pipeline. Stalling of the pipeline allows few instructions to proceed to completion while stopping the execution of those which results in hazards.

3.5.4. Structural Hazards

The performance of pipelined processor depends on whether the functional units are pipelined and whether they are multiple execution units to allow all possible combination of instructions in the pipeline. If for some combination, pipeline has to be stalled to avoid the resource conflicts then there is a structural hazard.

In other words, we can say that when two instructions require the use of a given hardware resource at the same time, the structural hazard occurs.

The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory for storage of the result while another

instruction or operand needed is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. To avoid such type of structural hazards many processors use separate caches for instruction and data.

3.5.5. Data Hazards

When either the source or the destination operands of an instruction are not available at the time expected in the pipeline and as a result pipeline is stalled, we say such a situation is a data hazard.

Consider a program with two ins ructions, I_1followed by I_2. When this program is executed in a pipeline, the execution of these two instructions can be performed concurrently. In such case the result of I_1 may not be available for the execution of I_2. If the result of I_2 is dependent on the result of I_1 we may get incorrect result if both are executed concurrently. For example, assume A = 10 in the following two operations :

I_1: $A \leftarrow A + 5$

I_2: $B \leftarrow A \times 2$

When these two operations are performed in the given order, one after the other, we get result 30. But if they are performed concurrently, the value of A used in computing B would be the original value, 10, leading to an incorrect result. In this case data used in the I_2 depend on the result of I_1. The hazard due to such situation is called data hazard or data dependent hazard. To avoid incorrect results we have to execute dependent instructions one after the other (in-order).

3.5.6. Control (Instruction) Hazards

The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. This stream is interrupted when pipeline stall occurs either due to cache miss or due to branch instruction. Such a situation is known as instruction hazard.

Instruction hazard can cause greater degradation in performance than data hazards.

3.5.7. Unconditional Branching

Fig. 7.8.5 shows a sequence of instructions being executed in a two-stage pipeline. The instruction I_2 is a branch instruction and its target instruction is I_K. In clock cycle 3, the instruction I_3 is fetched and at the same time branch instruction (I_2) is decoded and the target address is computed. In clock cycle 4, the incorrectly fetched instruction I_3 is discarded and instruction I_K is fetched. During this time execution unit is idle and pipeline is stalled for one clock cycle.

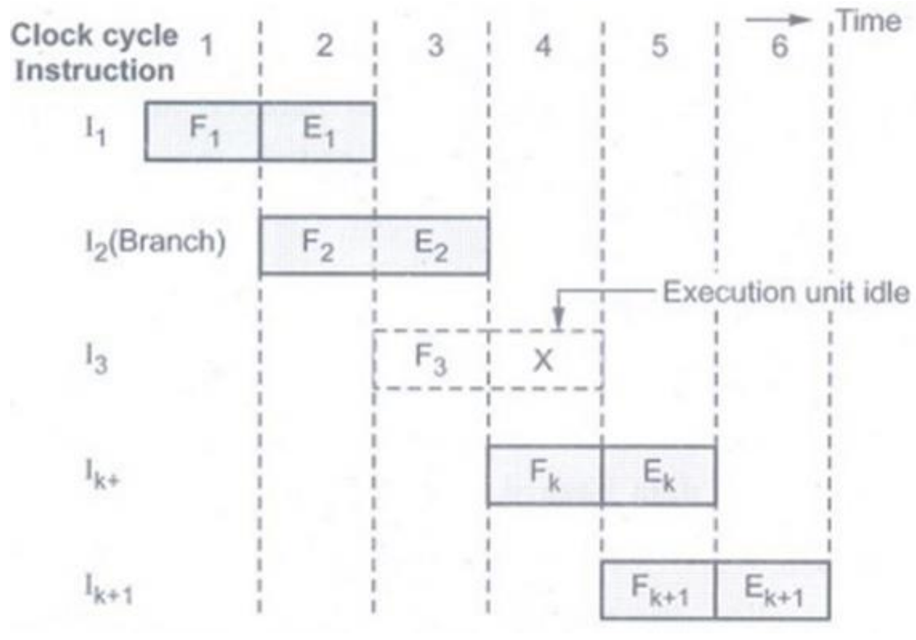


Fig 3.22 Effect of branching in two-stage pipelining

Branch Penalty: The time lost as a result of a branch instruction is often referred to as the branch penalty.

3.5.8 Factor effecting branch penalty

1. It is more for complex instructions.

2. For a longer pipeline, branch penalty is more..

In case of longer pipelines, the branch penalty can be reduced by computing the branch address earlier in the pipeline.

3.6. Pipelined Datapath and Control

The Fig. 3.23 shows the general structure of multistage pipeline. As shown in the Fig. 3.23, the usually pipeline processor consists of a sequence of m data processing circuits, called elements, stages or segments.

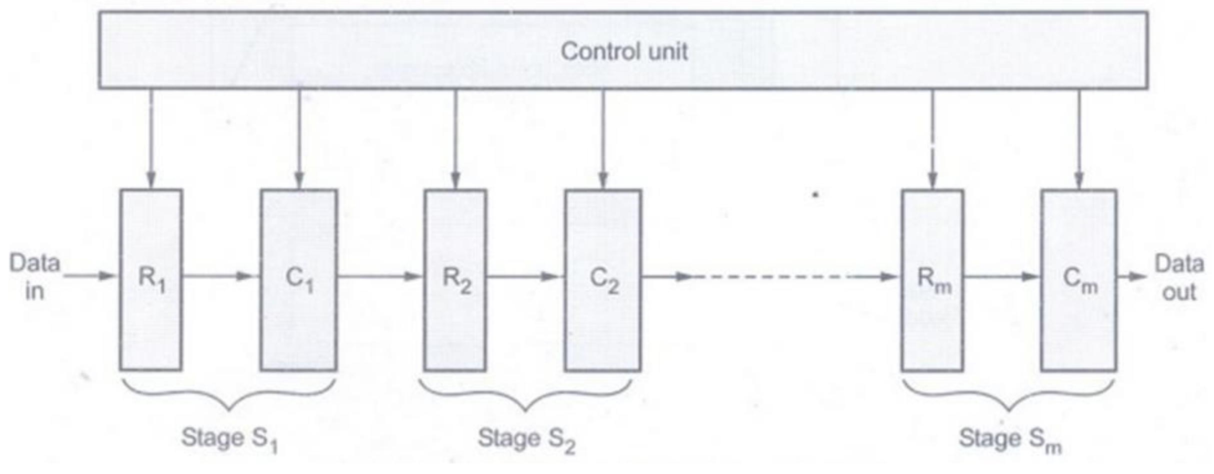


Fig 3.23 Structure of pipeline processor

These stages collectively perform a single operation on a stream of data operands passing through them. The processing is done part by part in each stage, but the final result is obtained only after an operand set has passed through the entire pipeline.

Each stage consists of two major blocks : Multiword input register and datapath circuit.

The multiword input registers R_i, hold partially processed results as they move through the pipeline and they also serves as buffers that prevent neighbouring stages from interfering with one another. In each clock period the individual stages process its data and transfers its results to the next stage.

The m-stage pipeline processor shown in Fig. 3.23 can simultaneously process up to m independent sets of data operands. Thus when the pipeline is full, m separate operations are being executed concurrently, each in a different stage. This gives a new final result from the pipeline every clock cycle.

If time required to perform single sub operation in the pipeline is T seconds then for m stage pipeline the time required to complete a single operation is mT seconds. This is called delay or latency of the pipeline.

The maximum number of operations completed per second can be given as 1/T. This is called throughput of the pipeline.

3.6.1 Implementation of Two Stage Instruction Pipelining

The simplest instruction pipelining breaks instruction processing into two parts: A fetch stage S_1 and an execute stage S_2. When these two stages are overlapped, we get two stage pipelining with increased throughput.

The Fig. 3.24 shows an implementation of a two-stage instruction pipeline. The fetch stage S_1 consists of the microprogram counter μPC, which is the source for

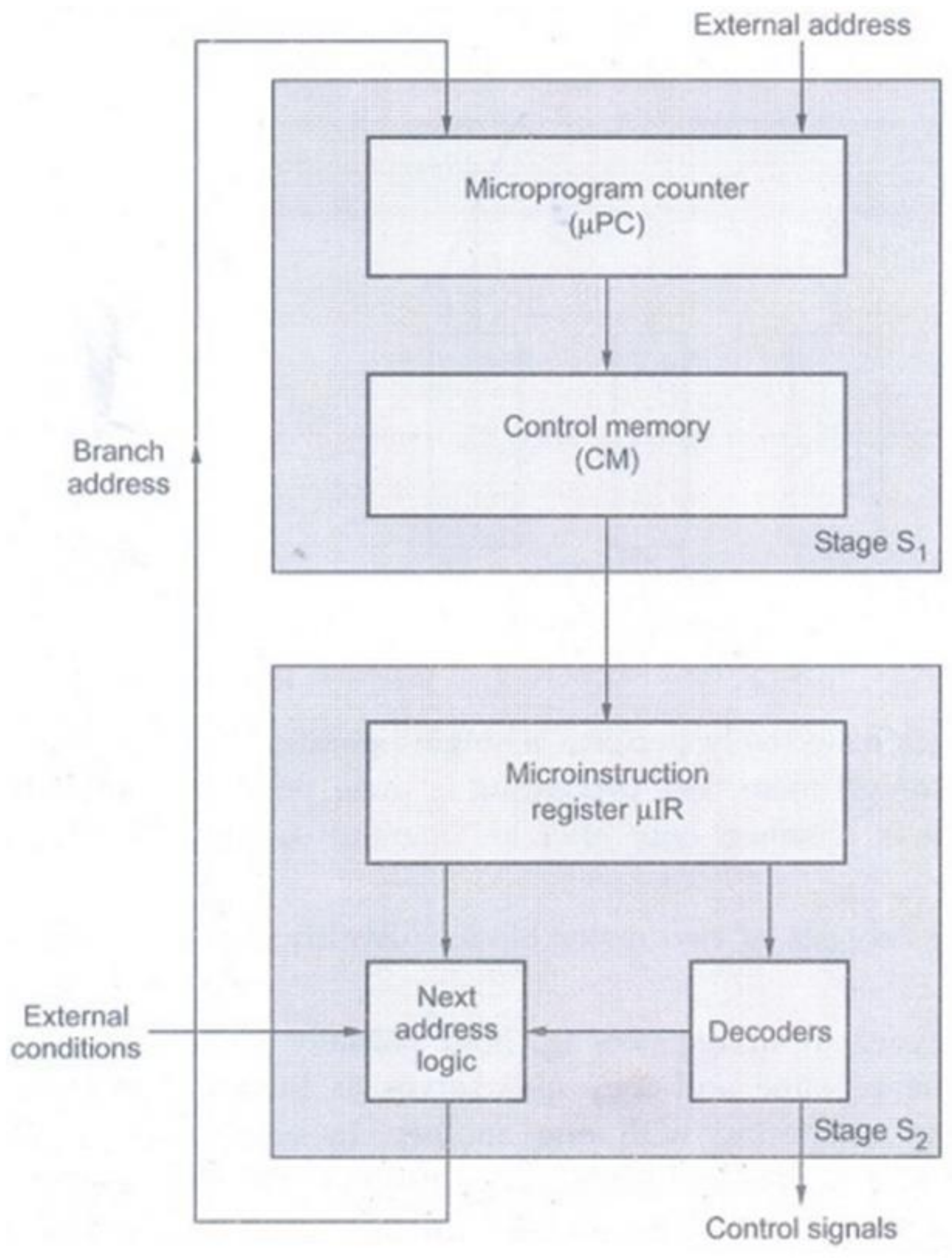


Fig 3.24 Two-stage pipelined microprogram control unit

microinstruction addresses, and the control memory CM, which stores the microinstructions.

The execution stage S_2consists of microinstruction register µIR, the decoders that extract control signals from the microinstructions in µIR, and the logic for determining next address or the branch address.

The microinstruction register acts as a buffer register for stage S_2. With these two stages it is possible that, while instruction I_i with address A_i is being executed by stage S_2, the instruction I_{i+1} with the next consecutive address$A_{i+1}$is fetched from memory by stage S_1. If on executing I_i in S_2 it is determined that a branch must be made to a nonconsecutive address, then the prefetched instruction I_{i+1} in S_1has to be discarded. In such cases branch address is obtained directly from µI itself and fed back to S_1. The branch address is then loaded into µPC and next instruction is fetched from the branch address.

3.6.2. Organization of CPU with Four Stage Instruction Pipelining

The Fig. 3.25 shows the implementation of four-stage instruction pipelining. As shown in the Fig. 3.25 the CPU is directly connected to a cache memory, which is split into instruction and data parts, called the I-cache and D-cache, respectively. This splitting of the cache permits both an instruction word and a memory data word to be accessed in the same clock cycle.

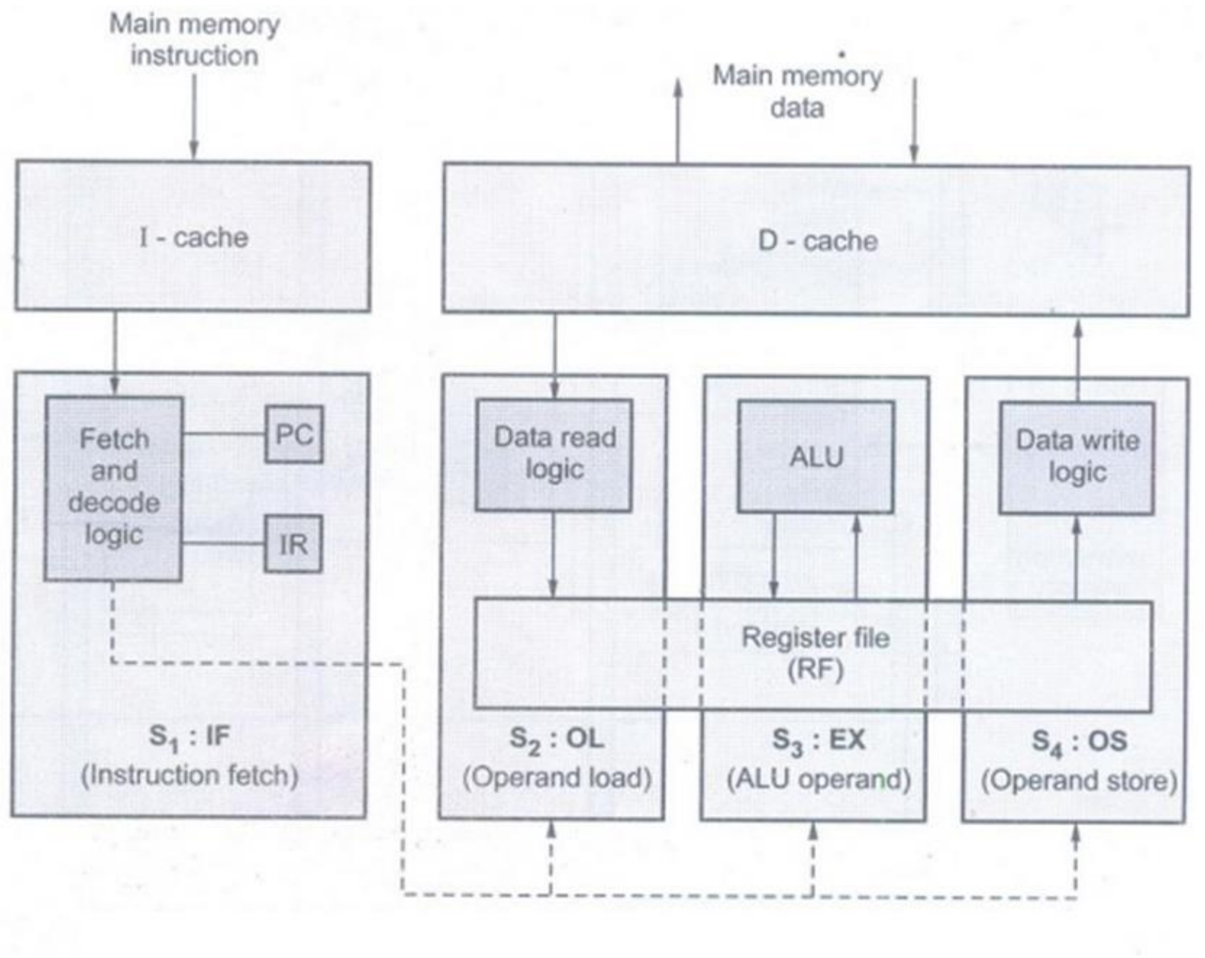


Fig 3.25 Organization of CPU for four-stage instruction pipelining

The four stages S_1:S_4 shown in Fig. 3.25 perform the following functions:

- S_1: IF: Instruction fetching and decoding using the I cache.
- S_2: OL: Operand loading from the D-cache to register file.
- S_3: EX: Data processing using the ALU and register file.
- S_4: OS: Operand storing to the D-cache from register file.

Stages S_2 and S_4 implements memory load and store operations, respectively.

Stages S_2, $S_3$and S_4 share the CPU's local Register File (RF). The registers in the register file act as interstage buffer registers.

The stage 3 implements data transfer and data processing operations of the register to register type using ALU of the CPU.

3.6.3. Implementation of MIPS Instruction Pipeline

Fig. 3.26 shows the single-cycle datapath with the pipeline stages. The instruction execution is divided into five stages means a five-stage pipeline.

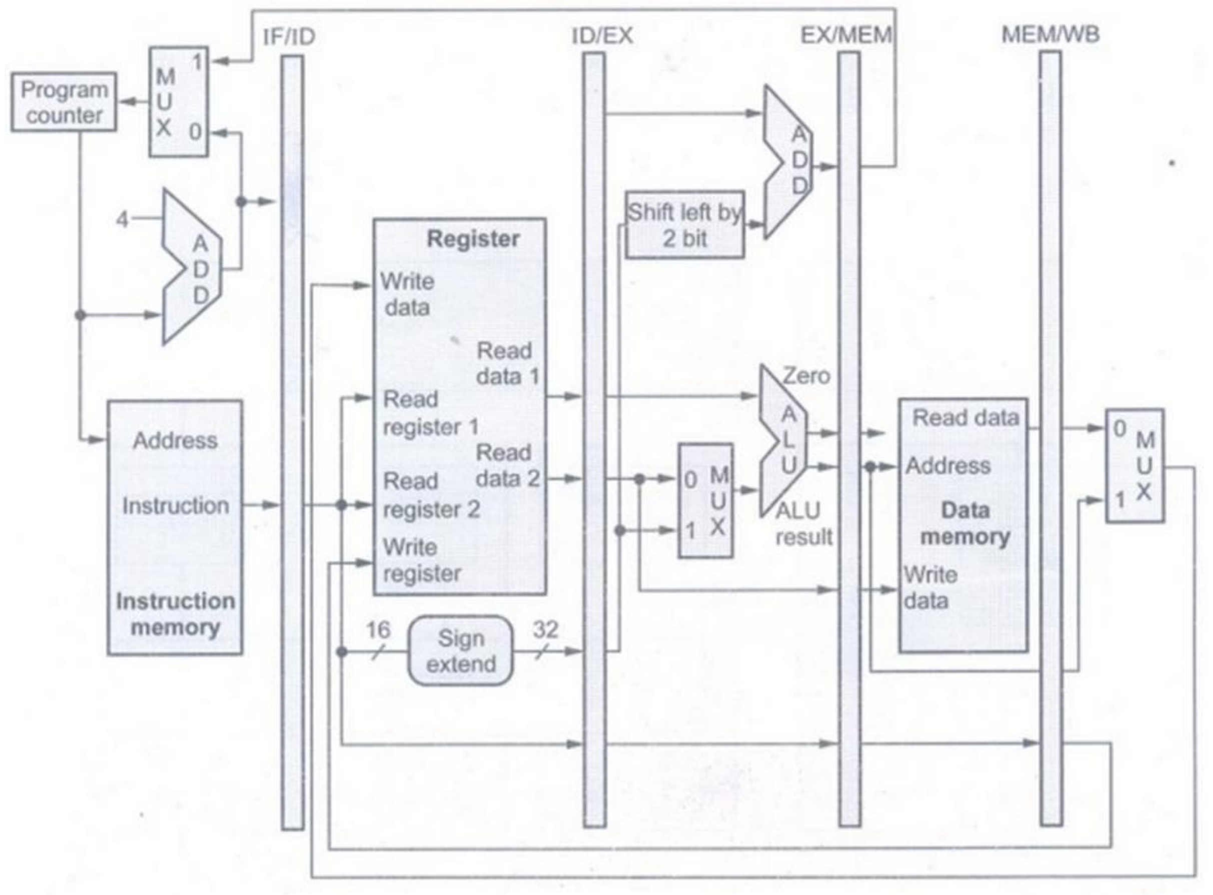


Fig 3.26 Single-cycle datapath with the pipeline stages

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.

In this pipeline stages, 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.

3.6.4. Pipeline Control

Fig. 3.27 shows the single-cycle datapath with added control to the pipelined datapath.

This datapath uses the control logic for PC source, register destination number, and ALU control discussed in the previous section.

Here, we need the 6-bits funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.

The 6-bits of funct field are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

We have already assumed that the PC is written on each clock cycle, so there is no separate write signal for the PC. Similarly, 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.

As shown in the Fig. 3.27 each control line is associated with a component active in only a single pipeline stage. Thus, we can divide the control lines into five groups according to the pipeline stage.

3.6.4.1. 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.

3.6.4.2. Instruction decode/register file read: Nothing special to control in this pipeline stage.

3.6.4.3. Execution/address calculation: The signals need to be control 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.

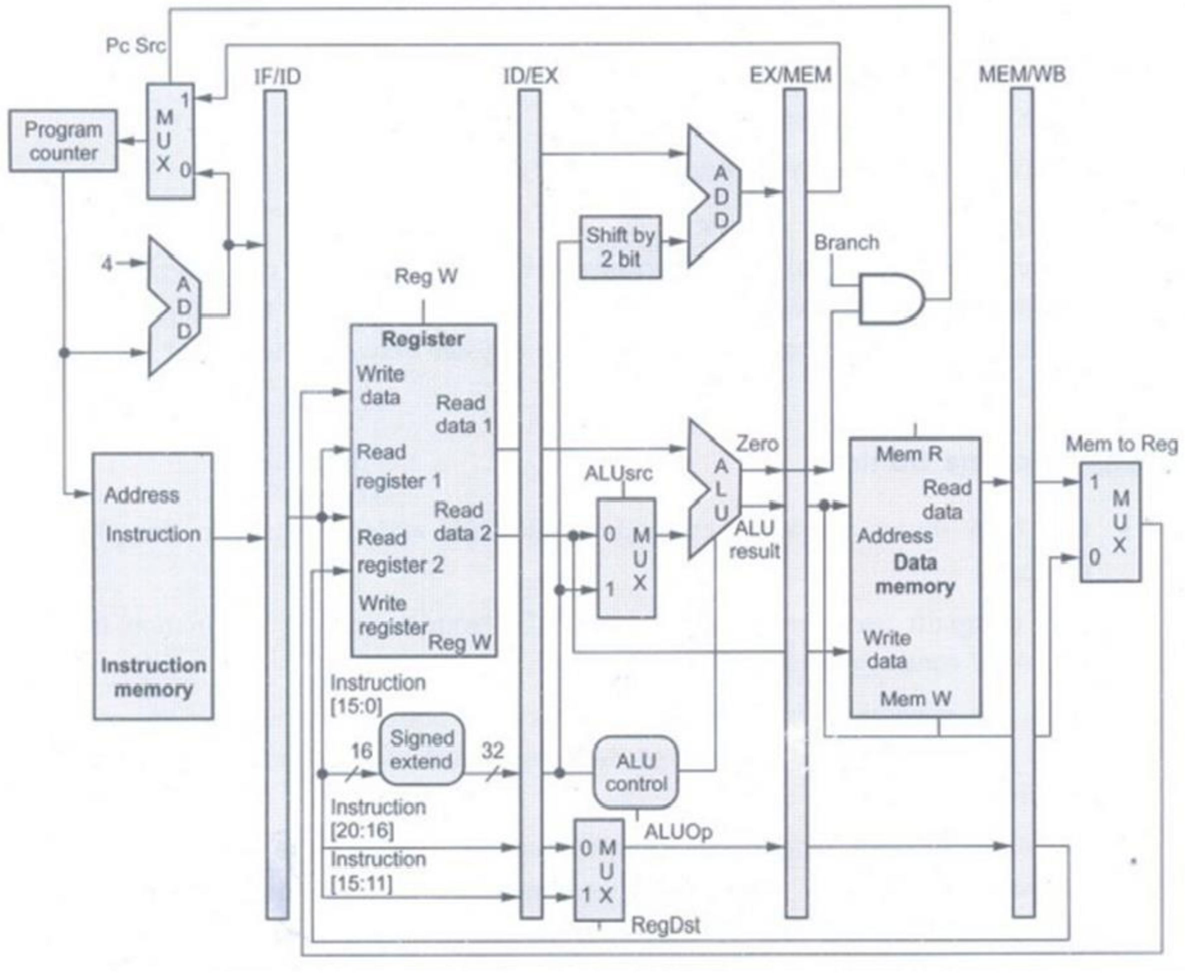


Fig 3.27 Single-cycle datapath with added control to the pipelined datapath

3.6.4.4. Memory access: The signals set in this stage are Branch, MemRead, and MemWrite. The branch equal, load, and store instructions set these signals, respectively. The PC Src selects the next sequential address unless control asserts Branch and the ALU result equal to 0.

3.6.4.5. Write-back: The two control signals set in this stage are Mem to Reg and RegWrite. The Mem to Reg signal decides between sending the ALU result or the memory value to the register file, and RegWrite signal writes the chosen value.

The nine control lines we have seen in Fig. 3.28 are grouped here by pipeline stage. Thus implementing control means setting the nine control lines in each stage for each instruction.

The simplest way to do this is to extend the pipeline registers to include control information as shown in Fig. 3.28.

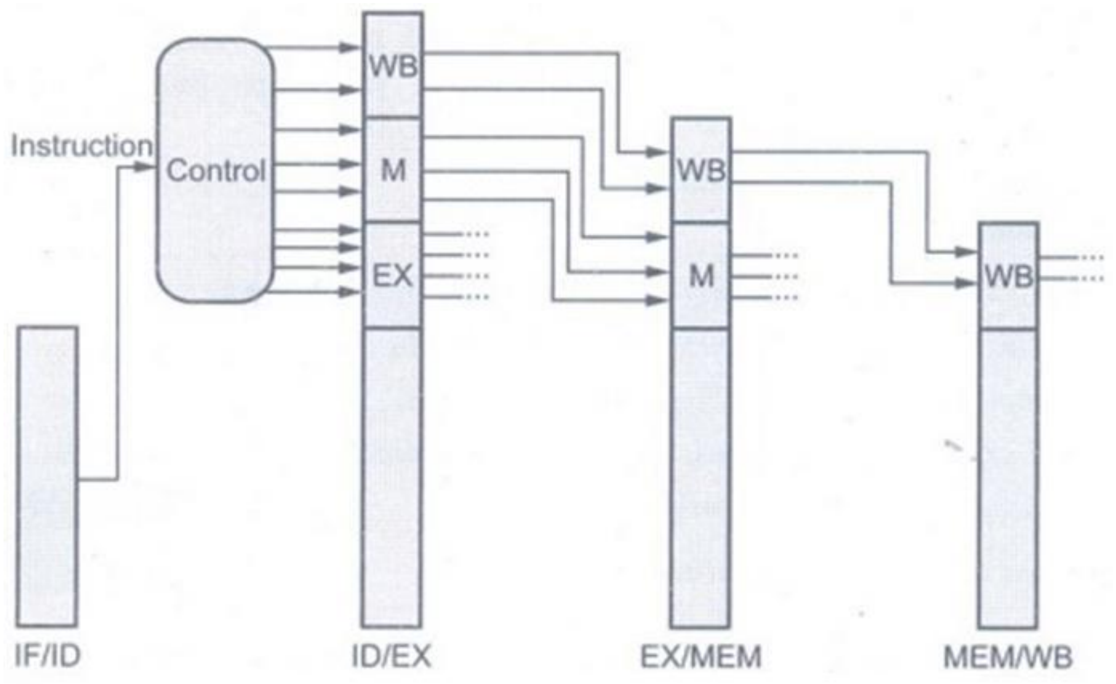


Fig 3.28 Control lines for the final three stages

Fig. 3.29 shows the full datapath with the extended pipeline registers and with the control lines connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode

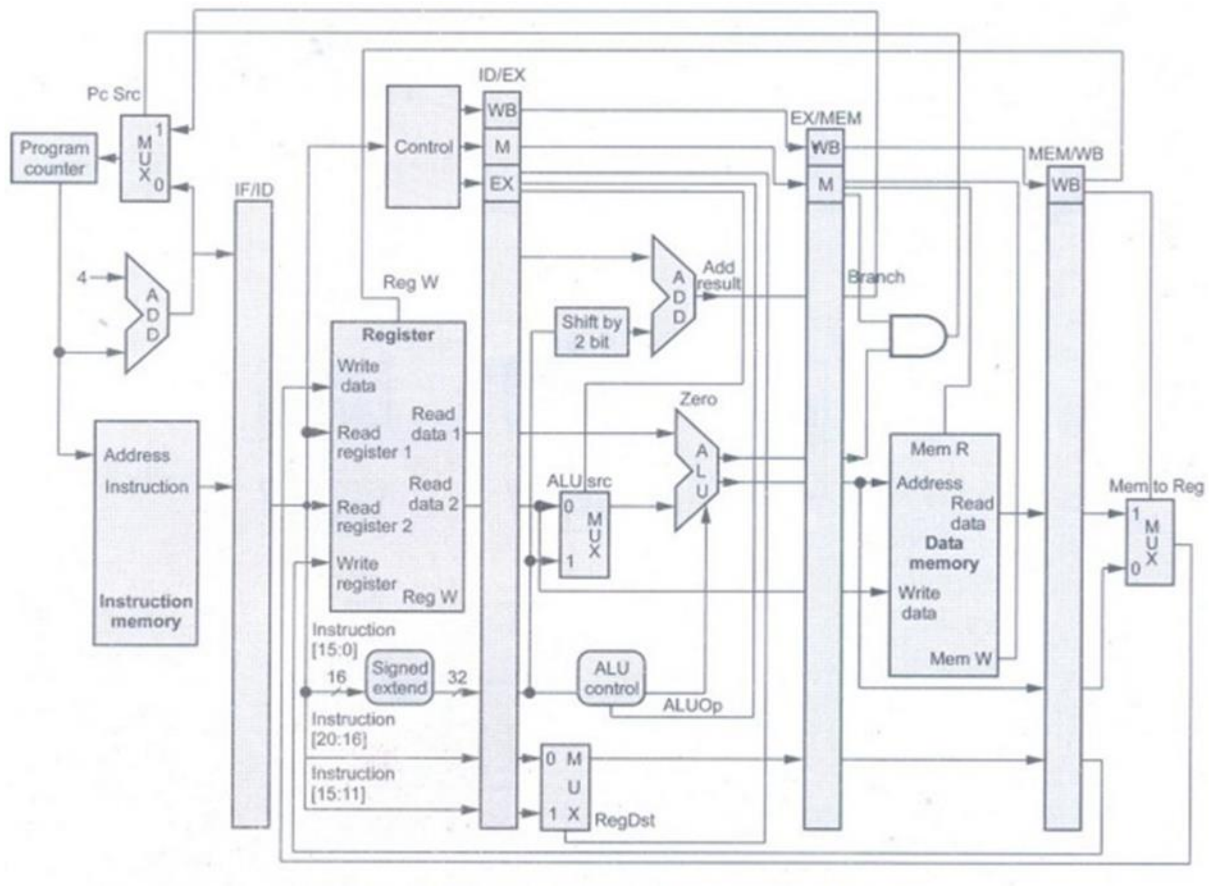


Fig 3.29 Pipelined datapath with control signals

stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

3.7. Handling Data Hazards

3.7.1. Operand Forwarding

A simple hardware technique which can handle data hazard is called operand forwarding or register by passing.

In this technique, ALU results are fed back as ALU inputs. When the forwarding logic detects the previous ALU operation has the operand for current instruction, it forwards ALU output instead of register file.

This is illustrated in Fig. 3.30 shows a portion of the processor datapath involving the ALU and the register file.

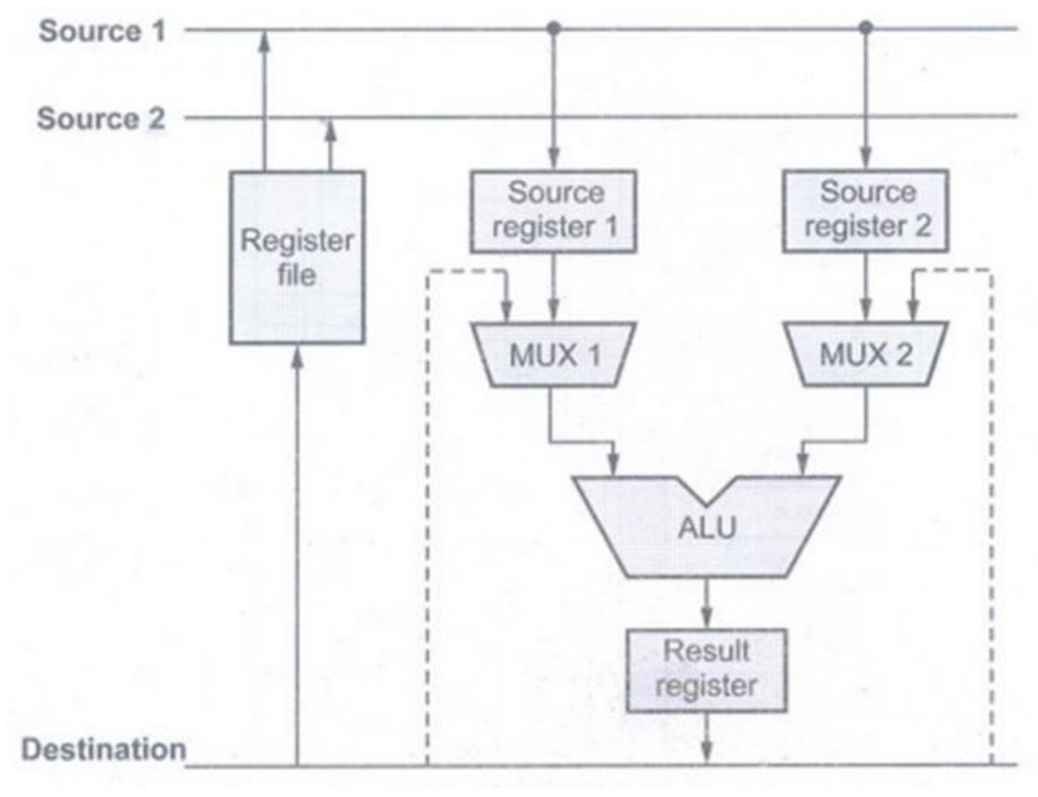


Fig 3.30 Operand forwarding in a pipelined processor- Data path

The source and result register constitute the interstage buffers needed for pipelined operation, as shown in Fig. 3.31

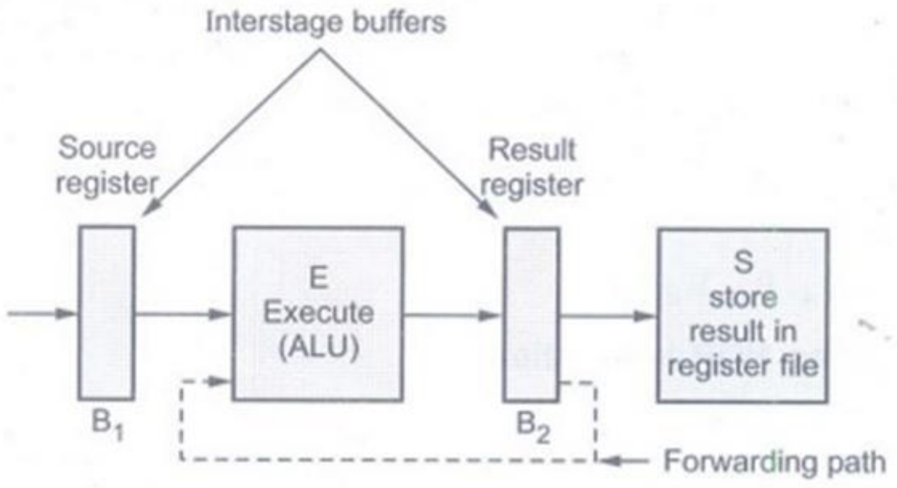


Fig 3.31 Operand forwarding in a pipelined processor – Forwarding path in the processor pipeline

The data forwarding mechanism is indicated by dashed lines.

The two multiplexers select the data for ALU either from destination bus or from source 1 and source 2 registers.

When the forwarding logic detects data dependency, it forwards ALU output available in the result register using data forwarding path to the ALU for the next operation. Hence the execution of next (dependent) instruction proceeds without interruption.

Fig. 3.32 shows the hardware necessary to support forwarding for operations that use results during the EX stage.

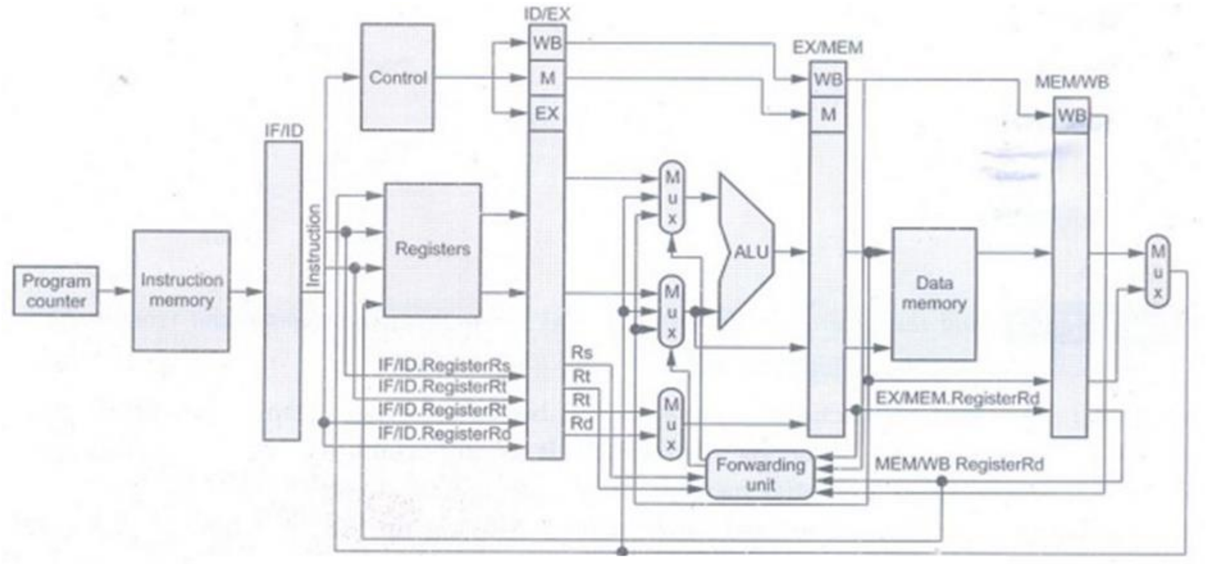


Fig 3.32 Modified datapath to resolve hazards via forwarding

Compared to data path shown in Fig. 3.32, the multiplexers are added to provide inputs to the ALU along with the forwarding unit.

3.7.2. Handling Data Hazards in Software

In this approach, software (compiler) detects the data dependencies. If data dependencies are found it introduces necessary delay between two instructions by inserting NOP (no-operation) instructions as follows :

I_1 :MUL R_2, R_3, R_4

NOP

NOP

ADD R_4, R_5, R_6

Disadvantages of adding NOP instructions

It leads to larger code size.

A given processor may have several hardware implementations. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and hence would lead to reduce performance on a different implementation.

To achieve better performance, the compilers are designed such that they can reorder instructions to perform useful task in the NOP slots.

Example 3.7.1 Convert the following code segment in C to MIPS instructions, assuming all variables are in memory and are addressable as offsets from $t0:

a = b + e;

c = b+f;

Solution :

lw $t1, 0($t0)

lw $t2, 4($t0)

add $t3, $t1,$t2

sw $t3, 12($t0)

lw $t4, 8($10)

add $t5, $t1,$t4

sw $t5, 16($t0)

Example 3.7.2 Find the hazards in the code segment of the previous example 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. It is important to note that bypassing eliminates several other potential hazards, including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction to become the third instruction eliminates both hazards:

lw $t1, 0($t0)

lw $t2, 4($10)

lw $t4, 8($t0)

add $t3, $t1,$t2

sw $t3, 12($t0)

add $t5, $t1,$t4

sw $t5, 16($t0)

Side Effects

When destination register of the current instruction is the source register of the next instruction there is a data dependency. Such data dependency is explicit and it is identified by register name. There are some instruction that change the contents of a register other than the one named as the destination. For example, instruction (stack instructions: push or pop) that uses an autoincrement or autodecrement addressing mode.

In autoincrement or autodecrement addressing mode, the instruction changes the contents of a source register used to access one of its operands. In such cases, we need to check data dependencies for registers affected by an autoincrement or autodecrement operation along with the destination register.

When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Let us assume that, $R_1$and R_2 holds a double-precision integer number and R_3 and R_4 holds another double-precision integer number.

The addition of these two numbers may be accomplished as follows:

ADD R_1, R_3

ADD with Carry R_2, R_4

Even though register names are different, the dependency (implicit) exists between these two instructions through the carry flag. This flag is Set/Reset by the first instruction and used in the second instruction, which performs the operation

$R_4 \leftarrow [R_2]+[R_4]+\text{Carry}$

Important Note :

Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have less side effects.

Example 3.7.3 The following sequence of instructions are executed in the basic 5-stage pipelined processor.

or r1, r2, r3

or r2, r1, r4

or r1, r1, r2

a) Indicate dependences and their type.

b) Assume there is no forwarding in this pipelined processor. Indicate hazards and add NOP instructions to eliminate them.

c) Assume there is full forwarding. Indicate hazards and add NOP instructions to eliminate them.

Solution: a) Dependences and their type

• Read After Write (RAW) dependency in r1 between Instructions 1, 2 and 3.

• Read After Write (RAW) dependency in r2 between Instructions 2 and 3.

• Write After Read (WAR) in r2 from Instruction 1 to 2.

• Write After Read (WAR) in r1 from Instruction 2 to 3.

• Write After Read (WAR) in r1 from Instruction 1 to 3.

b) No hazards from write after read and write after write, since there are 5 stages. Read after writes cause data hazards.

or r1, r2, r3

NOP

NOP

or r2, r1, r4

NOP

NOP

or r1, r1, r2

c) In full forwarding the data hazards above are eliminated, thus there is no need for NOP instructions.

3.8. Handling Control Hazards

3.8.1 Instruction Queue and Prefetching

To reduce the effect of cache miss or branch penalty, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. This is illustrated in Fig. 3.33

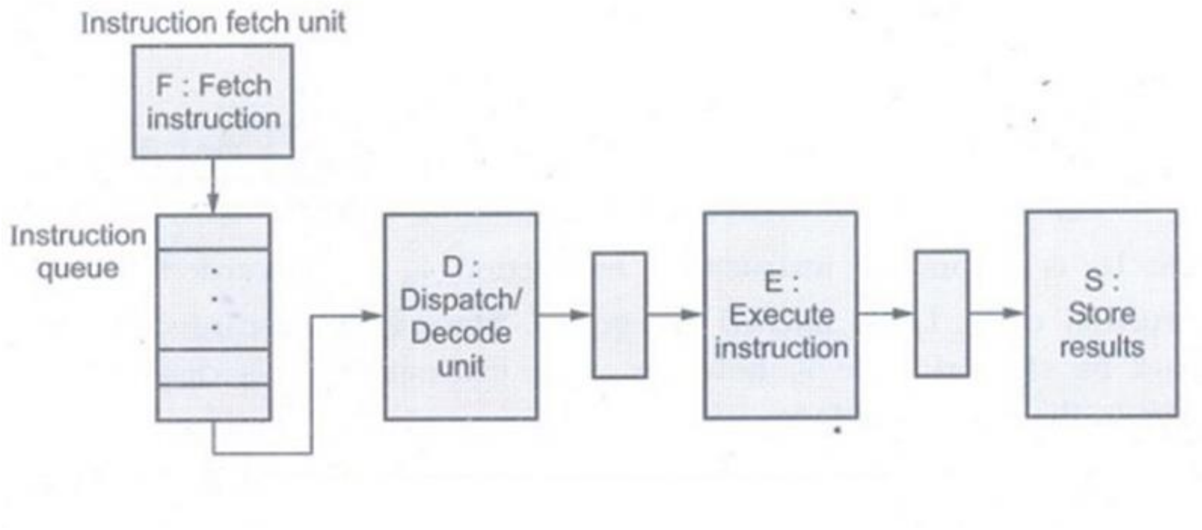


Fig 3.33 Use of instruction queue

A separate unit called dispatch unit takes instructions from the front of the queue and sends them to execution unit. It also performs the decoding function.

The fetch unit attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions during cache miss.

In case of data hazard, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue.

3.8.2. Use of instruction queue during branch instruction

Fig. 3.34 shows instruction time line. It also shows how the queue length changes over the clock cycles. Every fetch operation adds one instruction to the queue and every dispatch unit operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles.

The instruction I has 2-cycle stall. In these two cycles fetch unit adds two instructions but dispatch unit does not issue any instruction. Due to this, the queue length rises to 3 in clock cycle 6.

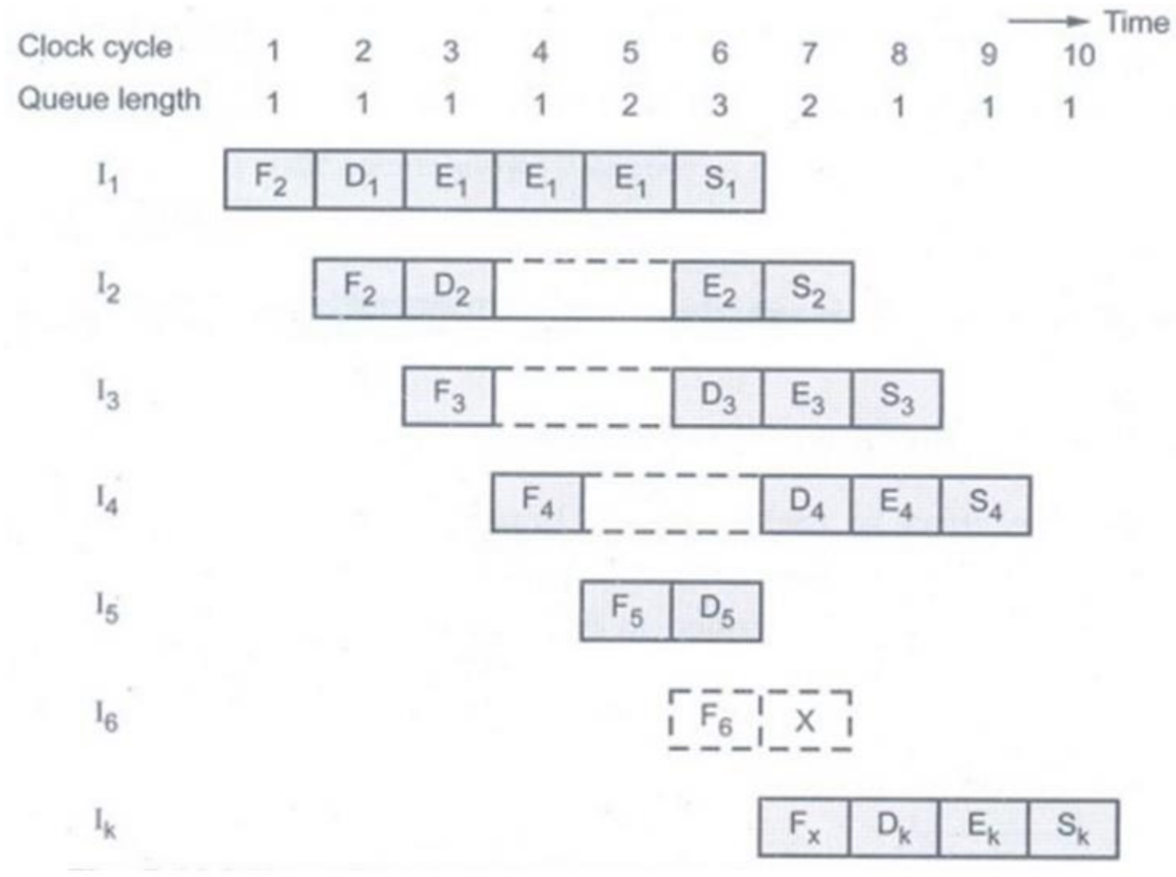

Time
Clock cycle 1 2 3 4 5 6 7 8 9 10
Queue length 1 1 1 1 2 3 2 1 1 1
I_1 F_2 D_1 E_1 E_1 E_1 S_1
I_2 F_2 D_2 E_2 S_2
I_3 F_3 D_3 E_3 S_3
I_4 F_4 D_4 E_4 S_4
I_5 F_5 D_5
I_6 F_6 X
I_k F_x D_k E_k S_k

Fig 3.34 Use of instruction queue during branch instruction

Since I_5 is a branch instruction, instruction I_6 is discarded and the target instruction of I_5, I_K is fetched in cycle 7. Since I_6 is discarded, normally there would be stall in cycle 7; however, here instruction I_4 is dispatched from the queue to the decoding stage.

After discarding I_6, the queue length drops to 1 in cycle 8. The queue length remains one until another stall is encountered. In this example, instructions I_1, I_2, I_3, I_4 and I_K complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time.

3.8.3 Branch Folding

The technique in which instruction fetch unit executes the branch instruction (by computing the branch address) concurrently with the execution of other instructions is called branch folding.

Branch folding occurs only if there exists at least one instruction in the queue other than the branch instruction, at the time a branch instruction is encountered.

In case of cache miss, the dispatch unit can send instructions for execution as long as the instruction queue is not empty. Thus, instruction queue also prevents the delay that may occur due to cache miss.

3.8.4 Approaches to Deal

The conditional branching is a major factor that affects the performance of instruction pipelining. There are several approaches to deal with conditional branching.

These are:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction.

3.8.5. Multiple streams

We know that, a simple pipeline suffers a penalty for a branch instruction. To avoid this, in this approach they use two streams to store the fetched instruction. One stream stores instructions after the conditional branch instruction and it is used when branch condition is not valid.

The other stream stores the instruction from the branch address and it is used when branch condition is valid.

Drawbacks :

Due to multiple streams there are contention delays for access to registers and to memory.

Each branch instruction needs additional stream.

3.8.6. Prefetch branch target

In this approach, when a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This already fetched target is used when branch is valid.

3.8.6.1. Loop buffer

A loop buffer is a small very high speed memory. It is used to store recently prefetched instructions in sequence. If conditional branch is valid, the hardware first checks whether the branch target is within the buffer. If so, the next instructions are fetched from the buffer, instead of memory avoiding memory access.

Advantages:

In case of conditional branch, instructions are fetched from the buffer saving memory access time. This is useful for loop sequences.

If a branch occurs to a target just a few locations ahead of the address of the branch instruction, we may find target in the buffer. This is useful for IF-THEN-ELSE-ENDIF sequences.

3.8.6.2. Delayed branch

The location following a branch instruction is called a branch delay slot. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction.

There are three ways to fill the delay slot :

1. The delay slot is filled with an independent instruction before branch. In this case performance always improves.

2. The delay slot is filled from the target branch instructions. Performance improves if only branch is taken.

3. The delay slot is filled with an instruction which is one of the fall through instruction. In this case performance improves if branch is not taken.

These above techniques are called delayed branching.

3.8.7. Branch Prediction

Prediction techniques can be used to check whether a branch will be valid or not valid. These techniques reduce the branch penalty.

• The common prediction techniques are:

• Predict never taken

•Predict always taken

• Predict by opcode

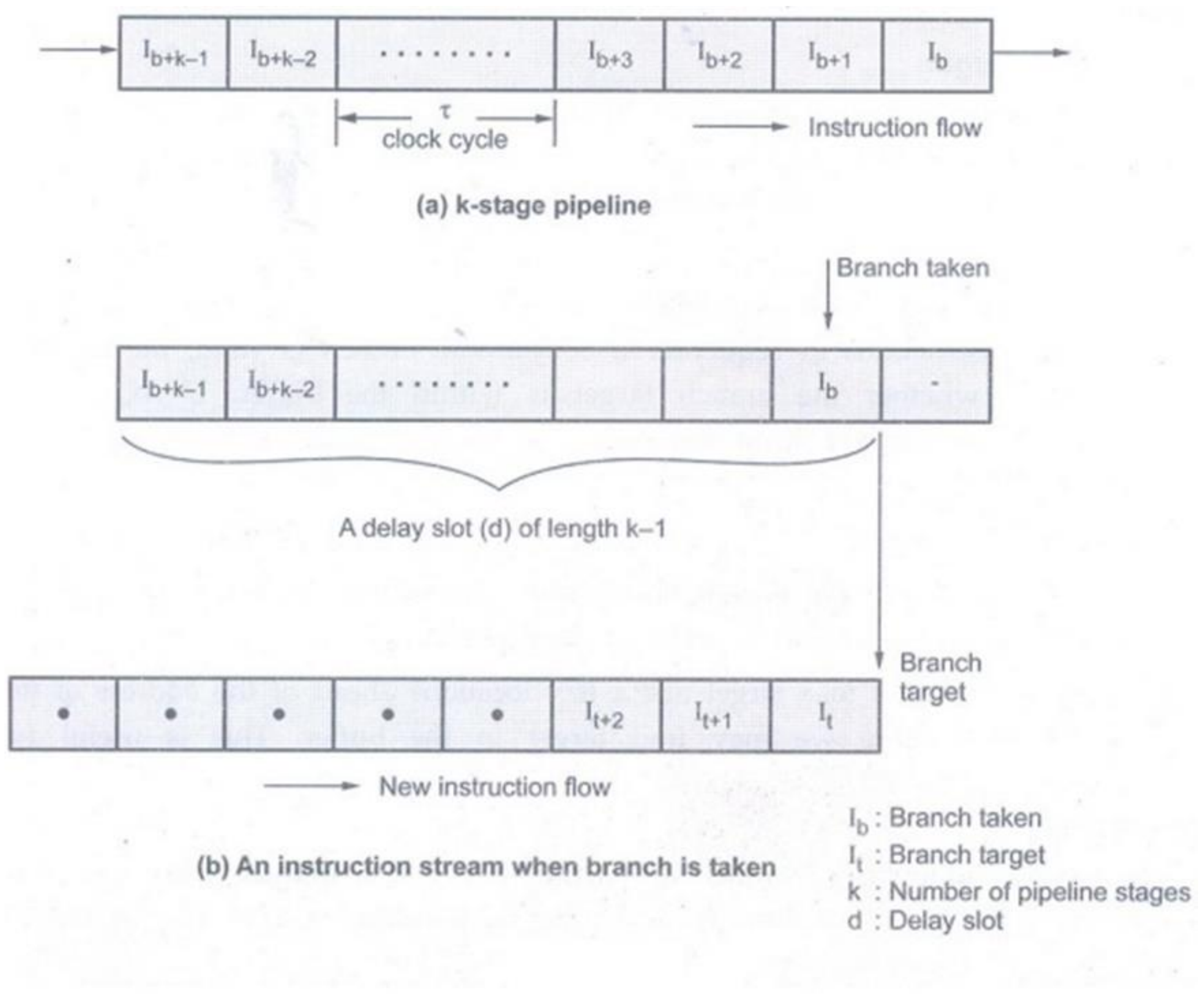


Fig 3.35 Branch Prediction

• Taken/Not taken switch

• Branch history table

In the first two approaches if prediction is wrong a page fault or protection violation error occurs. The processor then halts prefetching and fetches the instruction from the desired address.

In the third prediction technique, the prediction decision is based on the opcode of the branch instruction. The processor assumes that the branch will be taken from certain branch opcodes and not for others.

The fourth and fifth prediction techniques are dynamic; they depend on the execution history of the previously executed conditional branch instruction.

3.8.8. Branch Prediction Strategies

There are two types of branch prediction strategies :

• Static branch strategy

• Dynamic branch strategy.

3.8.8.1. Static Branch Strategy: In this strategy branch can be predicted based on branch code types statically. This means that the probability of branch with respect to a particular branch instruction type is used to predict the branch. This branch strategy may not produce accurate results every time.

3.8.8.2. Dynamic Branch Strategy: This strategy uses recent branch history during program execution to predict whether or not the branch will be taken next time when it occurs. It uses recent branch information to predict the next branch. The recent branch information includes branch prediction statistics such as:

T: Branch taken

N: Not taken

NN: Last two branches not taken

NT: Not branch taken and previous takes

TT: Both last two branch taken

TN: Last branch taken and previous not taken

• The recent branch information is stored in the buffer called Branch Target Buffer (BTB).

• Along with above information branch target buffer also stores the address of branch target.

Fig. 3.36 shows the organization of branch target buffer.

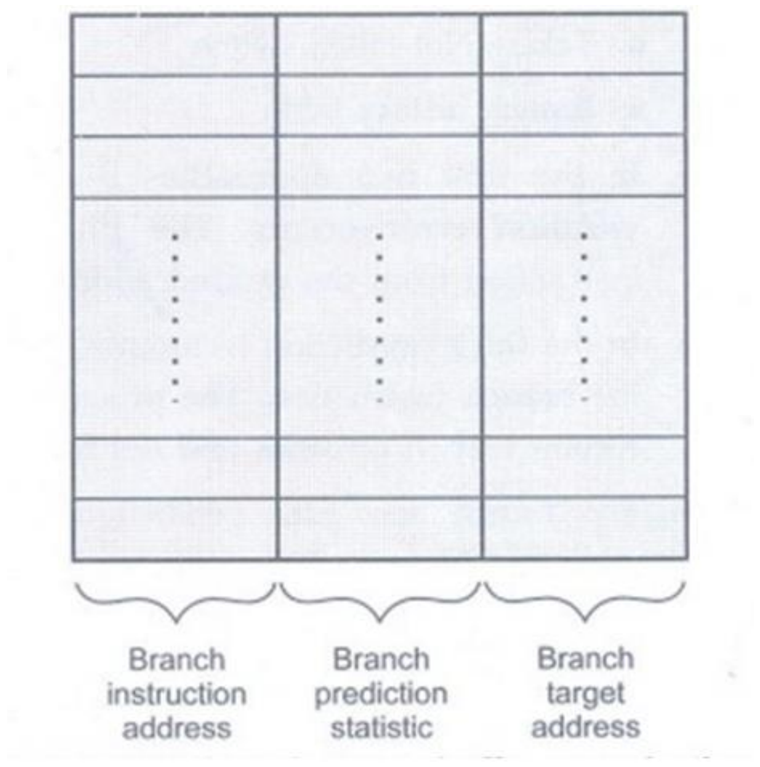


Fig 3.36 Branch target buffer organization

Fig. 3.37 shows a typical state diagram used in dynamic branch prediction.

This state diagram allows backtracking of last two instructions in a given program. The branch target buffer entry contains the backtracking information which guides the prediction.

The prediction information is updated upon completion of the current branch.

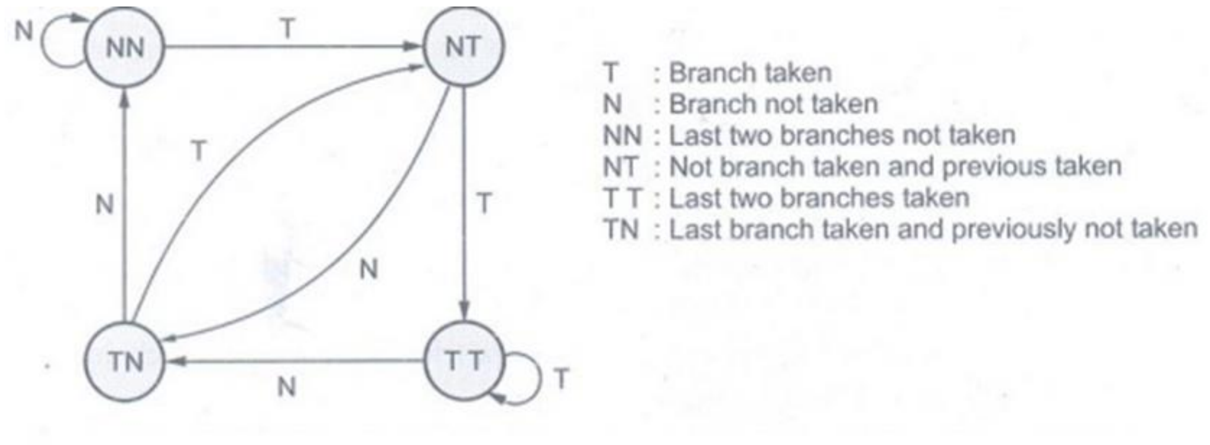

N
NN
T
NT
T
N
T
N
TN
N
TT
T
T : Branch taken
N : Branch not taken
NN : Last two branches not taken
NT : Not branch taken and previous taken
T T : Last two branches taken
TN : Last branch taken and previously not taken

Fig 3.37 A typical state diagram used in dynamic branch prediction

•To make branch overhead zero, the branch target buffer is extended to store the target instruction itself and a few of its successor instructions. This allows processing of conditional branches with zero delay.

Example 3.8.1 The following sequence of instructions are executed in the basic 5-stage pipelined processor :

1w$1, 40($6)

add $6, $2, $2

sw $6, 50($1)

Indicate dependence and their type. Assuming there is no forwarding in this pipelined processor, indicate hazards and add NOP instructions to eliminate them.

Solution: a) I1: 1W $1, 40($6): Raw dependency on $1 from I1 to I3

I2: add $6, $2, $2 : Raw dependency on $6 from I2 to I3

I3: SW $6 ($1): WAR dependency on $6 from I1 to I2 and I3

b) In the basic five-stage pipeline WAR dependency does not cause any hazards. Assuming there is no forwarding in this pipelined processor RAW dependencies cause hazards if register read happens in the second half of the clock cycle and the register write happens in the first half. The code that eliminates these hazards by inserting nop instruction is:

1w $1, 40($6)

add $6, $2, $2

nop; delay 13 to avoid RAW hazard on $1 from I1'

SW $6, 50($1)

Example 3.8.2 A processor has five individual stages, namely, IF, ID, EX, MEM, and WB and their latencies are 250 ps, 350 ps, 150 ps, 300 ps, and 200 ps respectively. The frequency of the instructions executed by the processor are as follows; ALU: 40%, Branch 25 %, load: 20% and store 15% What is the clock cycle time in a pipelined and non-pipelined processor? If you can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor? Assuming there are no stalls or hazards, what is the utilization of the data memory? Assuming there are no stalls or hazards, what is the utilization of the write register port of the "Registers" unit?

Solution: a) Clock cycle time in a pipelined processor=350 ps

Clock cycle time in non-pipelined processor

= 250 ps +350 ps + 150 ps + 300 ps + 200 ps= 1250 ps

b) We have to split one stage of the pipelined datapath which has a maximum latency i.e. ID.

After splitting ID stage with latencies ID1 = 175 ps and ID2 = 175 ps we have new

clock cycle time of the processor equal to 300 ps.

c) Assuming there are no stalls or hazards, the utilization of the data memory = 20% to 15 % = 35%.

d) Assuming there are no stalls or hazards, the utilization of the write-register port

of the register unit = 40 % + 25 %=65%