SVCE College of Engineering

Department of Information Technology

CS6303 COMPUTER ARCHITECTURE

Question Bank

UNIT - III

PROCESSOR AND CONTROL UNIT

Year: II year (2015-16) ODD SEMESTER

PART A

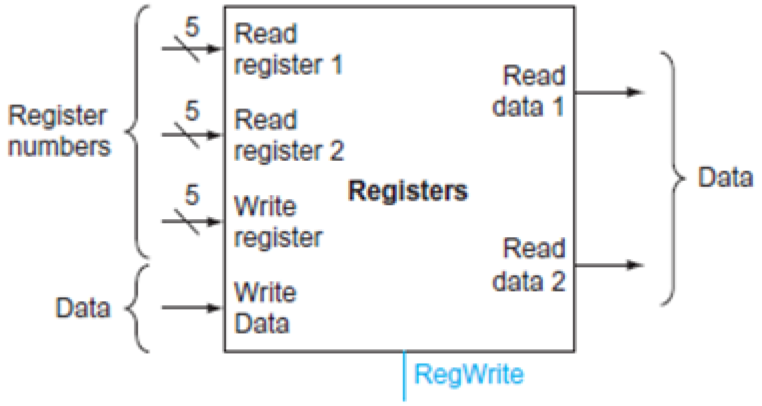
1. **What is combinational and state element?**

   Combinational element: An operational element, such as an AND gate or an ALU.
   State element: A memory element, such as a register or a memory.

2. **What is meant by data path element?**
   A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

3. **What is the use of PC register?**
   PC is the register containing the address of the instruction in the program being executed.

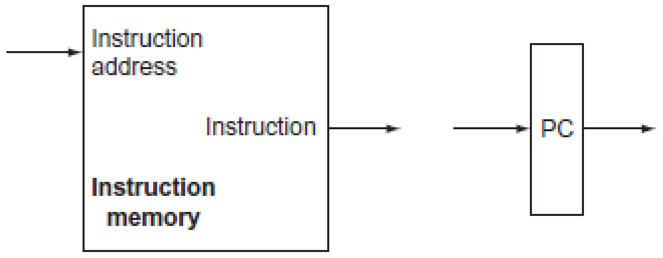
4. **What is meant by register file?**
   Register file is a state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.



5. **What are the two state elements needed to store and access an instruction?**

   Two state elements needed are:
   1. Instruction memory

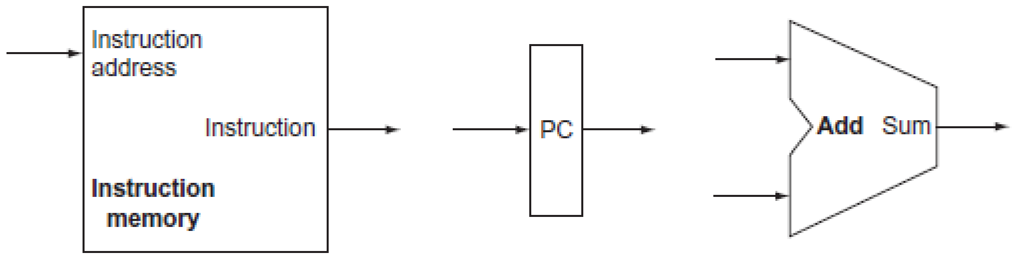
2. Program counter



a. Instruction memory b. Program counter

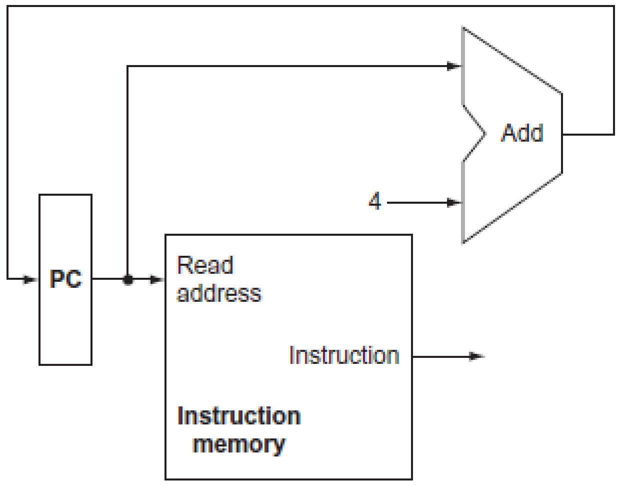
6. **What are the elements used to fetch instructions and increment the PC?**
   3 elements are used to fetch instructions and increment PC:

- Instruction memory
  - A state element where instruction is stored.
- PC – program counter
  - A state element or register containing the address of the next instruction to be executed.
- Adder
  - An ALU wired to always add its two 32-bit inputs and place the sum on its output.



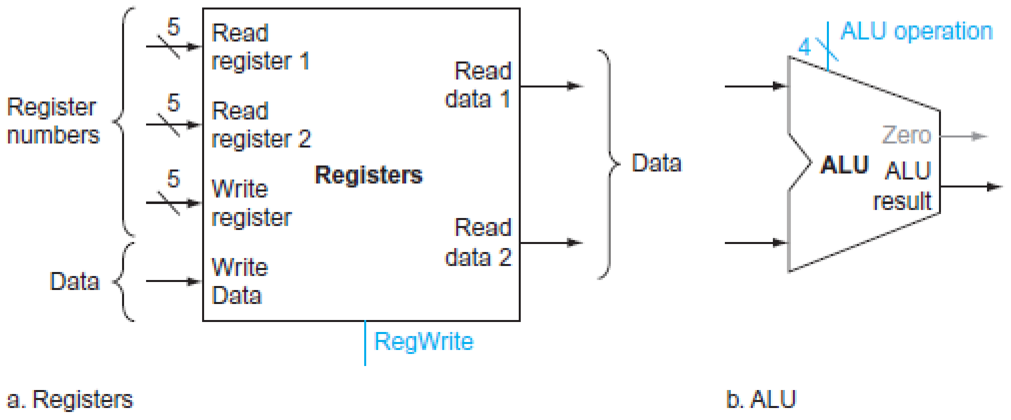
a. Instruction memory b. Program counter c. Adder

7. **Draw the diagram of portion of datapath used for fetching instruction.**



8. **What are the two elements needed to implement the R-format ALU operations?**
Two elements needed to implement R-format ALU operations are:

- Register file
  - A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.
- ALU
  - A combinational element that performs arithmetic and logical operations on the input data.



9. **Define – Sign Extend.**
To increase the size of a data item by replicating the high-order sign bit of the original data item in the high order bits of the larger, destination data item.

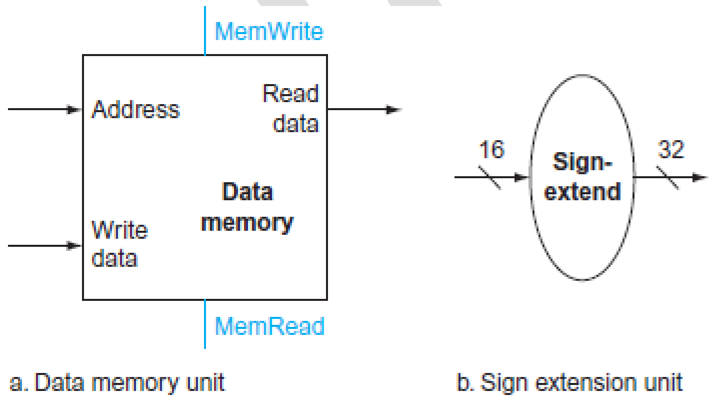
10. **What is meant by branch target address?**
The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the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

11. **Differentiate branch taken from branch not taken.**

branch taken - A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional jumps are taken branches.
branch not taken or (untaken branch) - A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

12. What are the units that are needed to implement load and store.
4 elements are used in Load/Store instructions

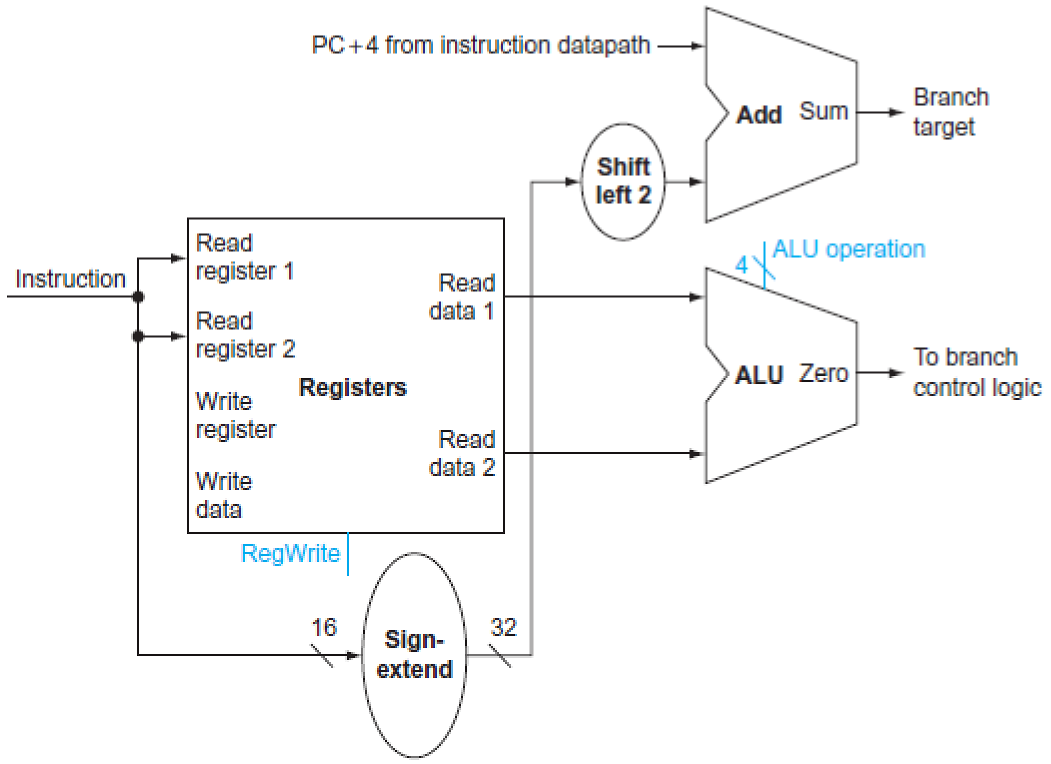
- Data memory
  - The memory unit is a state element with inputs for the address and the write data, and a single output for the read result
- Sign Extend
  - a 16-bit input that is sign-extended into a 32-bit result appearing on the output
- Register file
  - A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.
- ALU
  - A combinational element that performs arithmetic and logical operations on the input data.



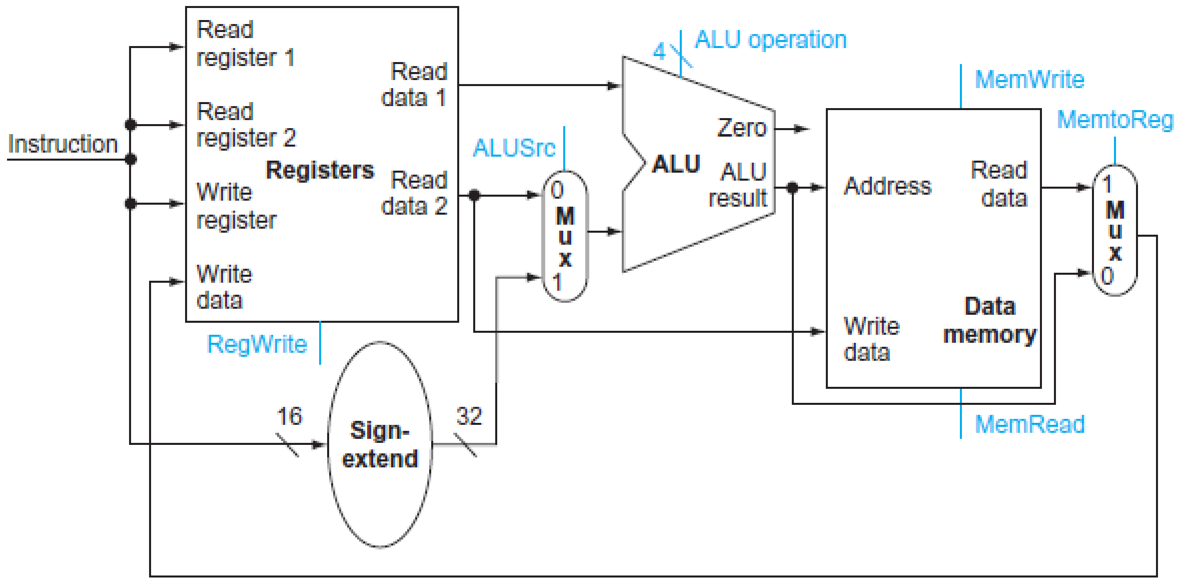
13. What is meant by delayed branch?
A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

14. Draw the datapath for branch instruction.

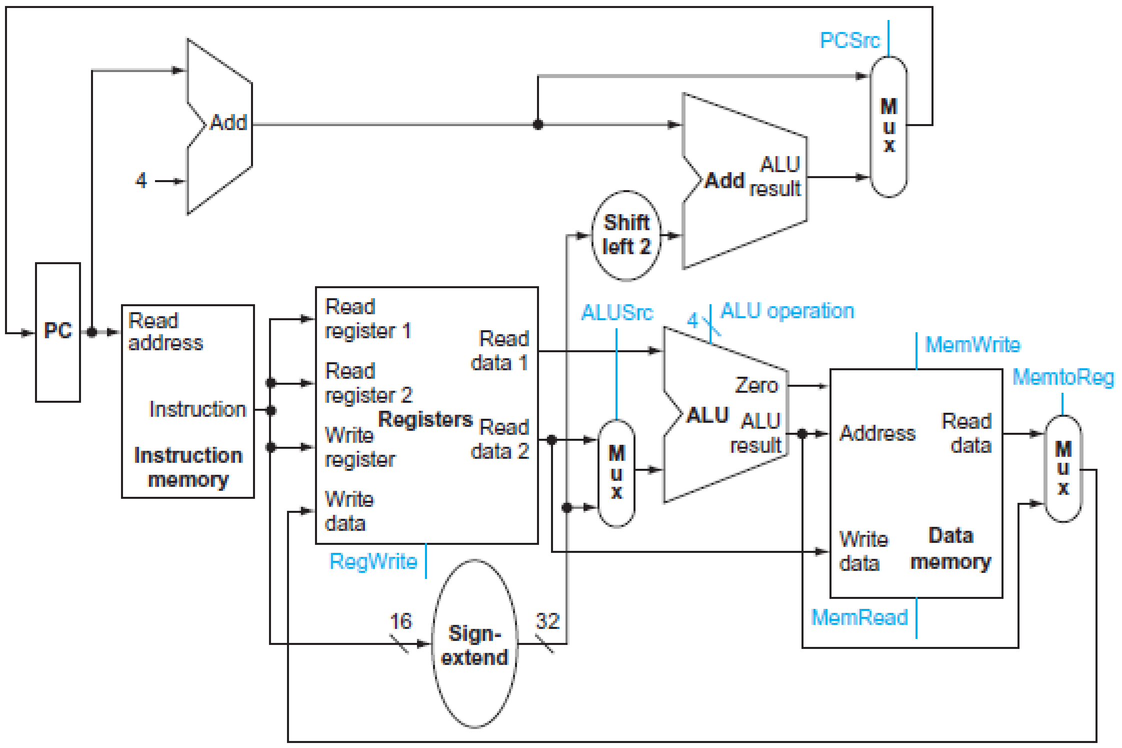
The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.



15. Draw the combined datapath for memory and R-type instructions.



16. Draw the combined datapath for instruction fetch, memory, R-type and branch instructions.



17. Draw the truth table for the 4 ALU control bits from ALUOp bits and the different function codes.

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

18. List the seven control signals with its effect when asserted or deasserted.

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

19. How are the control signals set for each type of MIPS instructions?

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

20. What are the three instruction classes and their instruction formats with bit positions?

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

a. R-type instruction

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

b. Load or store instruction

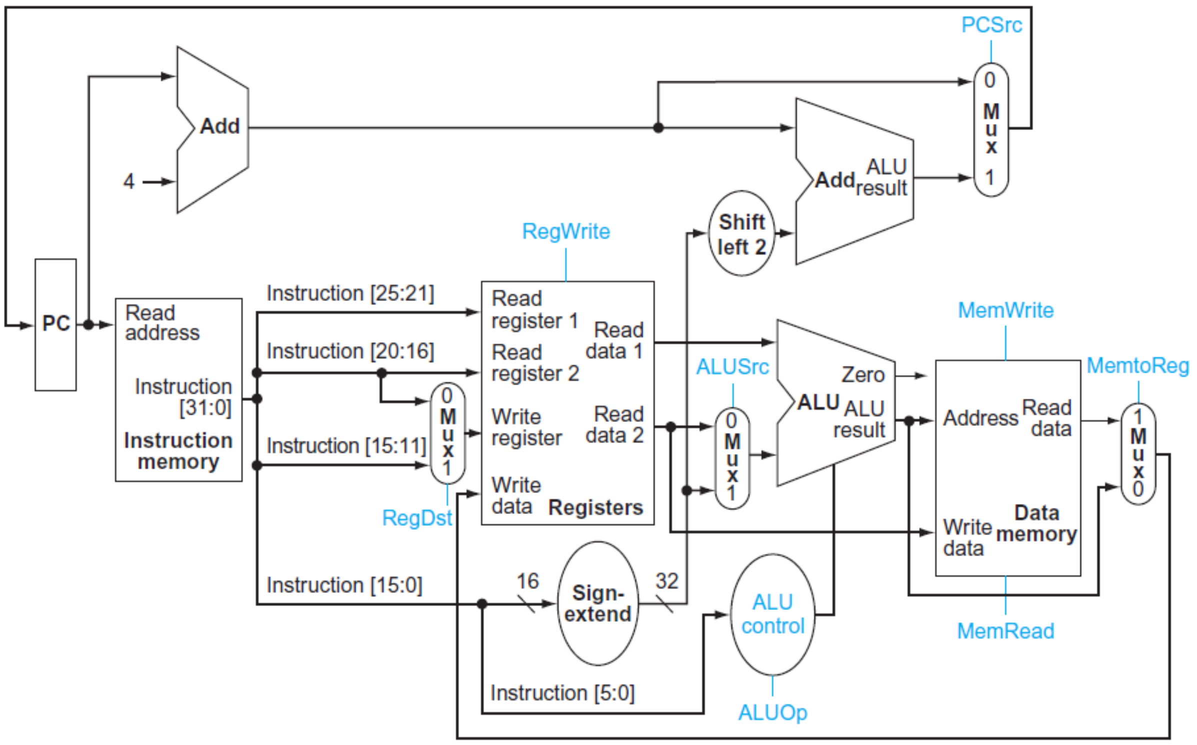
Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

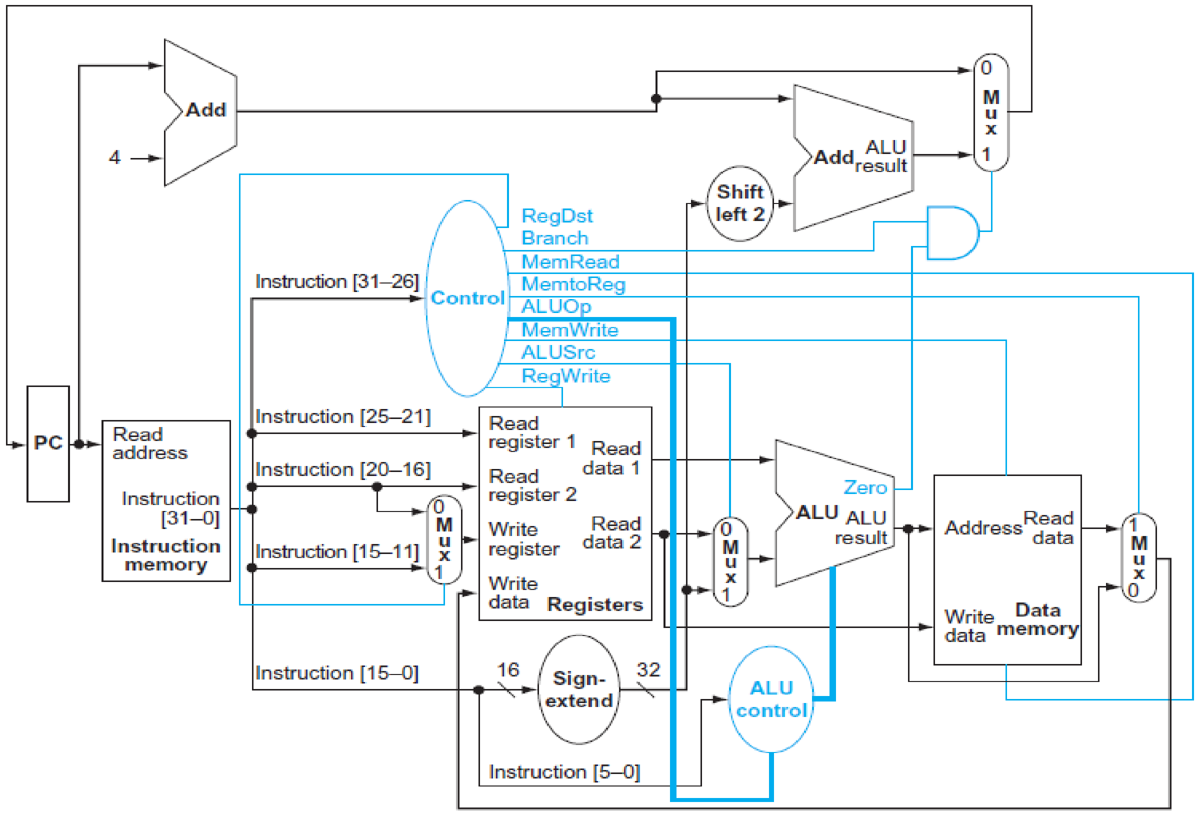
21. Draw the instruction format for jump instruction with its bit positions.

	OP	
Field	000010	address
Bit positions	31:26	25:0

22. Draw the datapath with control lines.



23. Draw the datapath with main control unit and ALU control unit.



24. Why no single clock cycle?

Assuming fixed-period clock every instruction datapath uses one clock cycle implies:

a. CPI = 1

b. cycle time determined by length of the longest instruction path (load)

   i. but several instructions could run in a shorter clock cycle: *waste of time*

   ii. consider if we have more complicated instructions like floating point!

c. resources used more than once in the same cycle need to be duplicated

   i. *waste of hardware and chip area*

25. What is meant by pipelining?

Pipelining is an implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.

26. Write the formula for calculating time between instructions in a pipelined processor.

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

27. What are the five steps in MIPS instruction execution?

MIPS instructions classically take five steps:

1. IF - Fetch instruction from memory.

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

28. What are pipeline hazards? List its types.

Pipeline hazards are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.

1. Structural hazards
2. Control hazards
3. Data hazards

29. Explain structural hazard with an example?

Structural hazard When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
Structural hazard occurs due to *inadequate hardware.*
Example: Use of single memory in execution of MIPS instruction. (i.e.) No separate memory for instruction and data.

30. Explain data hazard with an example?

Data hazard is also called as **pipeline data hazard**. It is when a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
Data hazard occurs due to data dependency (when output of one instruction is the input of next instruction.)
Example:

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

Here output of add instruction is the input of the next sub instruction. So there is a data dependency between add and sub instruction. Thereby we can say that there is a data hazard.

31. Explain control hazard with an example?

Control hazard also called as branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
Example:

```
      beq $s0, $s1, exit -> control hazard
      and $s2, $s3,$s4
exit: or $s2, $s3, $s4
```

32. What is meant by forwarding?

Forwarding also called as **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmers visible registers or memory.

33. What is meant by load-use data hazard?
Load-use data hazard - A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

34. What is pipeline stall?
Pipeline stall is also called **bubble**. A stall initiated in order to resolve a hazard.

35. What is meant by branch prediction?
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

36. What are the 5 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

37. What is pipeline register?
Pipeline register is the intermediate register between two 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. 4 pipeline registers are IF/ID, ID/EX, EX/MEM and MEM/WB.

38. List the control lines corresponding to each control line group?
Execution/address calculation stage control lines – RegDst, ALUOp1, ALUOp0, ALUSrc
Memory access stage control lines – Branch, Mem-Read, Mem-Write
Write back stage control lines – Reg-Write, Memto-Reg

39. What are exceptions and interrupts?
Exception – Also called as interrupt. It is an unscheduled event that disrupts program execution; used to detect overflow.
Interrupt - An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

40. What is vectored interrupts?
Vectored interrupt is an interrupt for which the address to which control is transferred is determined by the cause of the exception.

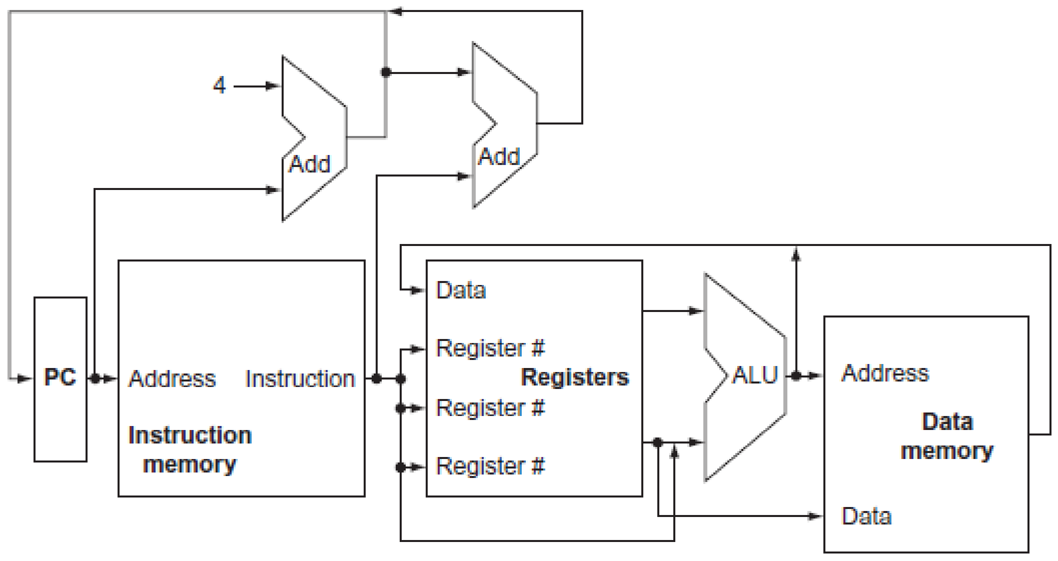
PART B

1. **Explain the basic MIPS implementation of instruction set. (10marks)**

For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction.

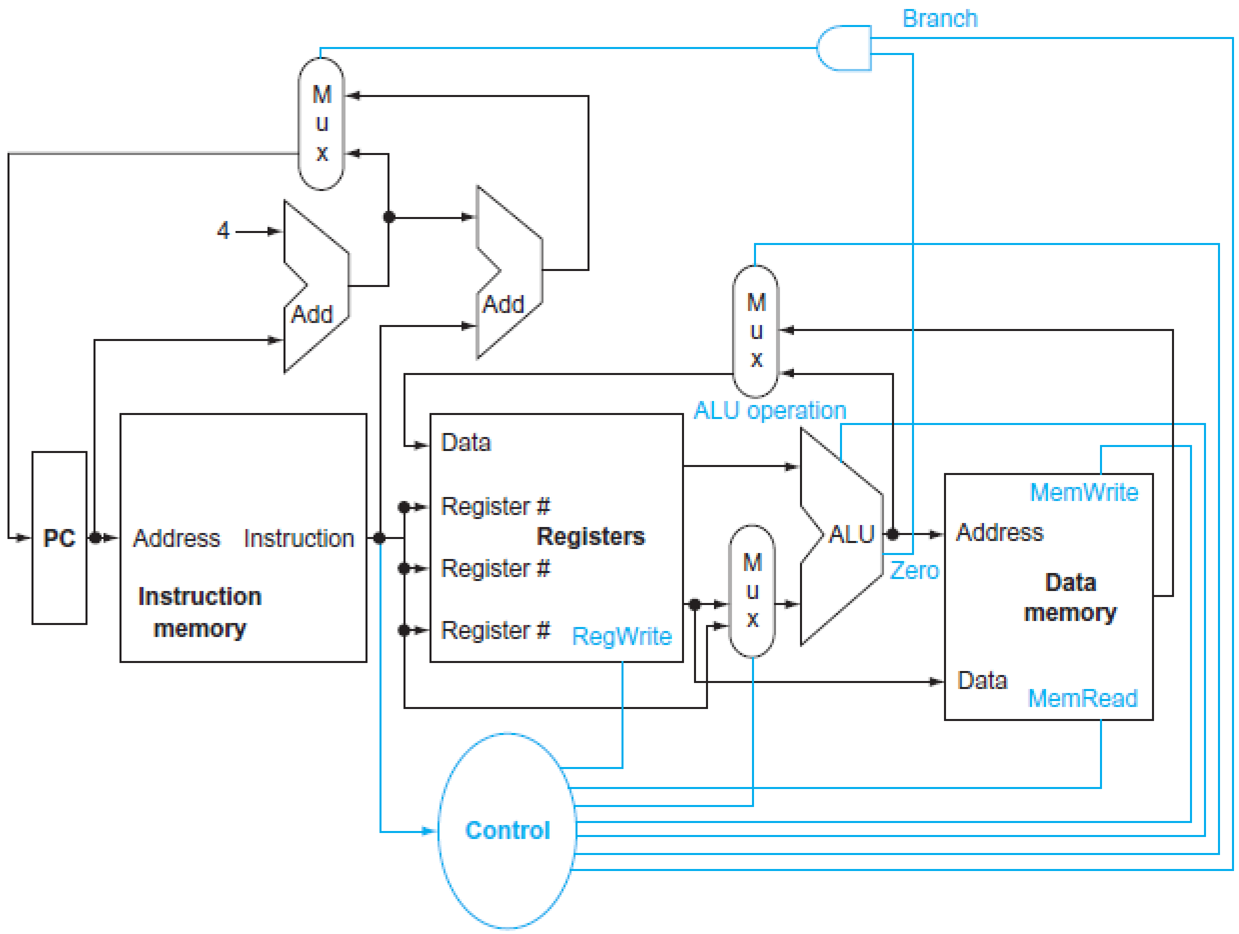


The above figure shows an abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register fi le. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch off set are summed) or from an adder that increments the current PC by 4.

A logic element that chooses from among the multiple sources and steers one of those sources to its destination is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*.

2. **Explain the basic MIPS implementation with necessary multiplexers and control lines (8 marks)**



The above figure shows the basic implementation of the MIPS subset, including the necessary multiplexors and control lines.

The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.

A *control unit*, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors. The third multiplexor, which determines whether PC + 4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a beq instruction. The regularity and simplicity of the MIPS instruction set means that a simple decoding process can be used to determine how to set the control lines.

3. What is data hazard? How do you overcome it? What are its side effects? (8marks)

Data hazard is also called as **pipeline data hazard**. It is when a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
Data hazard occurs due to data dependency (when output of one instruction is the input of next instruction.)

Example:

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

Here output of add instruction is the input of the next sub instruction. So there is a data dependency between add and sub instruction. Thereby we can say that there is a data hazard.

Hardware Solution:

- Forwarding
- Stalling with Forwarding

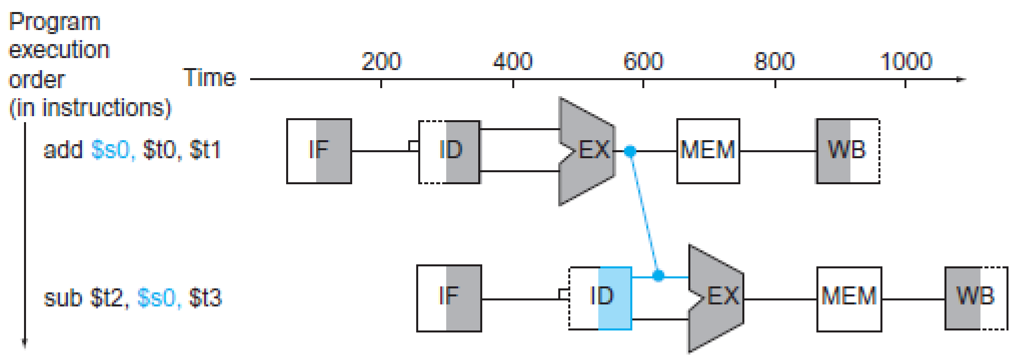
Software solution:

- Code reordering

1. Forwarding:

Forwarding also called as **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmers visible registers or memory.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.



Th e connection shows the forwarding path

from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register
$s0 read in the second stage of sub.

2. Forwarding with stalling:

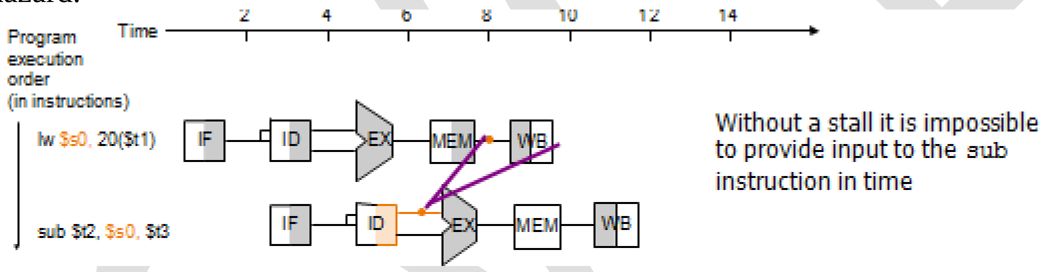
Load-use data hazard - A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

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

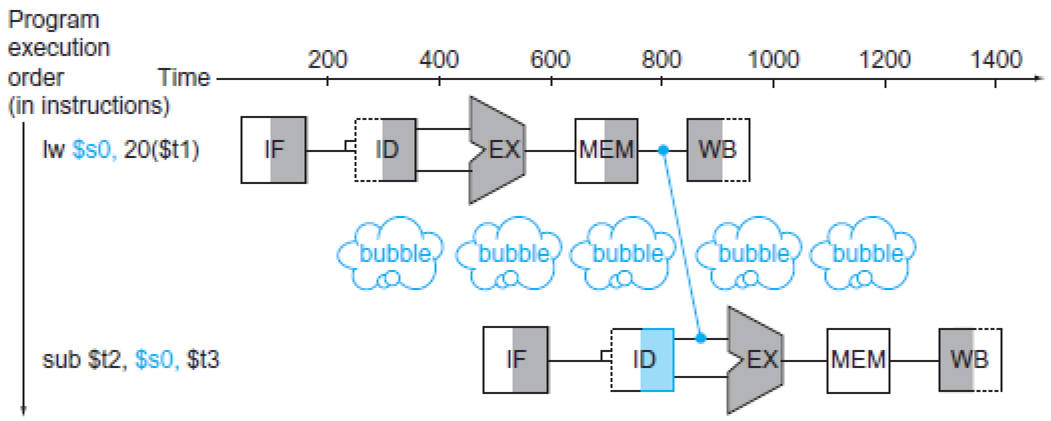
Forwarding may not be enough - if an R-type instruction following a load uses the result of the load – called *load-use data hazard*

Only forwarding:

If we use only forwarding in load-use data hazard then the forwarding line goes back in time which is not possible. So forwarding alone is not enough to solve this data hazard.


Program execution order (in instructions)
Time
2
4
6
8
10
12
14
lw $s0, 20($t1)
IF
ID
EX
MEM
WB
sub $t2, $s0, $t3
IF
ID
EX
MEM
WB
Without a stall it is impossible to provide input to the sub instruction in time

Forwarding with stalling:



We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible.

Software solution: Reordering Code to Avoid Pipeline Stall

Example:
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)

There is a data hazard between 2nd and 3rd instructions in the above example.

- **Reordered code:**

lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)

The data hazard is solved by reordering the code. The 3rd and 4th instruction has been interchanged to solve data hazard.

4. What are control hazards? Explain the methods for dealing with the control hazards. (16marks)

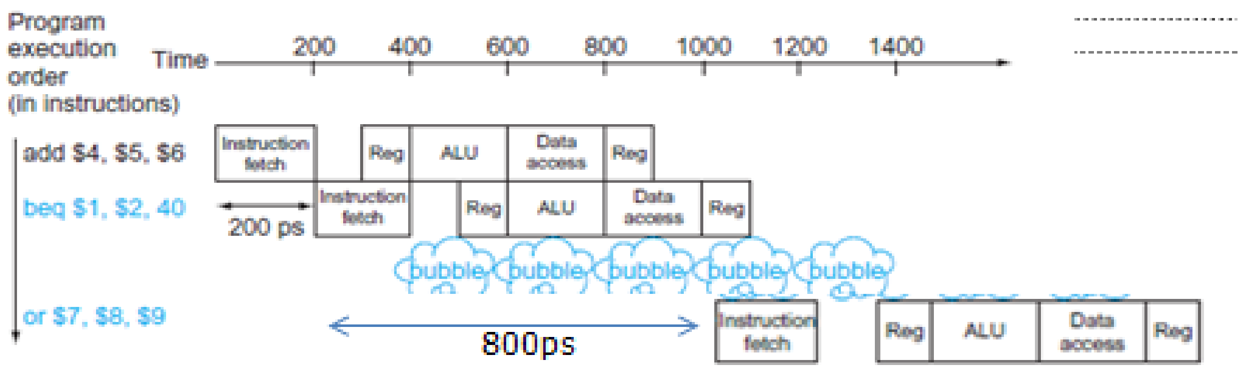
control hazard – Also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
We can identify there is a control hazard whenever there is a branch instruction in the given code sequence.

Solution:

- Stalling
- Branch Prediction – not taken
- Reduce branch delay
- Dynamic branch prediction

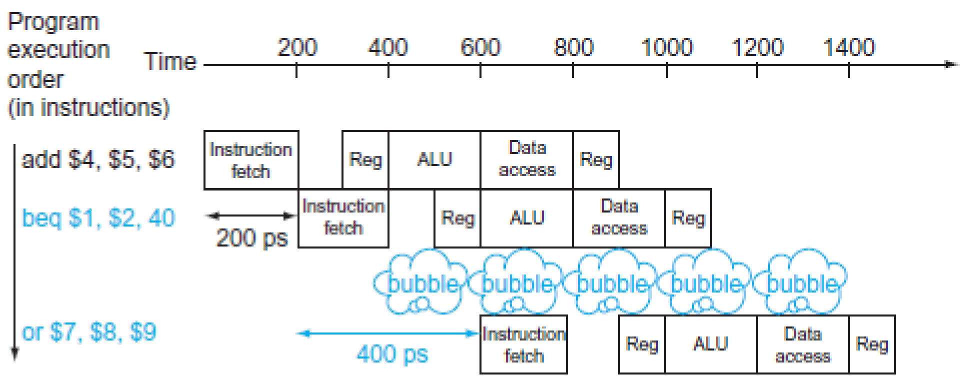
1. **Stalling:**

The decision task in a computer is the branch instruction. we must begin fetching the instruction following the branch on the very next clock cycle. Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory. One possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.
In the pipeline diagram, we assumed that the branch decision is available in the 4th stage MEM. So 3 bubbles have been inserted (i.e.) 3 clock cycles are wasted to solve the control hazard.



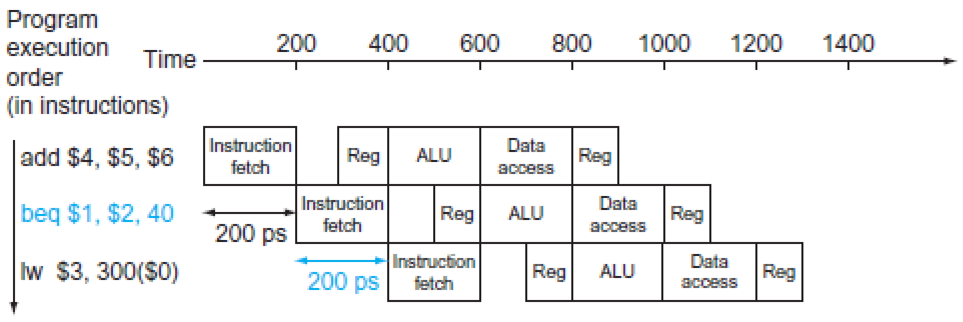
2. Reduce branch delay:

We can reduce the branch delay further by making the branch decision to be taken in the 2nd stage ID itself. Then we need to insert only one bubble (i.e.) one clock cycle is wasted to solve the branch hazard.



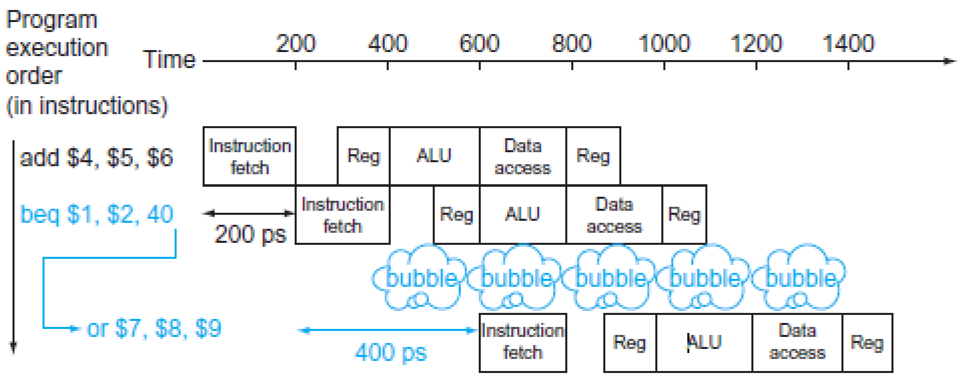
3. Branch prediction – untaken branch:
A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.
UnTaken :
Predict always that branches will be untaken. When you're right, the pipeline proceeds at full speed.

Prediction success:
If the prediction is a success, the pipeline proceeds as such without any change.



Prediction failure:

If the prediction is a failure, the branch not-taken instruction (lw as per the example given) is flushed and bubble is inserted. The branch taken instruction(or as per the example given) is fetched and starts to execute in the next clock cycle.



4. **Dynamic prediction**

Branch prediction is 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.

Hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.

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

Two schemes under dynamic prediction are:

- 1-bit and
- 2-bit scheme

1-bit scheme:

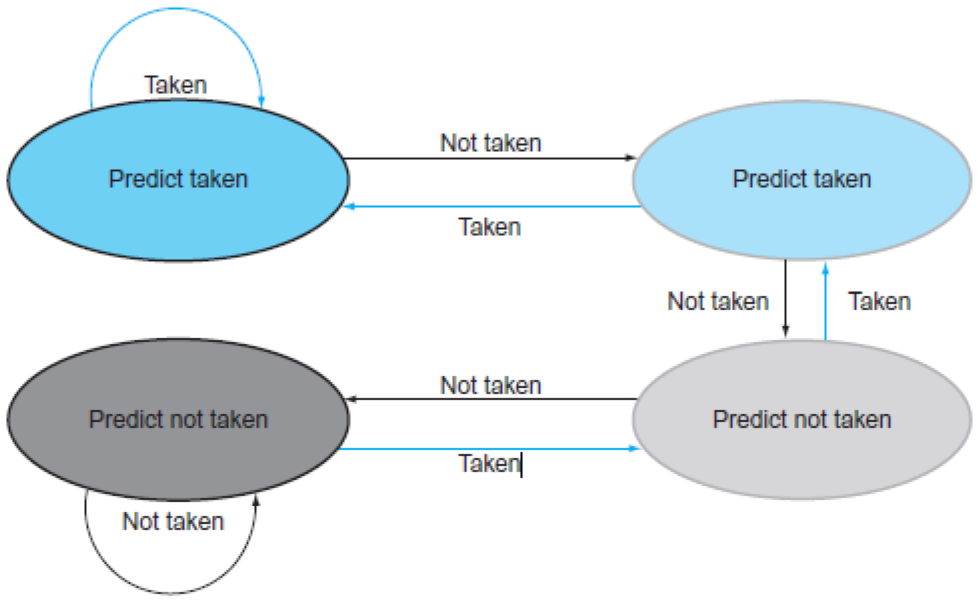
If the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.
This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

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

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

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, 2-bit prediction schemes are oft en used.

2-bit scheme:
In a 2-bit scheme, a prediction must be wrong twice before it is changed. The figure shows 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.

5. **Explain dynamic branch prediction. (8marks)**

Branch prediction is 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.

Hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.

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

Two schemes under dynamic prediction are:

- 1-bit and
- 2-bit scheme

1-bit scheme:

If the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.
This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

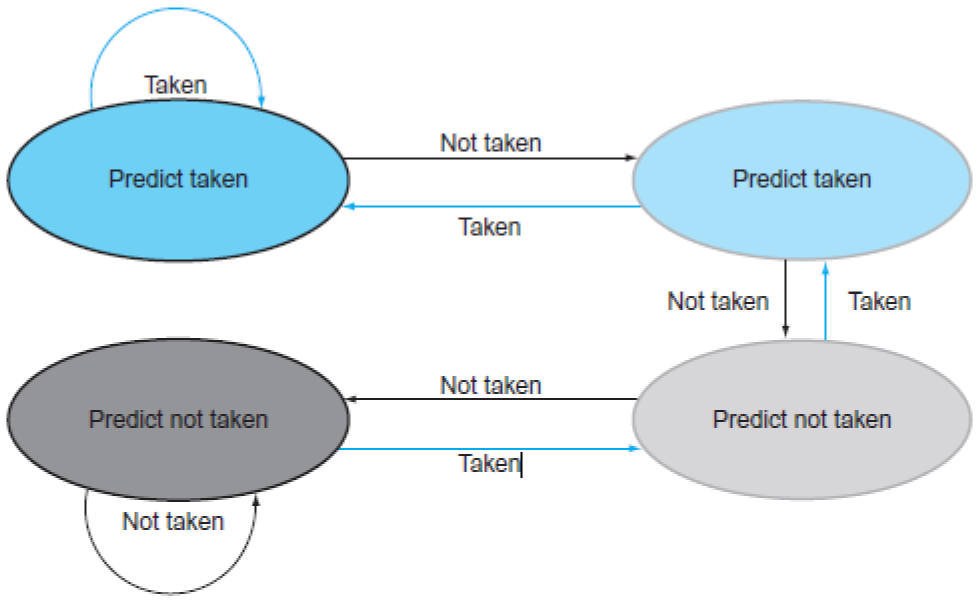
Example:

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

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

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, 2-bit prediction schemes are oft en used.

2-bit scheme:
In a 2-bit scheme, a prediction must be wrong twice before it is changed. The figure shows 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.

6. **What is pipelining hazard? List and explain the types of pipeline hazards and its solution with neat sketches. (16 marks)**

Pipelining:

Pipelining is an implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.

Pipeline hazard:

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

3 types of hazard:
1. **Structural hazard**
2. **Data hazard**
3. **Control hazard**

1. **Structural hazard:**

Structural hazard When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
Structural hazard occurs due to *inadequate hardware.*

Example: Use of single memory in execution of MIPS instruction. (i.e.) No separate memory for instruction and data.

2. **Data hazard:**

Data hazard is also called as **pipeline data hazard**. It is when a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. Data hazard occurs due to data dependency (when output of one instruction is the input of next instruction.)

Example:
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Here output of add instruction is the input of the next sub instruction. So there is a data dependency between add and sub instruction. Thereby we can say that there is a data hazard.

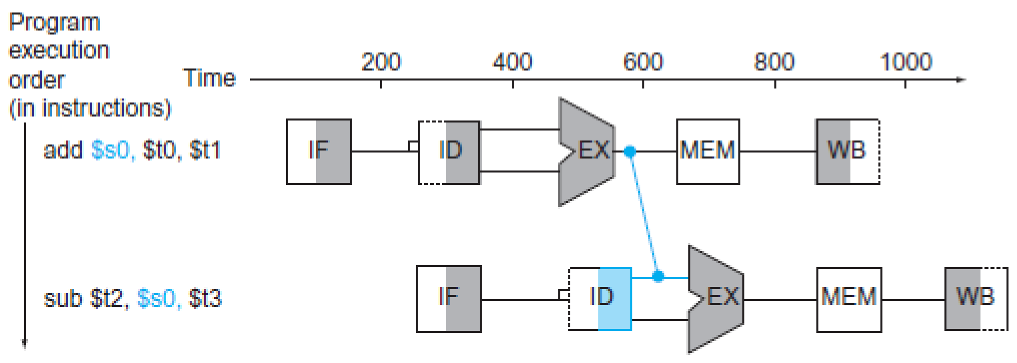
Hardware Solution:
- Forwarding
- Stalling with Forwarding

Software solution:
- Code reordering

2. **Forwarding:**

Forwarding also called as **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmers visible registers or memory.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.



Th e connection shows the forwarding path
from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register

$s0 read in the second stage of sub.

2. Forwarding with stalling:

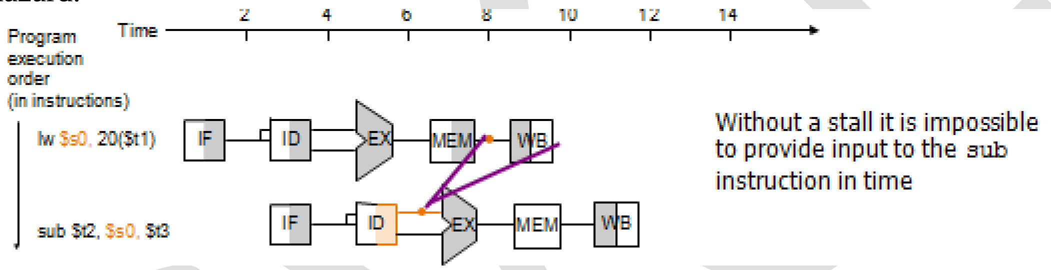
Load-use data hazard - A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

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

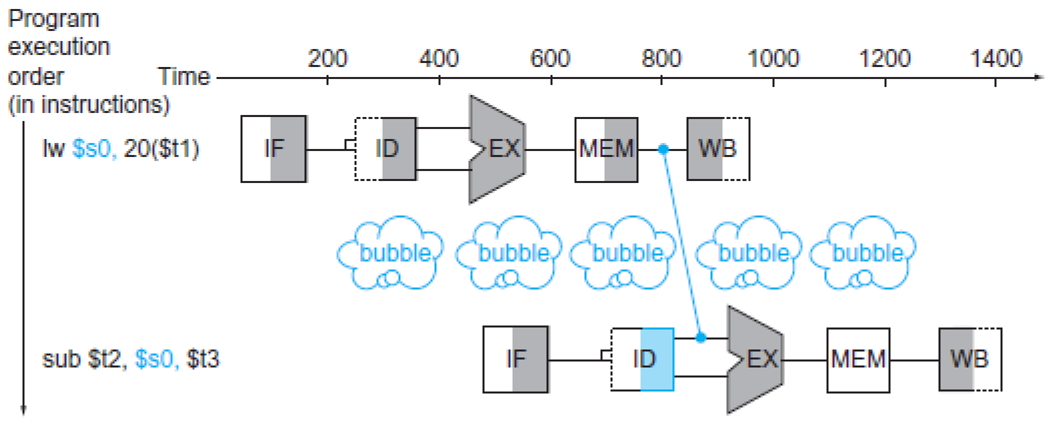
Forwarding may not be enough - if an R-type instruction following a load uses the result of the load – called *load-use data hazard*

Only forwarding:

If we use only forwarding in load-use data hazard then the forwarding line goes back in time which is not possible. So forwarding alone is not enough to solve this data hazard.


Program execution order (in instructions)
Time
2
4
6
8
10
12
14
lw $s0, 20($t1)
sub $t2, $s0, $t3
IF
ID
EX
MEM
WB
Without a stall it is impossible to provide input to the sub instruction in time

Forwarding with stalling:



We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible.

Software solution: Reordering Code to Avoid Pipeline Stall
Example:
lw $t0, 0($t1)

lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)

There is a data hazard between 2nd and 3rd instructions in the above example.

- **Reordered code:**

lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)

The data hazard is solved by reordering the code. The 3rd and 4th instruction has been interchanged to solve data hazard.

3. Control hazard

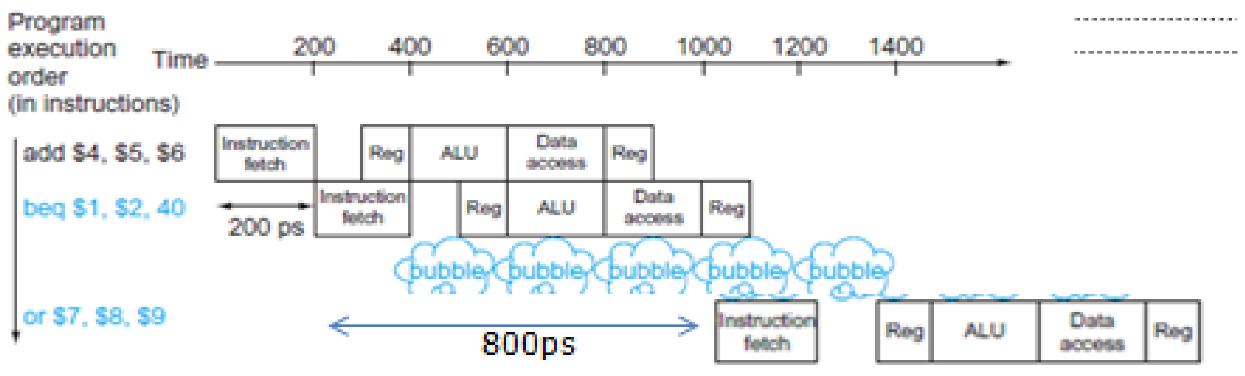
control hazard – Also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
We can identify there is a control hazard whenever there is a branch instruction in the given code sequence.

Solution:

- Stalling
- Branch Prediction – not taken
- Reduce branch delay
- Dynamic branch prediction

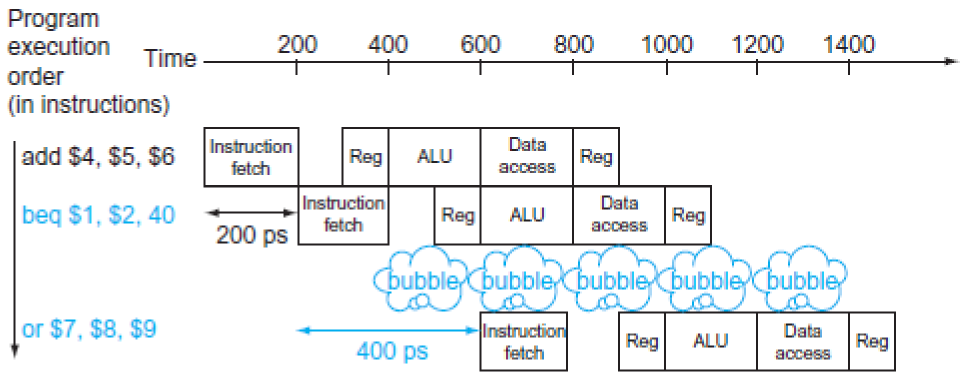
4. Stalling:

The decision task in a computer is the branch instruction. we must begin fetching the instruction following the branch on the very next clock cycle. Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory. One possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.
In the pipeline diagram, we assumed that the branch decision is available in the 4th stage MEM. So 3 bubbles have been inserted (i.e.) 3 clock cycles are wasted to solve the control hazard.



2. Reduce branch delay:

We can reduce the branch delay further by making the branch decision to be taken in the 2nd stage ID itself. Then we need to insert only one bubble (i.e.) one clock cycle is wasted to solve the branch hazard.



3. Branch prediction – untaken branch:

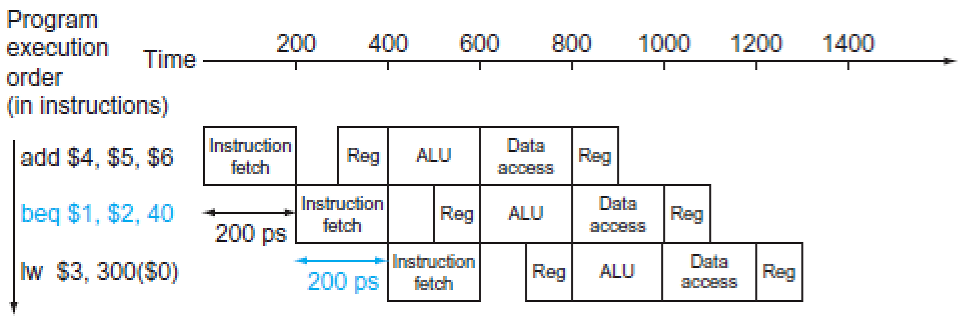
A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

UnTaken :

Predict always that branches will be untaken. When you're right, the pipeline proceeds at full speed.

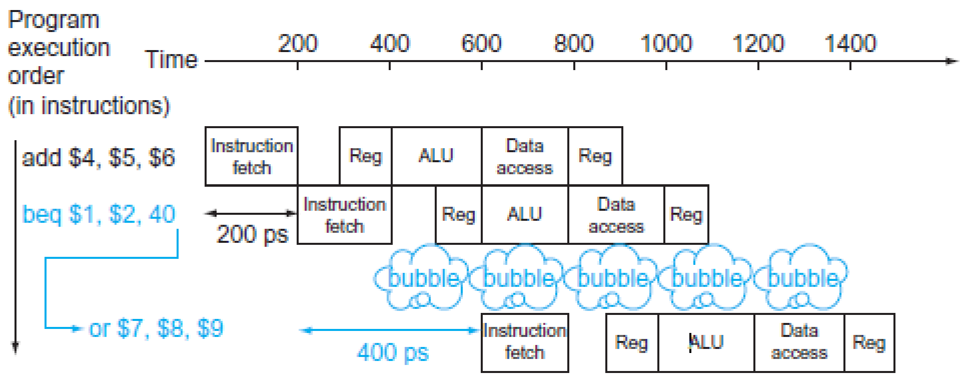
Prediction success:

If the prediction is a success, the pipeline proceeds as such without any change.



Prediction failure:

If the prediction is a failure, the branch not-taken instruction (lw as per the example given) is flushed and bubble is inserted. The branch taken instruction(or as per the example given) is fetched and starts to execute in the next clock cycle.



4. **Dynamic prediction**

Branch prediction is 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.

Hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.

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

Two schemes under dynamic prediction are:

- 1-bit and
- 2-bit scheme

1-bit scheme:

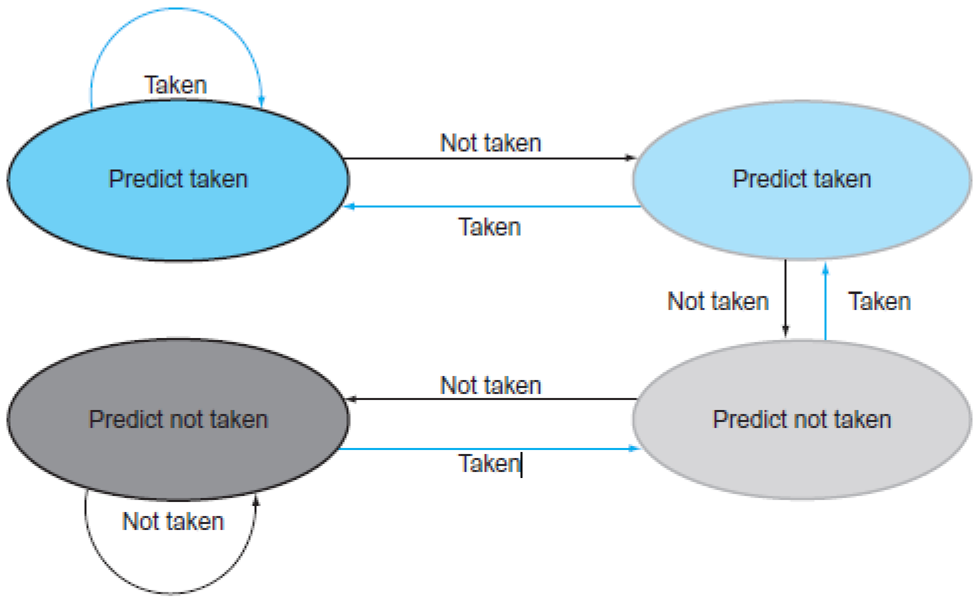
If the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.
This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

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

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

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, 2-bit prediction schemes are oft en used.

2-bit scheme:
In a 2-bit scheme, a prediction must be wrong twice before it is changed. The figure shows 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.

7. How exceptions are handled in MIPS

exception – Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.
interrupt - An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

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

The two types of exceptions that our current implementation can generate are:

- execution of an undefined instruction and
- an arithmetic overflow

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

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception.

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

In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception. For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

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

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or eight instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

We will need to add two additional registers to our current MIPS implementation:

■ *EPC:* A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)

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

8. Explain in detail about building a datapath. (16marks)

Building a Datapath

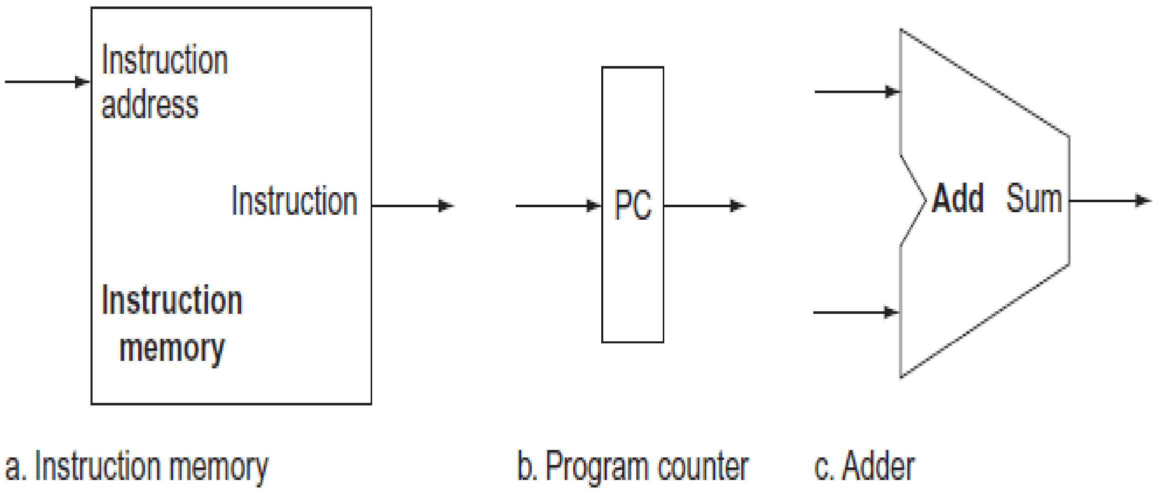
Two components of processor are Datapath and Control.

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

Elements used to fetch instructions and increment the PC:

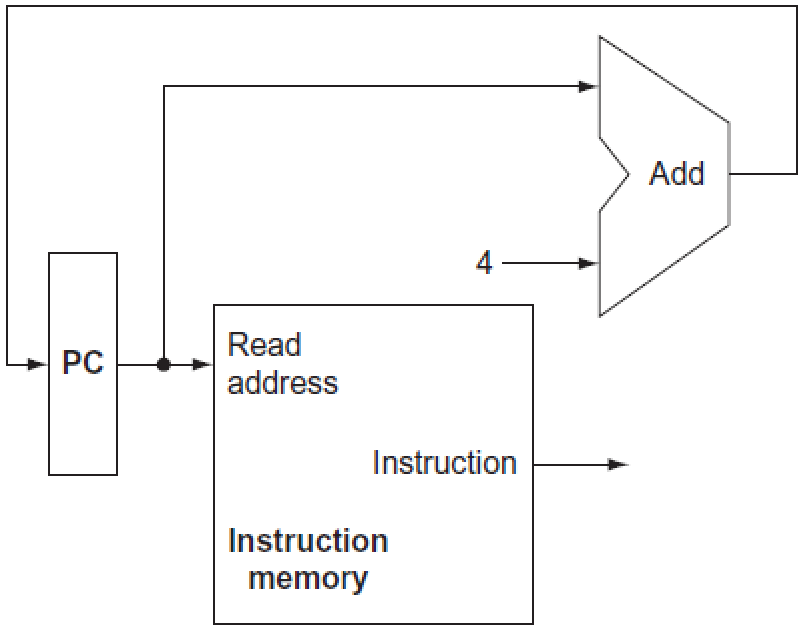
3 elements are used to fetch instructions and increment PC:

1. Instruction memory - A state element where instruction is stored.
2. PC – program counter - A state element or register containing the address of the next instruction to be executed.
3. Adder - an ALU wired to always add its two 32-bit inputs and place the sum on its output.



Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

Datapath for incrementing PC and fetching the instruction:



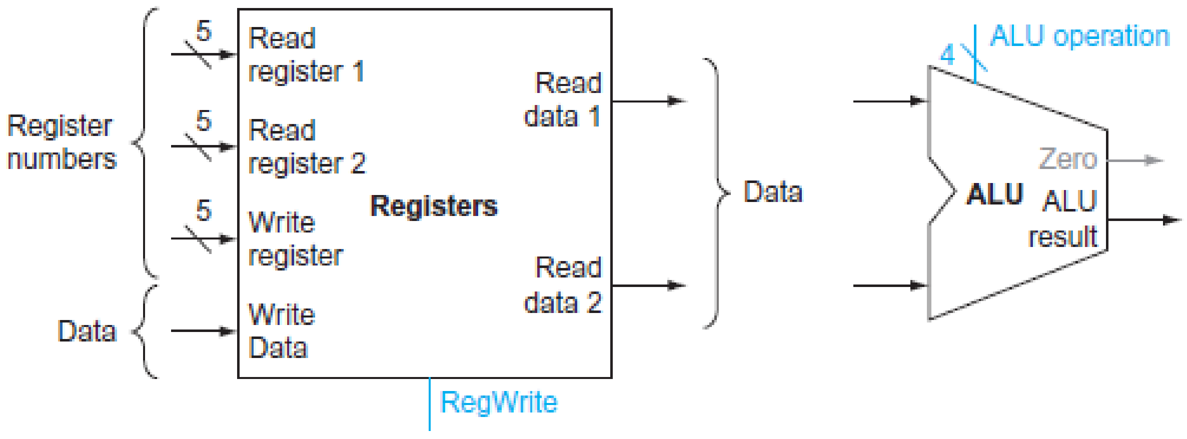
Implementing Datapath for each instruction classes:

- Arithmetic and logical instructions – R-type
- Load store instructions – I-type
- Control – conditional – I-type
- Control – unconditional instructions - J-Type

Elements used in R-Type instructions

2 elements are used in R-Type instructions

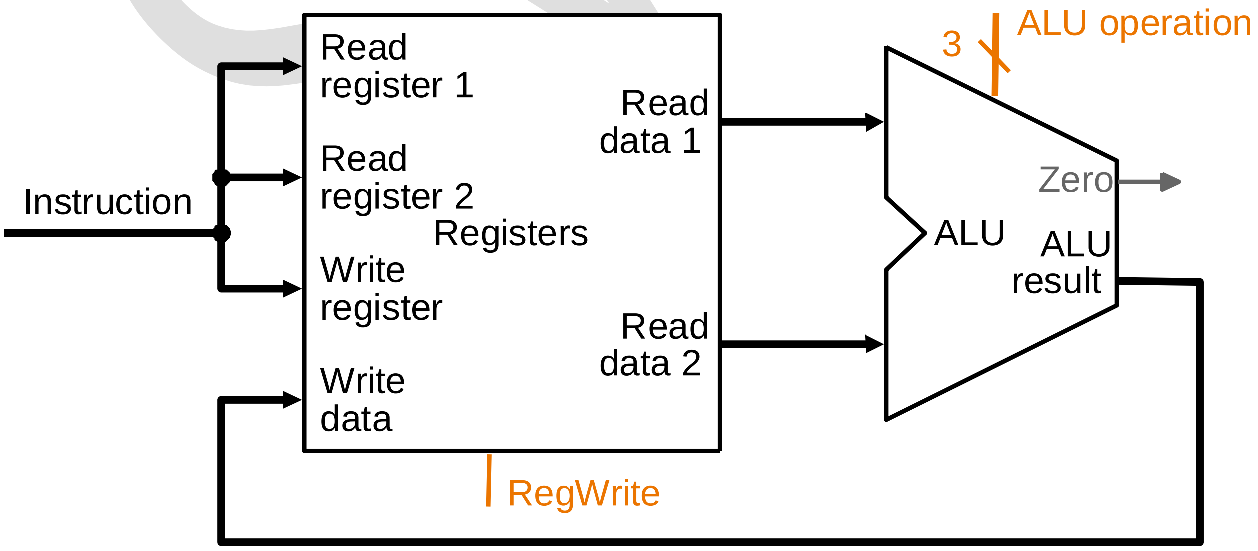
- **Register file -** A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.
- **ALU -** A combinational element that performs arithmetic and logical operations on the input data.



a. Registers b. ALU

The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal. We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until, when we discuss exceptions; we omit it until then.

Datapath for implementing R-type instructions:



The inputs to register file are the register numbers from the MIPS instruction. The output Read data 1 and 2 outputs the value stored in the respective registers in the register file.

Two operations take place in register file:

1. Read
2. Write

Read register file:

Inputs are Read register 1 and Read register 2 numbers. The outputs are Read data 1 and 2.

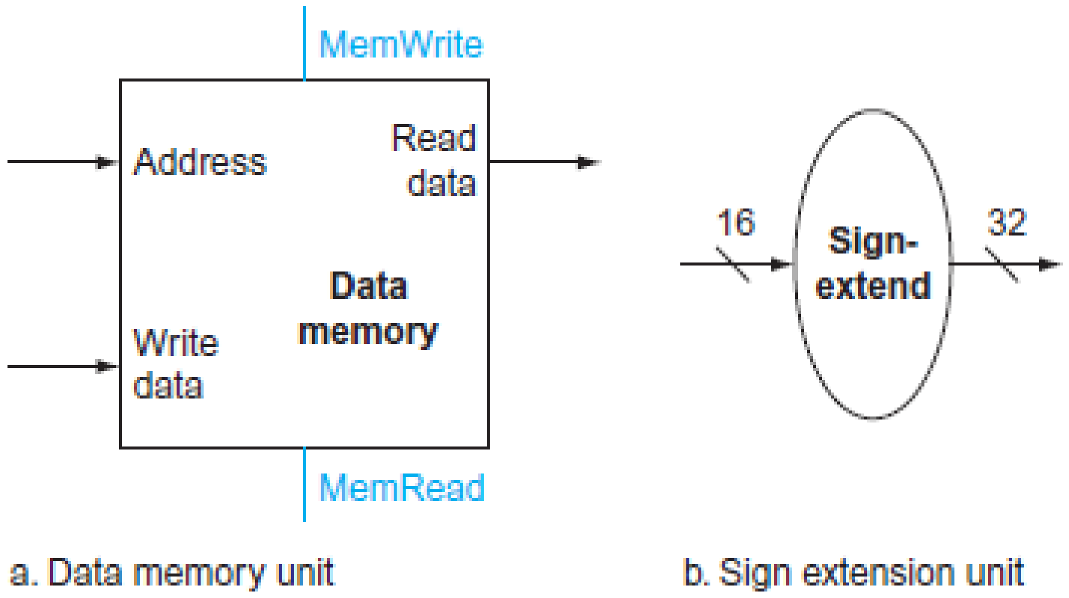
Write register file:

Inputs are Write register number and write data. There is no output during writing to a register file.

Elements used in Load/Store instructions

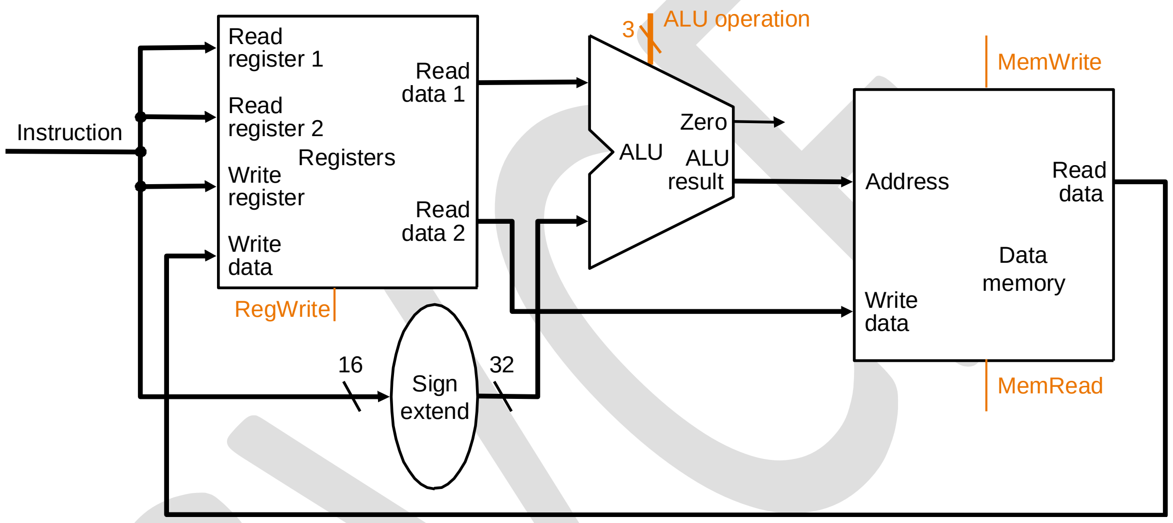
4 elements are used in Load/Store instructions

- Data memory - The memory unit is a state element with inputs for the address and the write data, and a single output for the read result
- Sign Extend- a 16-bit input that is sign-extended into a 32-bit result appearing on the output
- Register file
- ALU



The two units needed to implement loads and stores, in addition to the register file and ALU of are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output. Standard memory chips actually have a write enable signal that is used for writes.

Datapath for implementing Load/Store Instruction:



In load instruction execution the data memory is read and not written. Also we read and write data from Register file. In the above figure the highlighted lines shows the flow of data.

In store instruction execution the data memory is written and not read. Also we only read from register file. In the above figure the highlighted lines shows the flow of data.

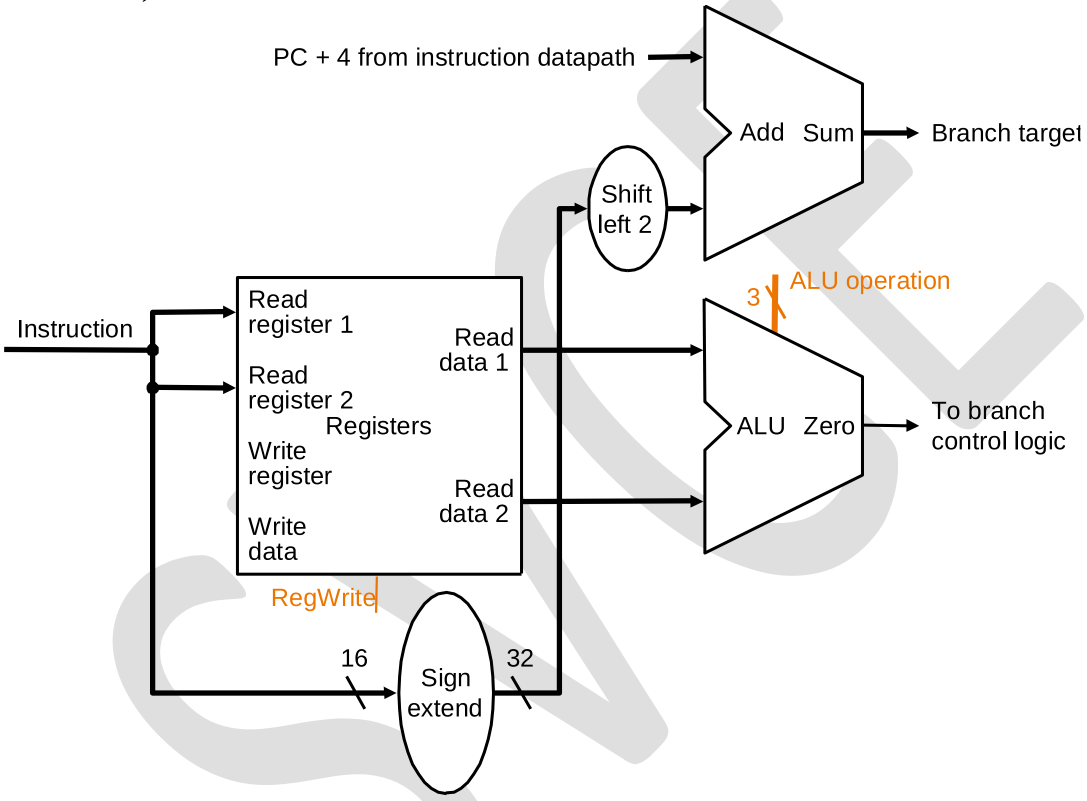
Implementing Datapath for Branch Instruction:

Branch A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.
Branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.
Branch taken - A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional jumps are taken branches.
Branch not taken or (untaken branch) A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch

The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.

The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00two to the low-order end of the sign-extended off set field. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only “sign bits.” Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

If zero is 0, then branch condition is false. So PC = PC + 4
If zero is 1, then branch condition is true. So PC = BTA



- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch. Since we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.
- The architecture also states that the offset field is shifted left 2 bits so that it is a word off set; this shift increases the effective range of the offset field by a factor of 4.

The highlighted lines show the flow of data in the datapath for branch instructions.

Creating a Single Datapath:

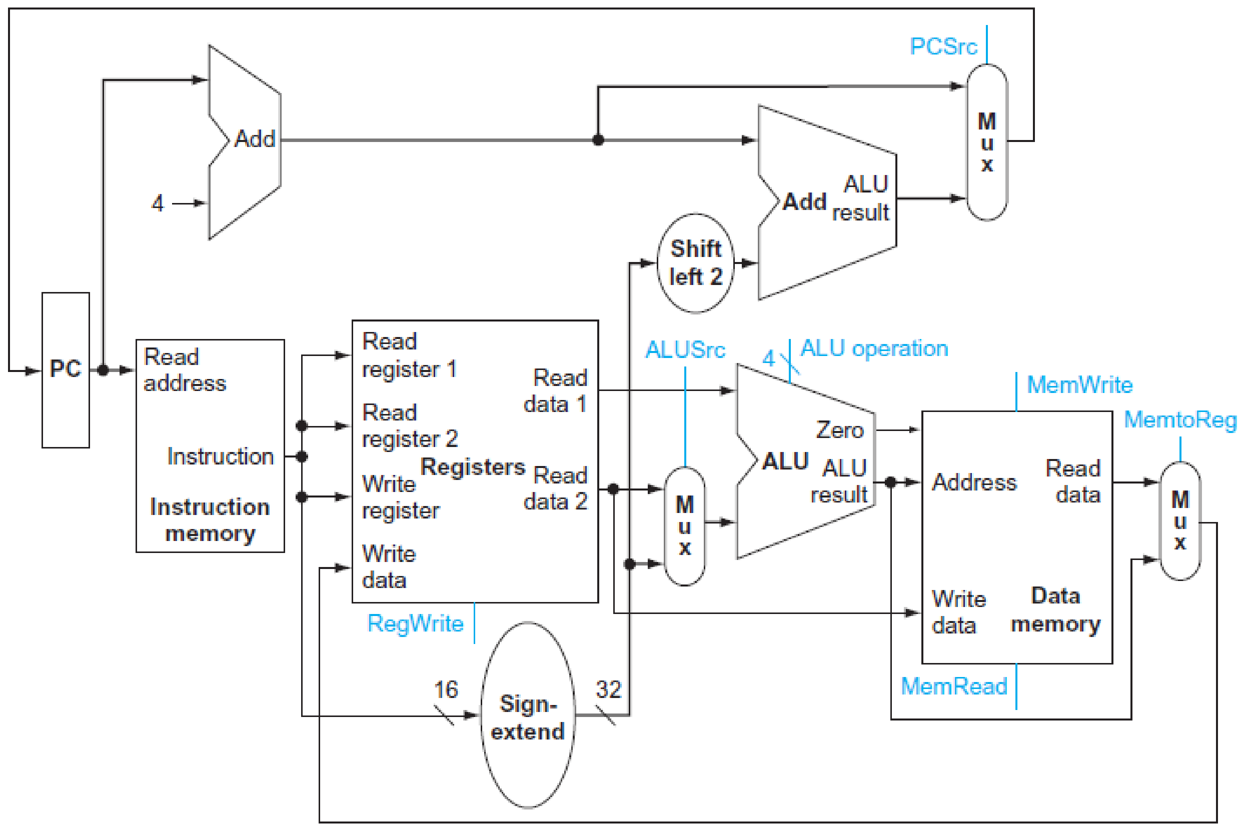
Combining the datapaths for R-type instructions, load/stores and branch instructions:

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 16-bit off set field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load). 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 multiplexors.

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 multiplexor is placed at the ALU input and another at the data input to the register file.***

The simple datapath for the core MIPS architecture combines the elements required by different instruction classes. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches. The support for jumps will be added later.



9. Explain in detail about control implementation scheme.

Two components of processor are

- Datapath
- Control

Control - Sends control signals to all other units

Input and output of control unit:

Input is the instruction opcode bits.

Control unit generates (Output):

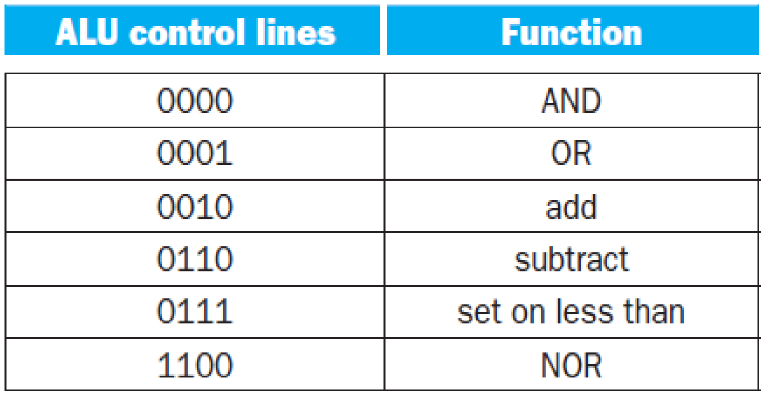
- ALU control input
- write enable (possibly, read enable also) signals for each storage element
- selector controls for each multiplexor

The ALU Control

Inputs to ALU control are:

- ALUOp and
- Instruction funct field.

Output of ALU control:



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

Setting ALU Control Bits

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

How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we "don't care" about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

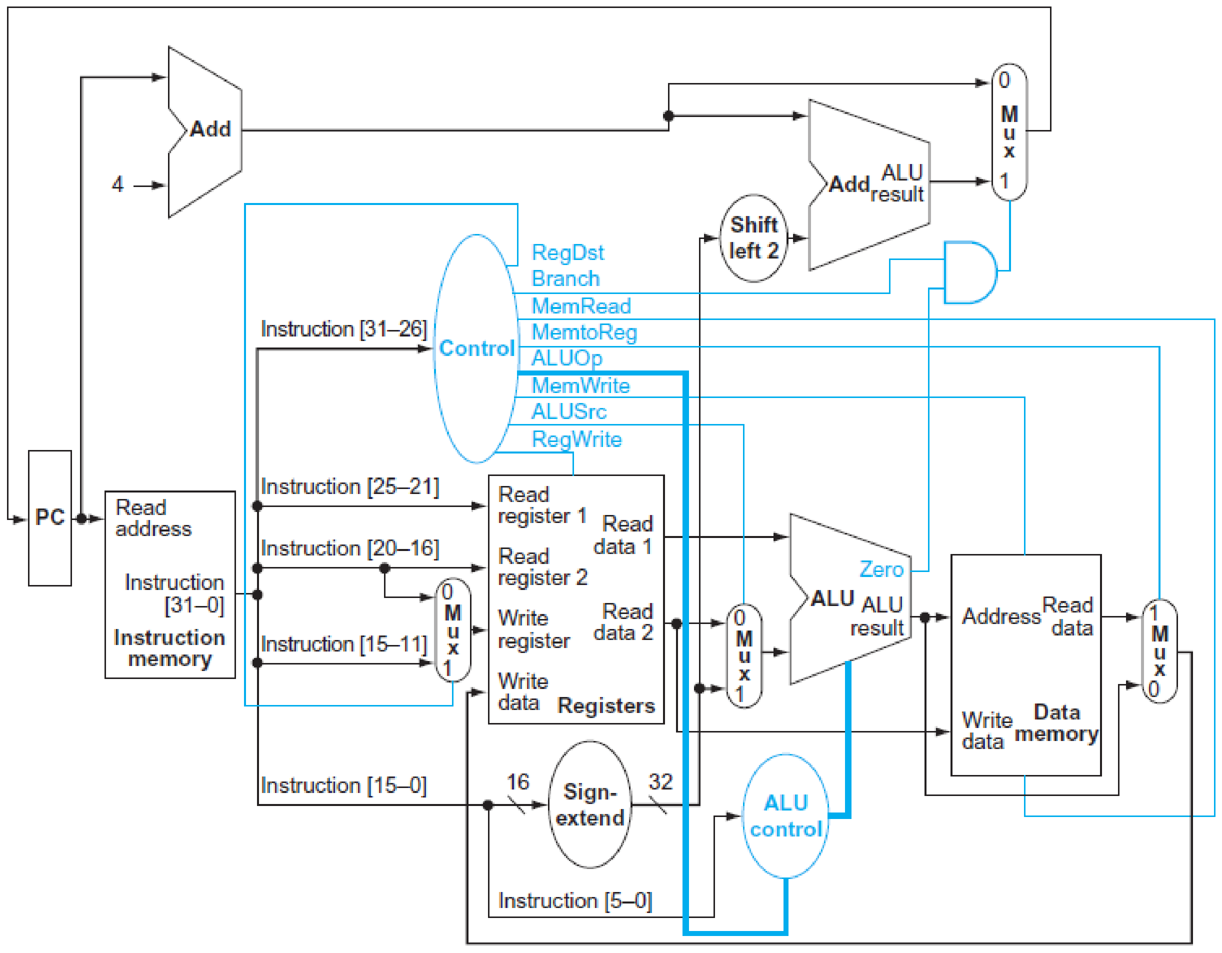
ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Designing the Main Control Unit

The simple datapath with the control unit.

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



The control lines are shown in blue color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

The above figure shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution. The below table describes the function of these seven control lines.

The effect of each of the seven control signals. When the 1-bit control to a two way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems.

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

The setting of the control lines is completely determined by the opcode fields of the instruction.

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

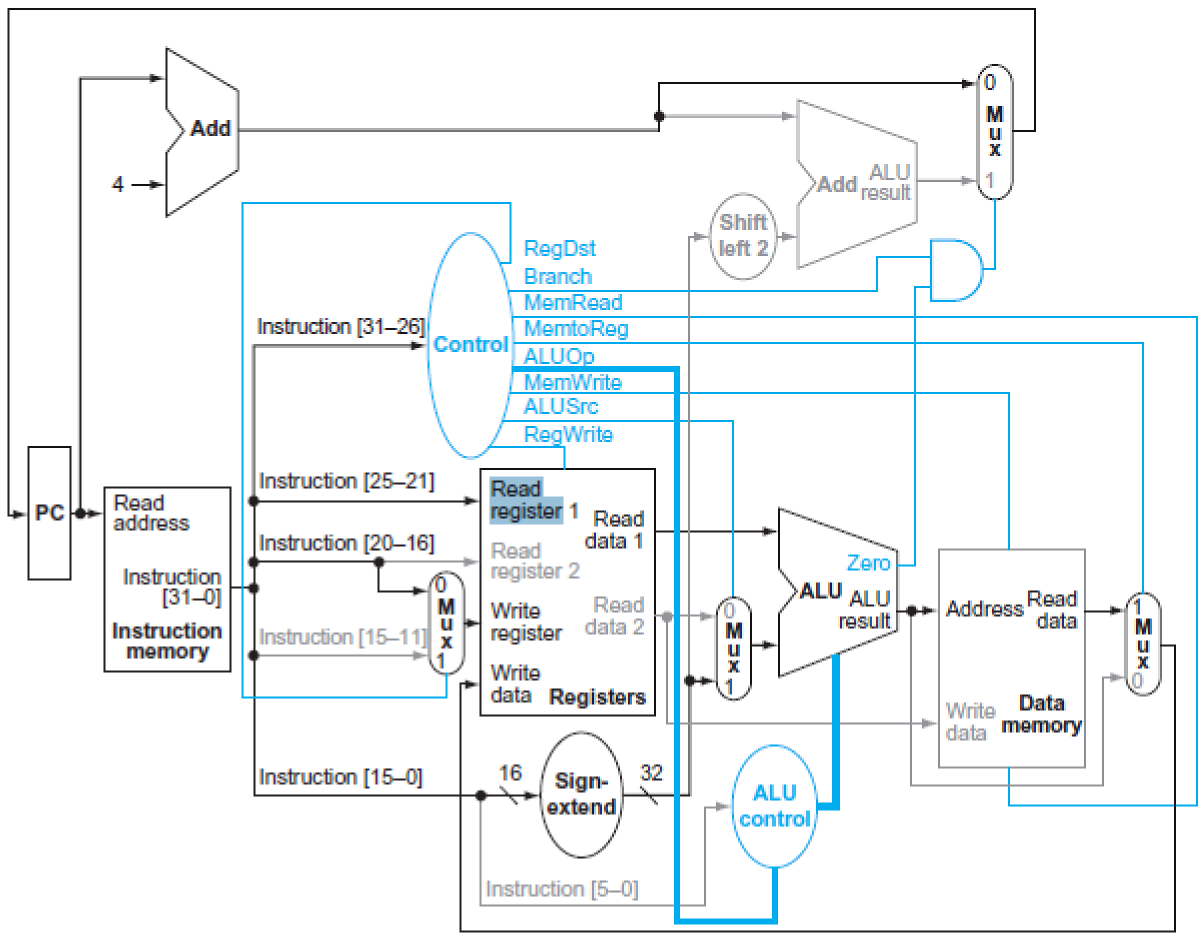
The first row of the table corresponds to the R-format instructions (add, sub, AND, OR, and slt). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory. When the Branch control

signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp
field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp
field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite 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 MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

10. Draw and explain the datapath in operation for a load word instruction (8marks)

The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted.

The execution of a load word, such as lw $t1, offset($t2) is illustrated in the below figure.
We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):
1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ($t2) value is read from the register file. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

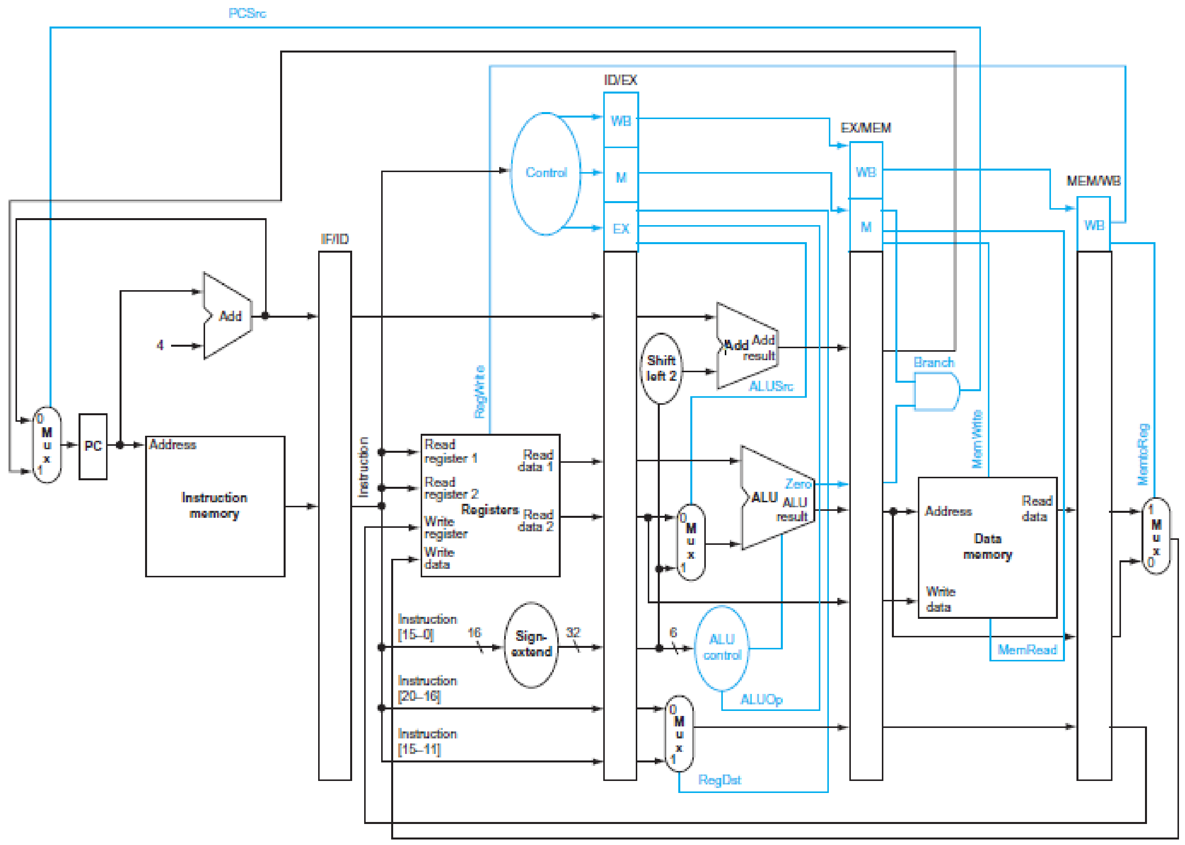


A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

11. Draw and explain the pipelined datapath with control signals (16marks)

The pipelined datapath with the control signals connected to the control portions of the pipeline registers

The control values for the last three stages are created during the instruction decode 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.



The datapath is separated into five pieces, with each piece named corresponding to a stage of instruction execution:
1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

Pipeline register is the intermediate register between two 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. 4 pipeline registers are IF/ID, ID/EX, EX/MEM and MEM/WB.

The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

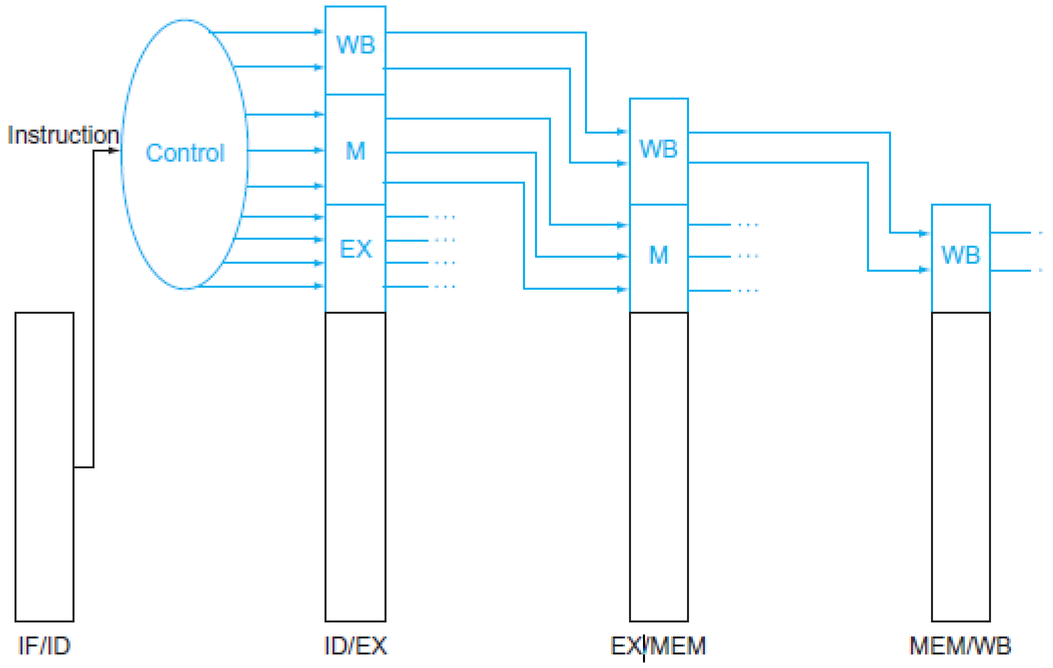
The 9 control signals are grouped as below according to the stage in which it is used.

- Execution/address calculation stage control lines – RegDst, ALUOp1, ALUOp0, ALUSrc
- Memory access stage control lines – Branch, Mem-Read, Mem-Write
- Write back stage control lines – Reg-Write, Memto-Reg

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

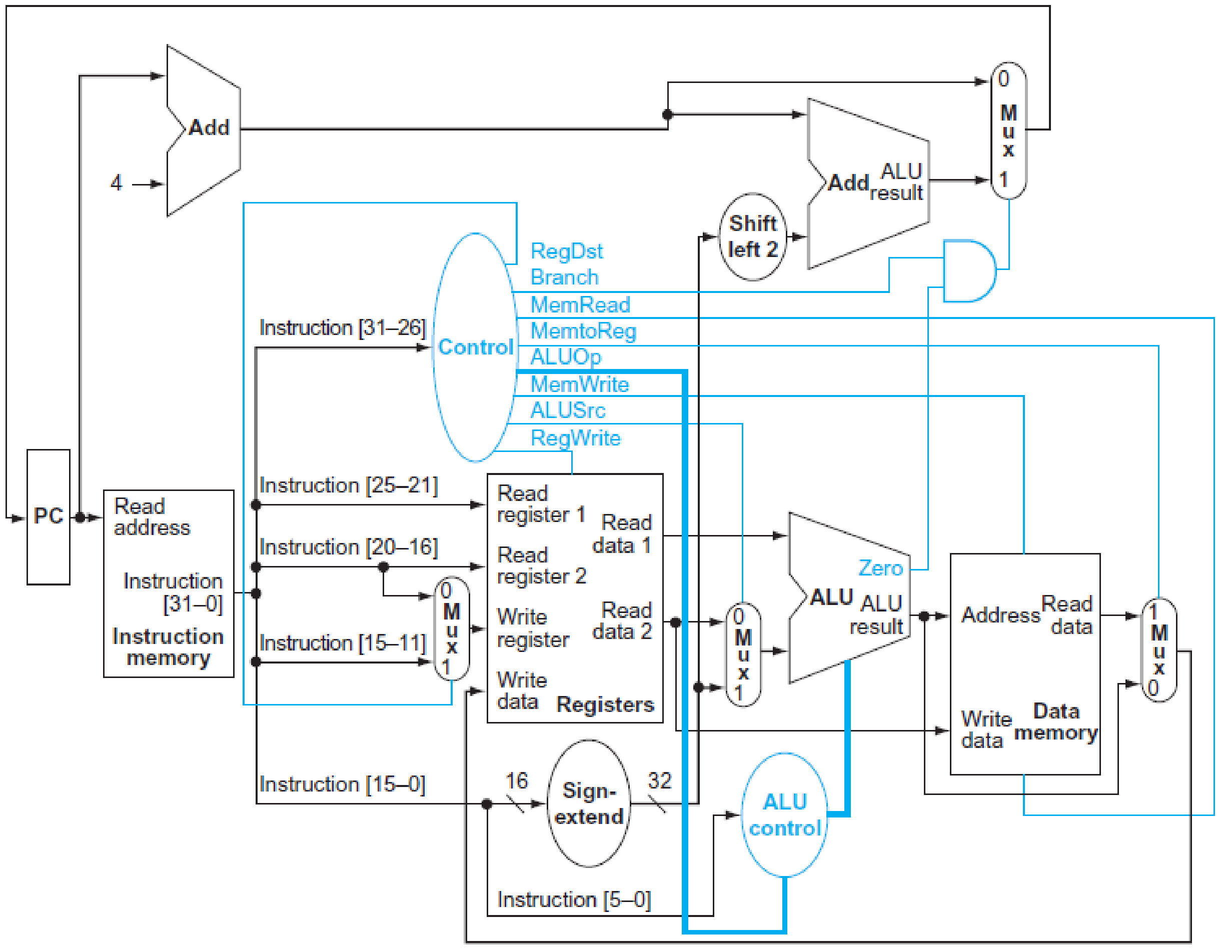
The control lines for the final three stages.

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.

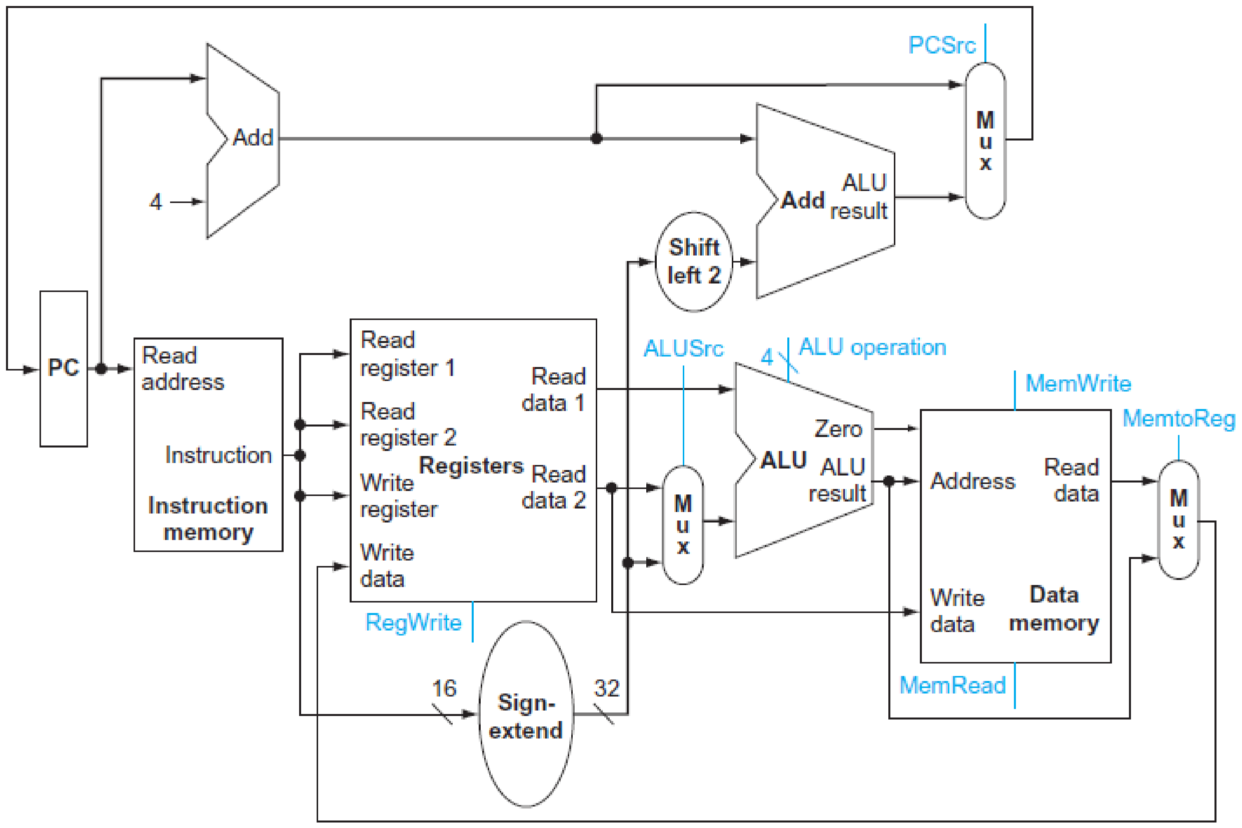


Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/ WB for use in the WB stage.

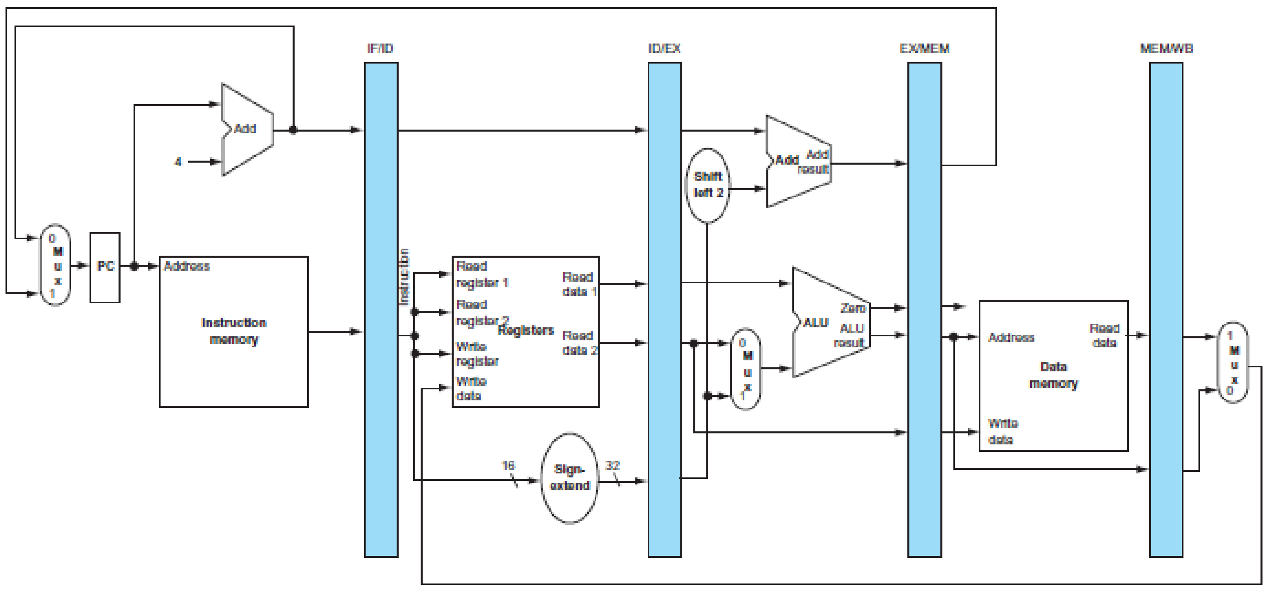
12. Draw the simple datapath with main and ALU control unit that shows the complete datapath Control implementation



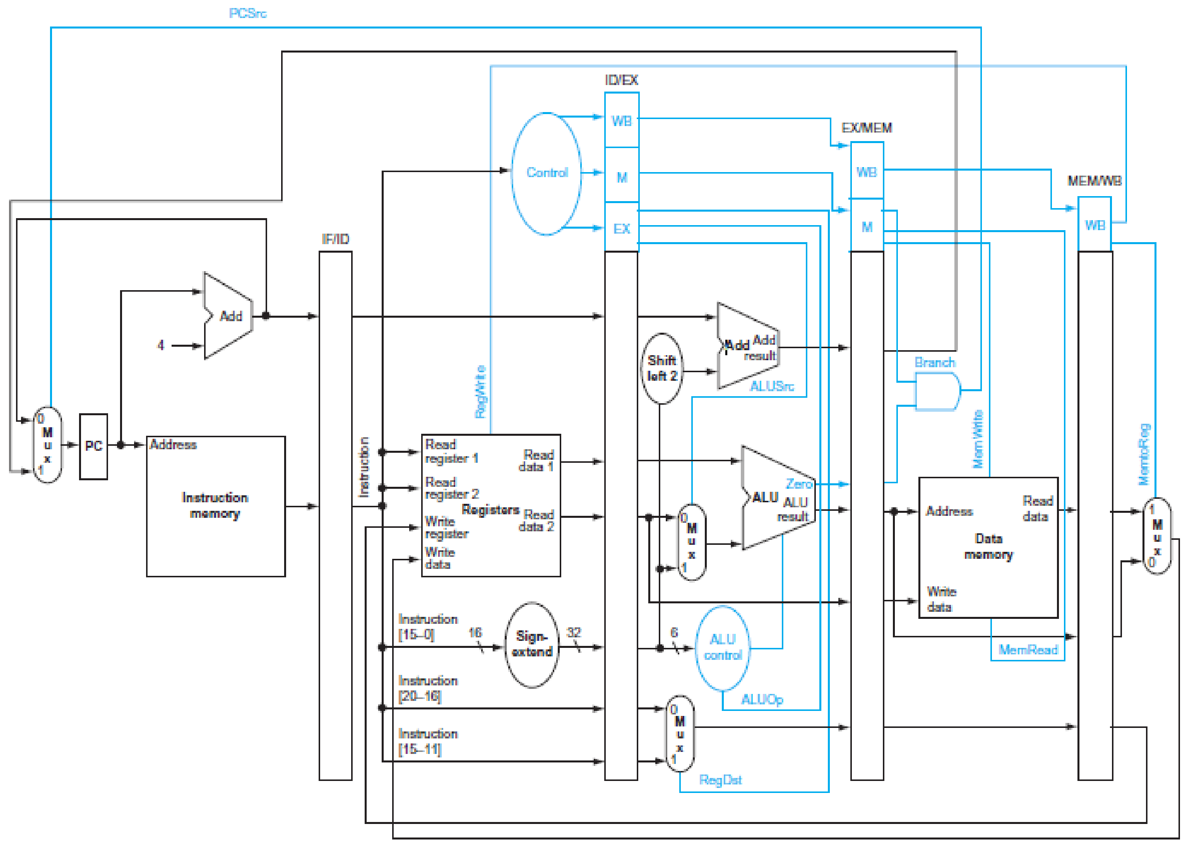
13. Draw the datapath diagram for the MIPS architecture which executes all the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle?



14. Draw the pipelined datapath with pipeline registers that shows the complete pipelined datapath implementation



15. Draw the pipelined datapath with pipeline registers, main control unit and ALUcontrol unit that shows the complete pipelined datapath control implementation


PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
Add
4
Add
Add result
Shift left 2
ALUSrc
Branch
Zero
ALU
ALU result
MemWrite
MemtoReg
RegWrite
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Address
Read data
Data memory
Write data
Instruction [15–0]
16
Sign-extend
32
6
ALU control
MemRead
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst
M u x