Single Cycle MIPS Processor

CSE212: Computer Organization

Mohamed Hussein Mohamed Abomakhlouf 19P2570
Mohamed Ashraf Aboutaleb 19P7766

Participation Table

Mohamed Ashraf 19P7766	Mohamed Hussein 19P2570
Adder	Instruction memory
ALU	Control
Data memory	MUX
PC	Register File
Shift-left	Top module
Sign extention	Pipelined 5 stage registers
Top module	Forwarding unit
Hazard unit	Testbench

Table of contents

Ctrl + Click to go to slide

Table of contents

Ctrl + Click to go to slide

Table of contents

Ctrl + Click to go to slide

Table of contents

Ctrl + Click to go to slide

Table of contents

Ctrl + Click to go to slide

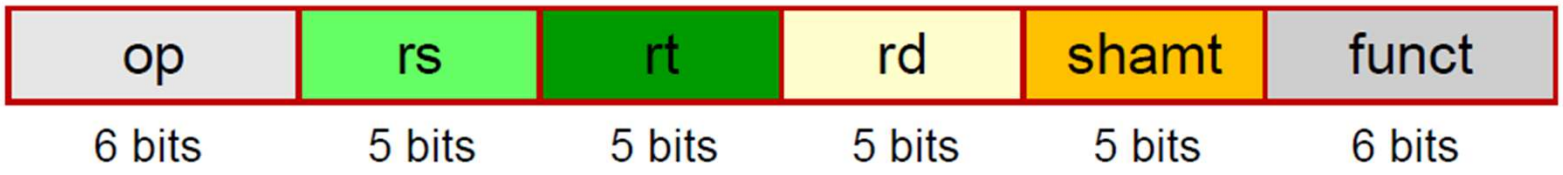
Processor implemented functions:

	(Opcode)$_{dec}$	**(Function)$_{dec}$**
Add	0	32
Add immediate	8	X
Subtract	0	34
Logic And	0	36
Logic Or	0	37
Set Less Than	0	42
Load word	35	X
Store Word	43	6
Branch if equal	4	0
Jump	2	X

R-type instructions

The MIPS instructions for R-type are add, sub, and, or, and slt.

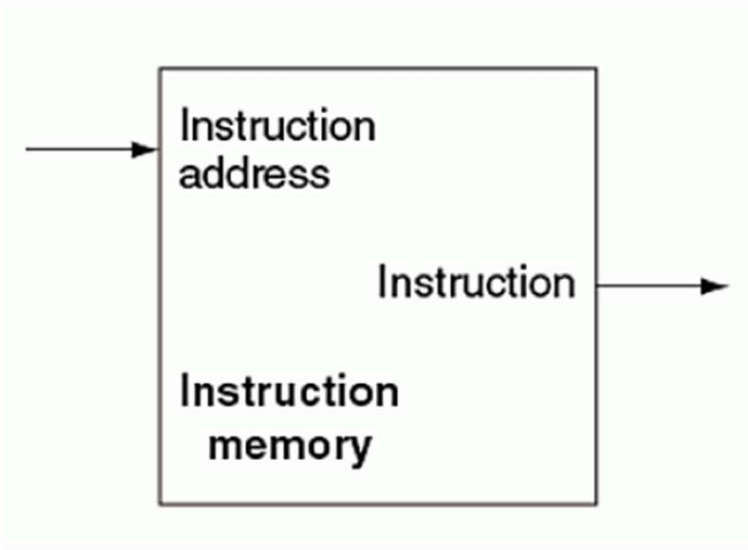
The R-type has two read registers(rs and rt)and a destination register for the arithmetic operation (rd)



Design Entry

Instruction memory

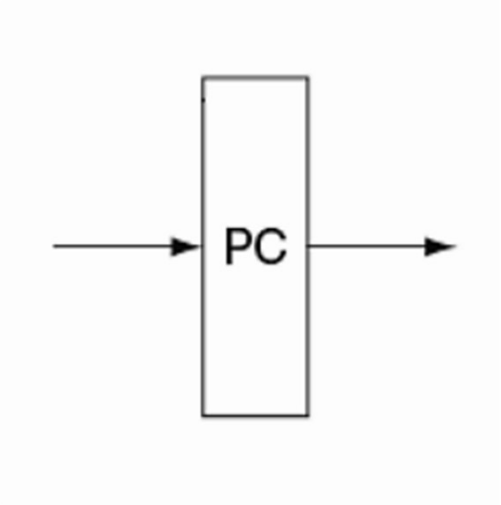
The instruction memory is used to store the instruction. It provides read access to the instructions of program, given an address as input, supplies the corresponding instruction at that address.



Design Entry

PC

it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.

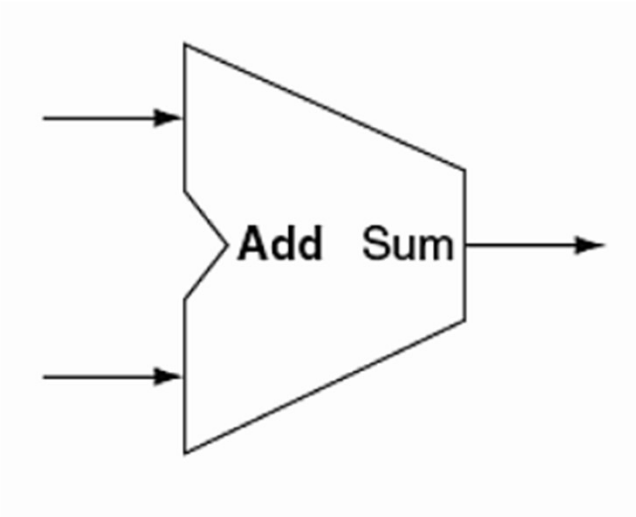


Design Entry

Adder

The adder is used to increment the PC to hold the address of the next instruction.

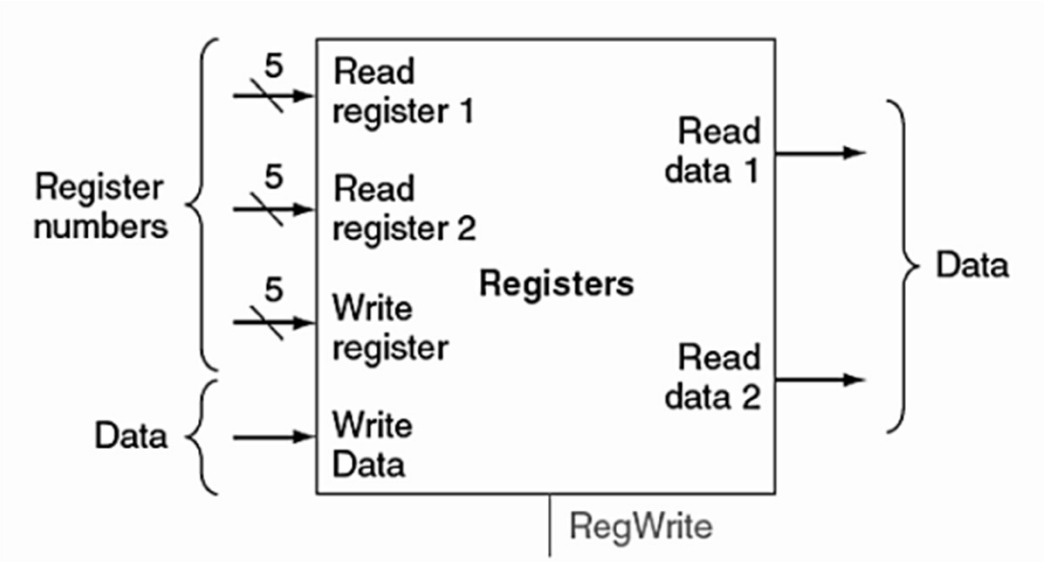
It takes two input values, adds them together and outputs the result.



Design Entry

Register file

To make the R-type instructions we will need a register file which is collection readable and writeable registers.



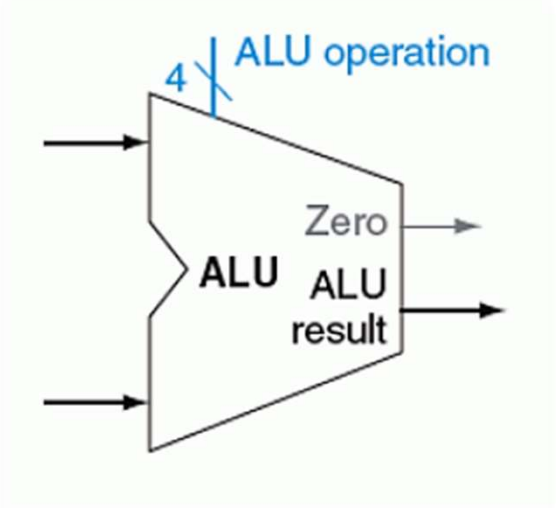
Design Entry

ALU

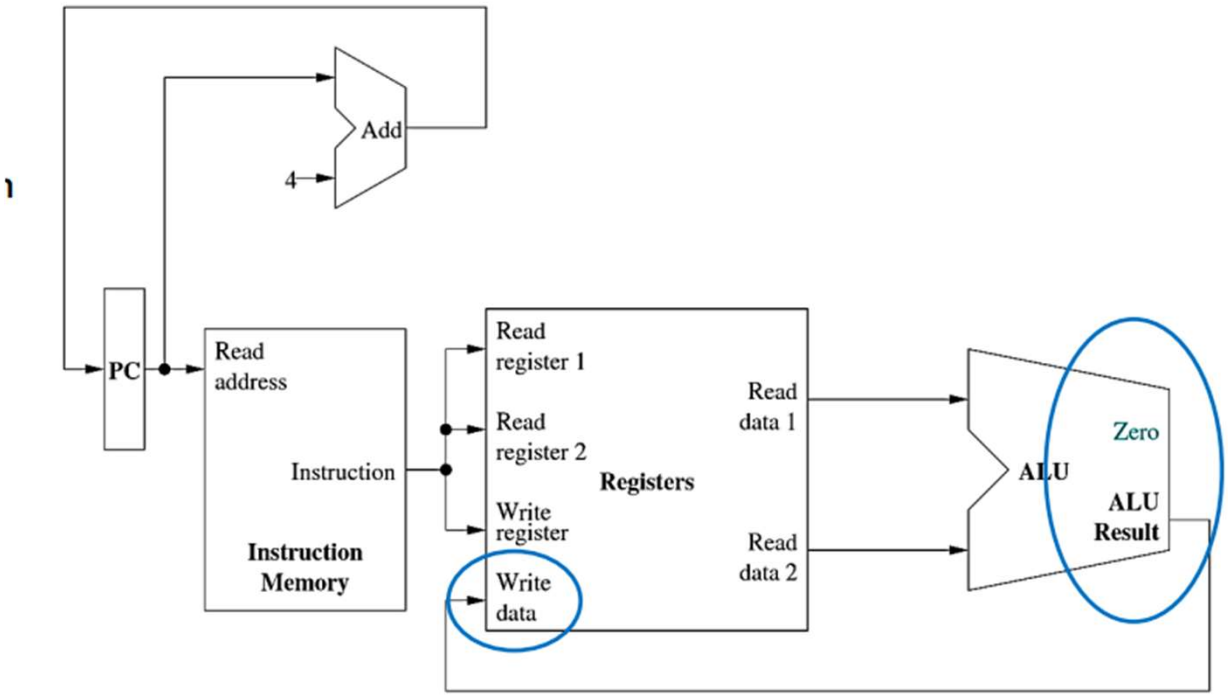
To execute the R-type instructions we need the alu.

The alu performs the operation indicated by the instruction.

It takes two inputs and a 4-bit wide operation selector value. We have two outputs one of them is specifically for branching



R-Type Datapath



I-type instructions

The MIPS instructions for I-type are lw, sw, and beq

In I-type we have a source register (rs) and the other register is either the source or destination register (rt)

The op is used to define the type of instruction

The immediate field is zero-extended if it is a logical operation. Otherwise, it is sign extended.

Design Entry

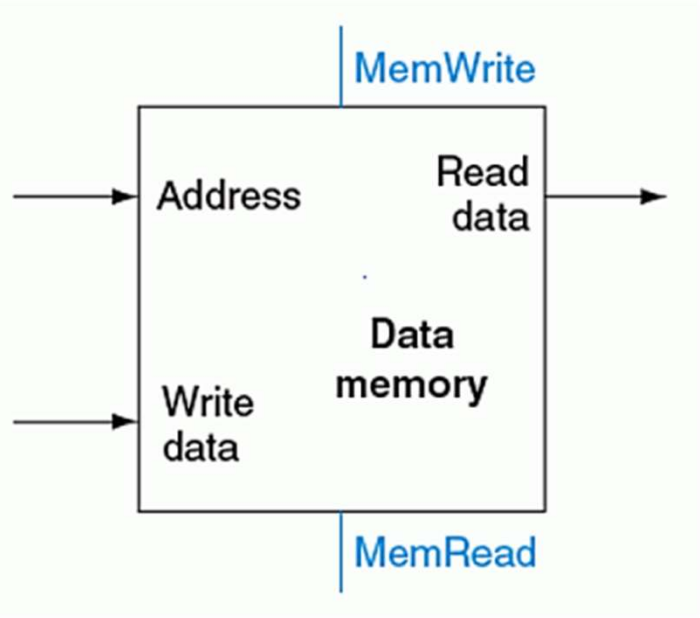
Data memory

Data memory is used for reading and writing data from and to memory.

It has two inputs one for the address of the memory location to access and the other for the data to be written to memory.

The output is the data read from the memory location accessed.

The data memory has a MemRead and Memwrite wo signal whether to read or write

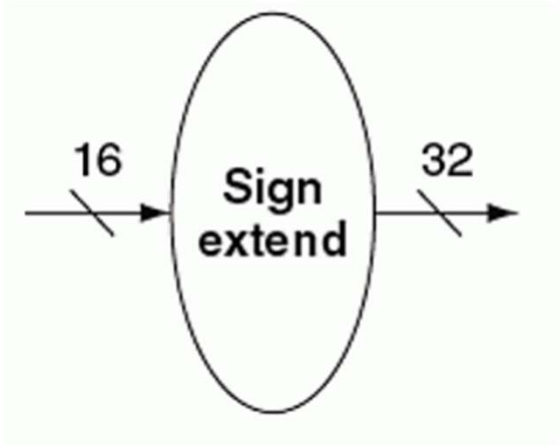


Design Entry

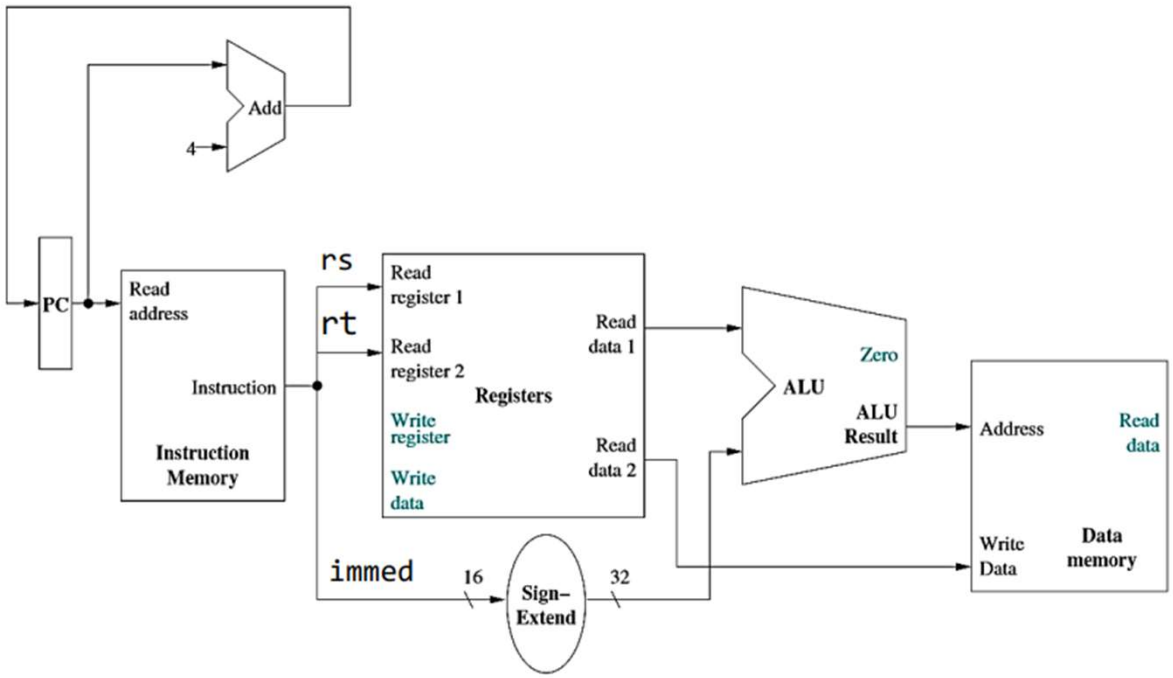
Sign extension

The sign extension is an element takes as input a 16-bit wide value to be extended to 32-bits.

To sign extend, we simply extend the most-significant bit of the original field until we have reached the desired field width.



Datapath for I- type



Branching instructions

This instruction compares between two registers value and uses the 16-bit immediate to compute the target address of the branch relative to the current address.

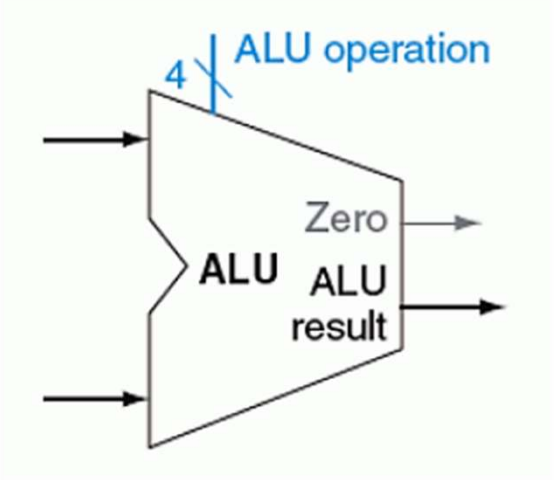
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	rt		immed	

Branching instructions

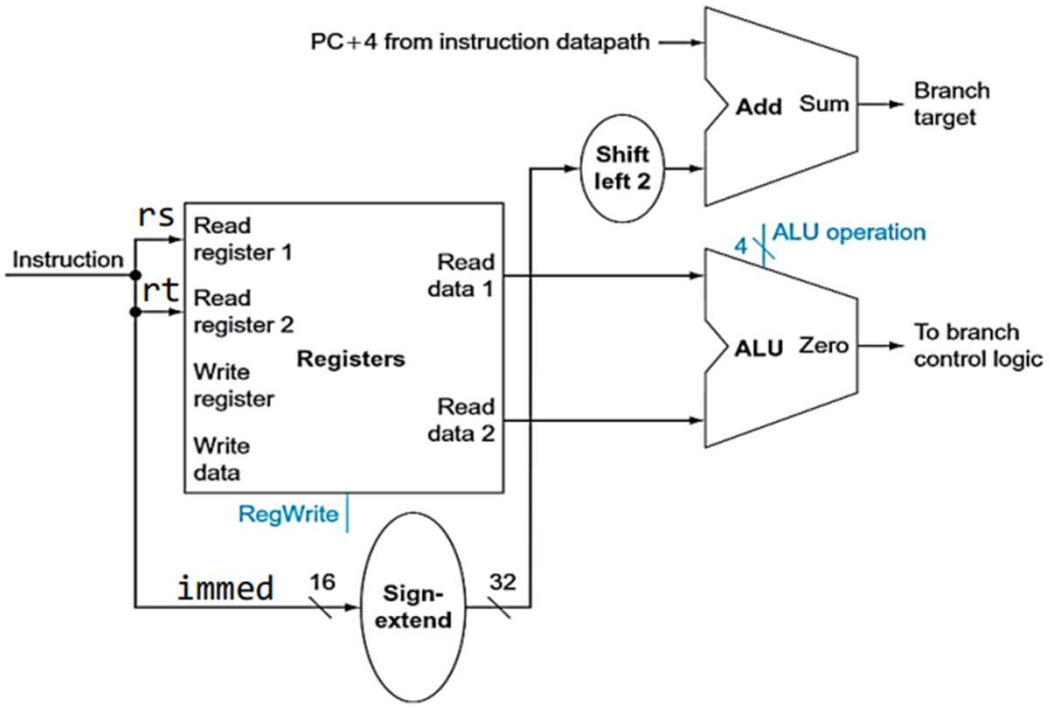
The immediate field is only 16-bits so it can’t specify the 32-bit. So it will be shifted by two because the immediate represents the number of words offset from PC+4, not the number of bytes, then we sign extend the immediate value and add it to PC+4

Branching instructions

As mentioned before in the Alu, it has an output specified when the result of the operation is zero which will be used for the branching instruction.



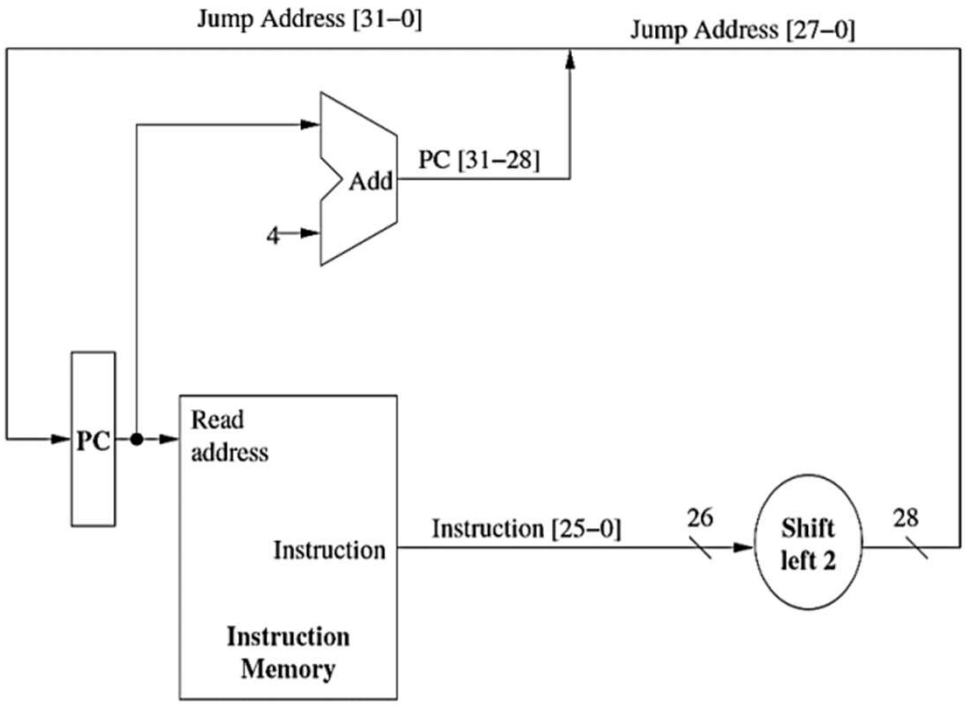
Datapath for branching instructions



J-type

This instruction indicates that the next instruction to be executed is at the address of a target address

Datapath for J-type

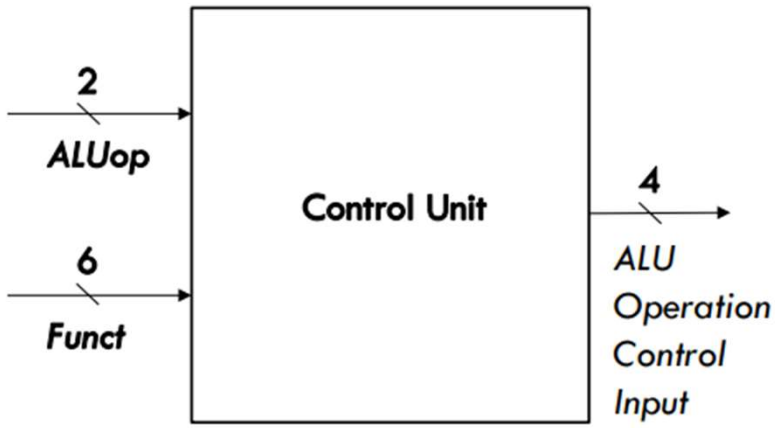


Design Entry

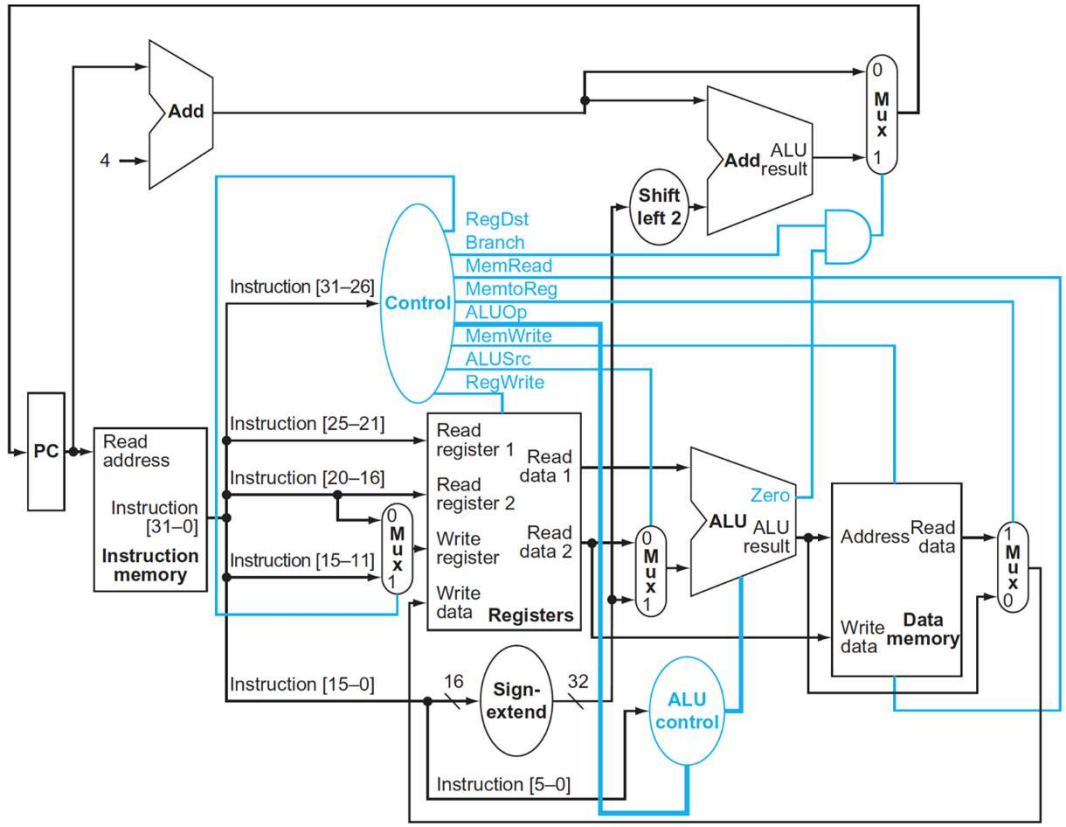
Control

The control is responsible for taking the instructions from the instruction memory and generating the appropriate signals for the Datapath.

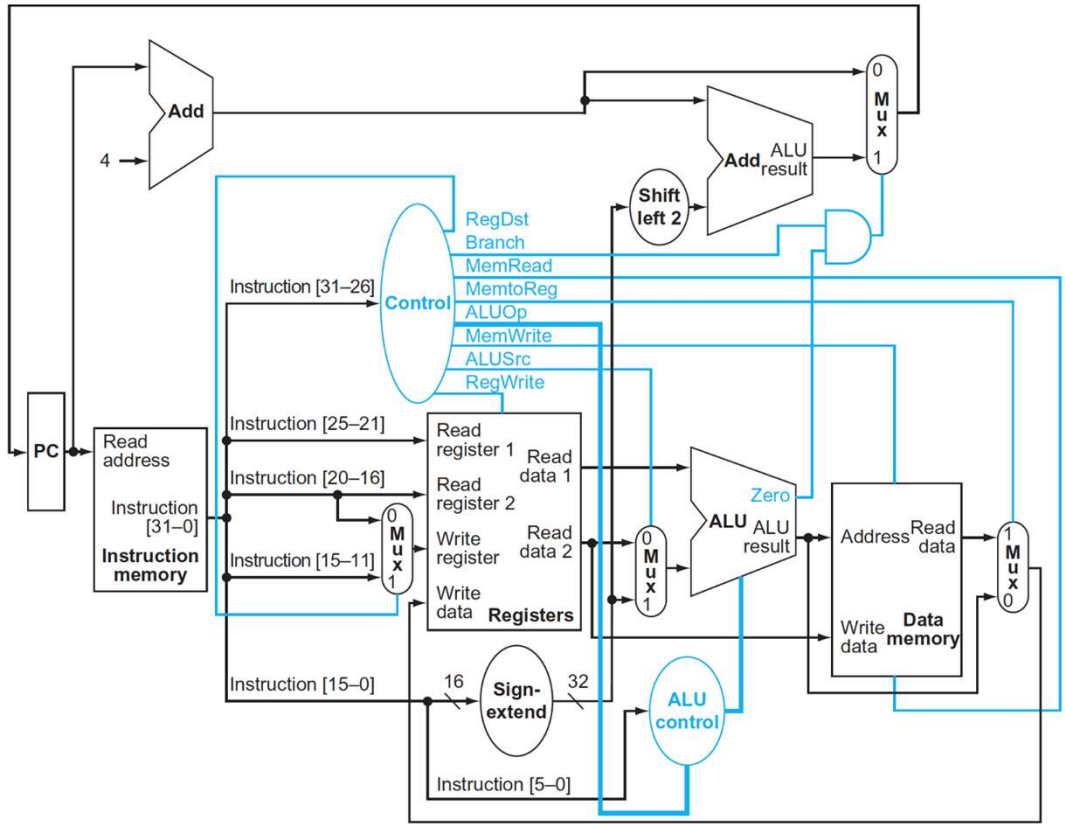
the control signals are chosen based on the upper 6 bits of the instruction. That is the opcode is used to set the control lines.



Processor combined Architecture:



After designing a single cycle processor, we will start by looking at the datapath for pipelining. In the figure shown below this was the single cycle processor



PIPELINED DATAPATH

We will break up the instructions into five stages:

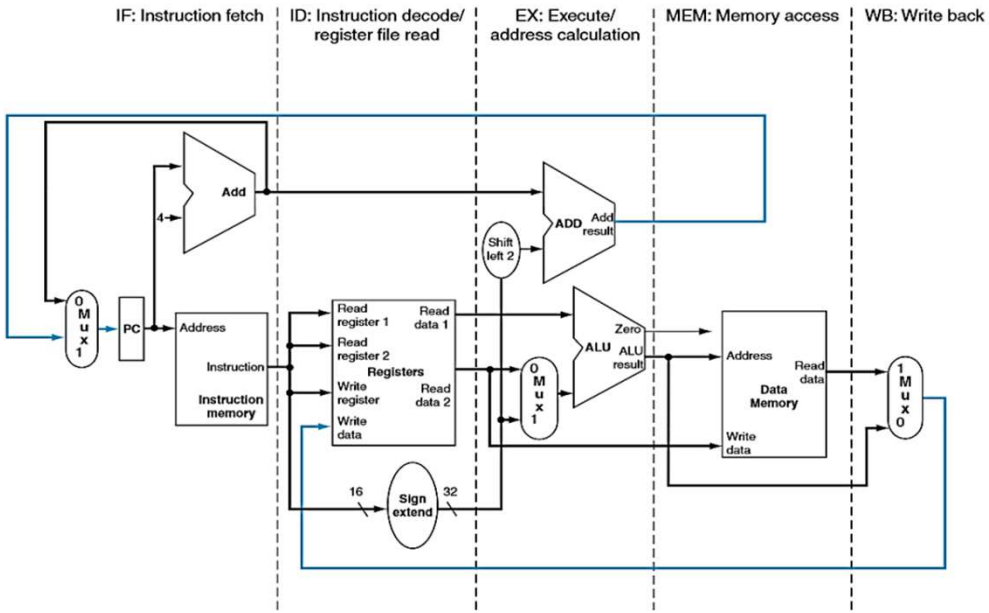
-IF – Instruction Fetch

-ID – Instruction Decode

-EX – Execution

-MEM – Memory Access

-WB – Write Back



PIPELINED DATAPATH

Instructions and data move from left to right except in two stages :

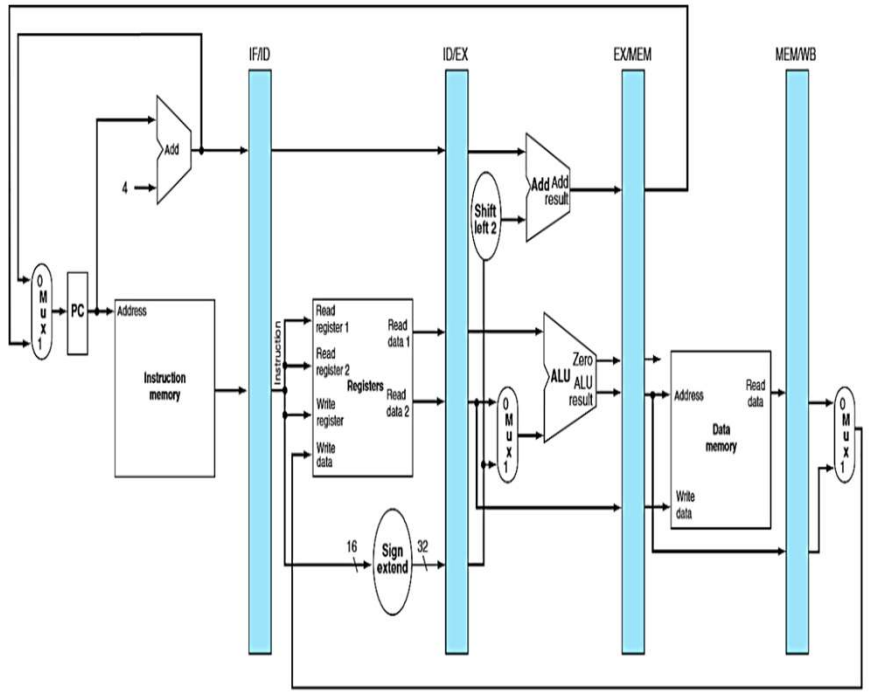
The WB stage places the result back into the register file in the middle of the datapath à leads to data hazards.

The selection of the next value of the PC – either the incremented PC or the branch address à leads to control hazards.

PIPELINED DATAPATH

After each stage we will implement a register which is used to hold the data between the cycles.

During each cycle, an instruction advances from one pipeline register to the next pipeline register.

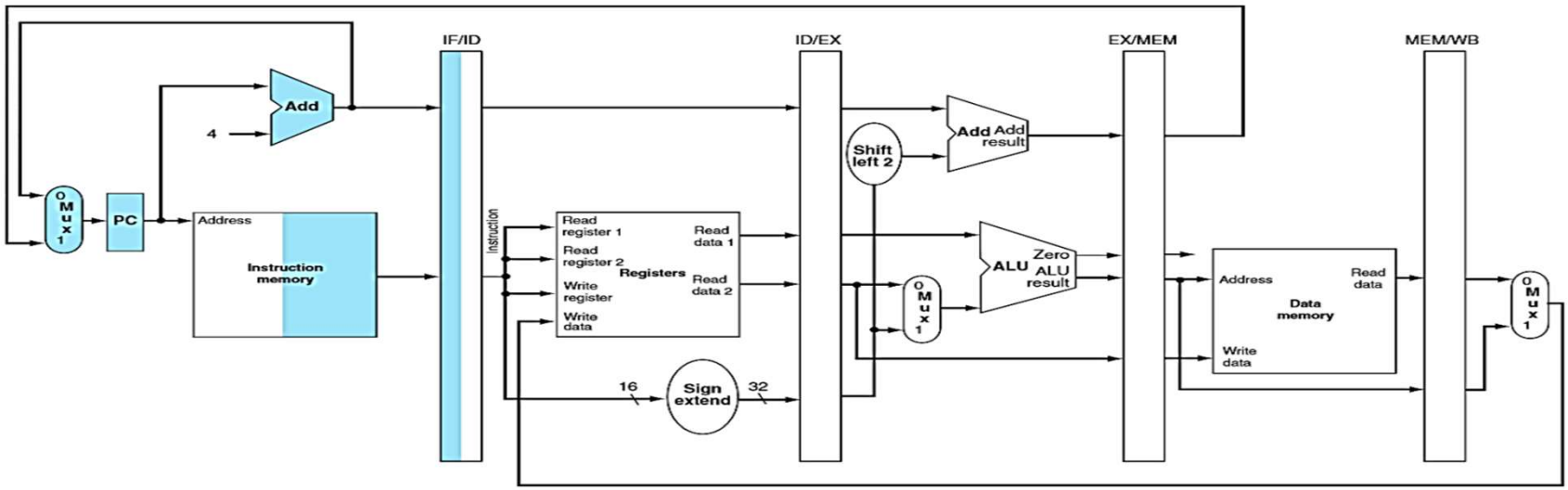


PIPELINED DATAPATH

Instruction Fetch (IF)

The instruction is read from memory using the content of PC placed in the IF/ID register.

The PC address incremented by 4 and written back to the PC register



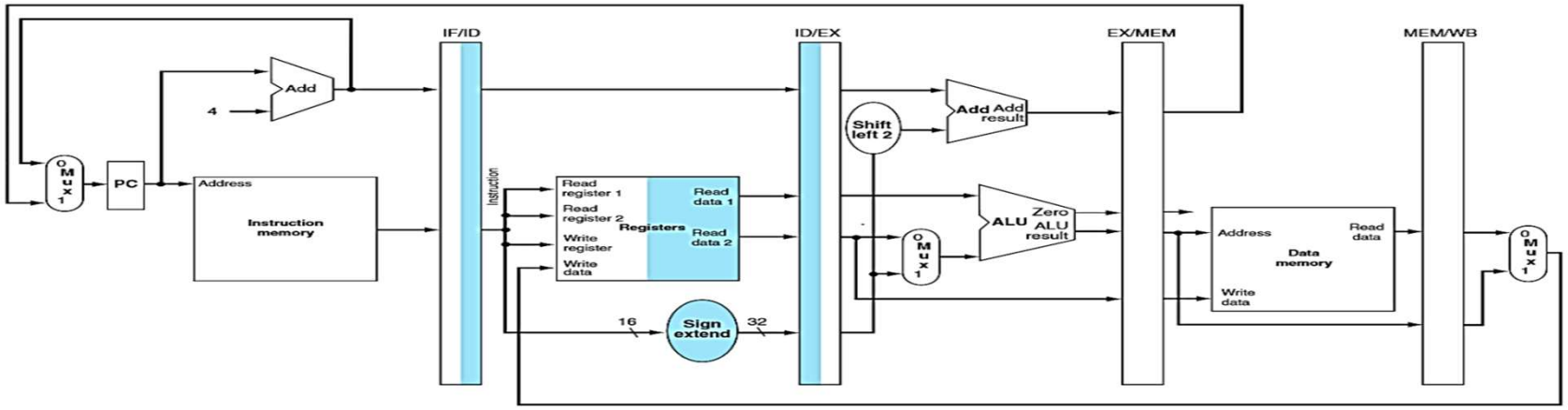
PIPELINED DATAPATH

Instruction Decode (ID)

The registers rs and rt are read from the register file and stored in the ID/EX register.

The 16-bit immediate field is sign-extended to 32-bits and stored in the ID/EX pipeline register.

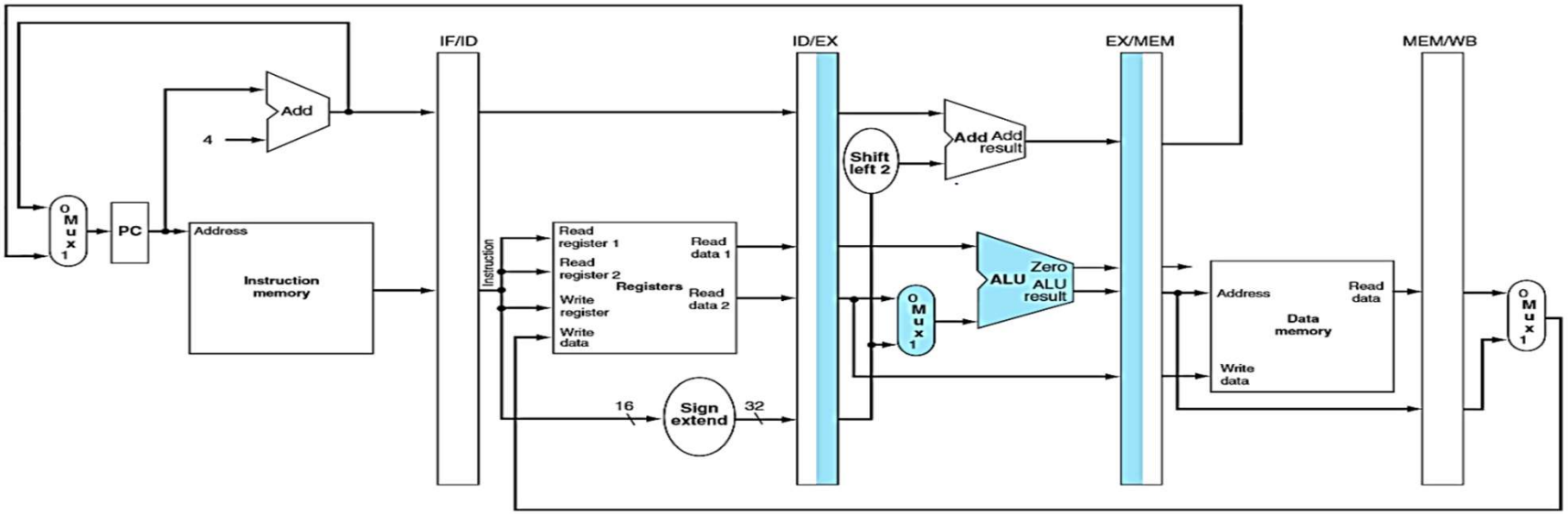
The next instruction is copied from the IF/ID register into the ID/EX register by incrementing the PC by four.



PIPELINED DATAPATH

Execution (EX)

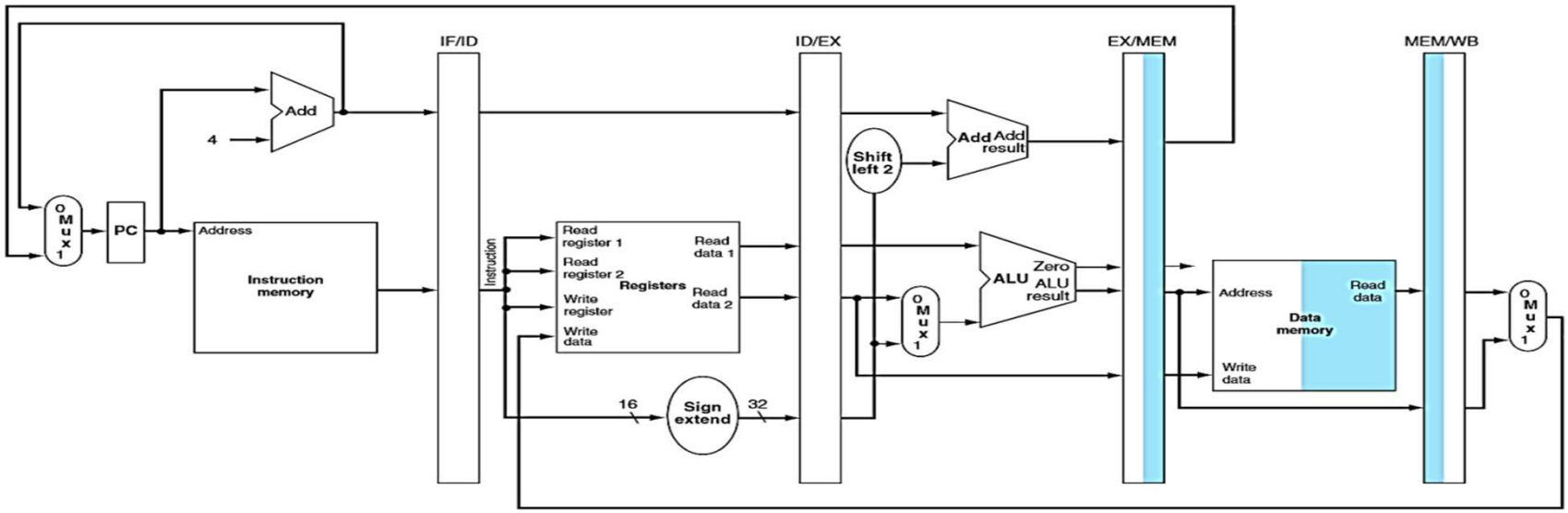
From the ID/EX register take the values of rs and the sign-extended immediate field as inputs to the ALU, which performs the operations . The result is placed in the EX/MEM register.



PIPELINED DATAPATH

Memory Access (MEM)

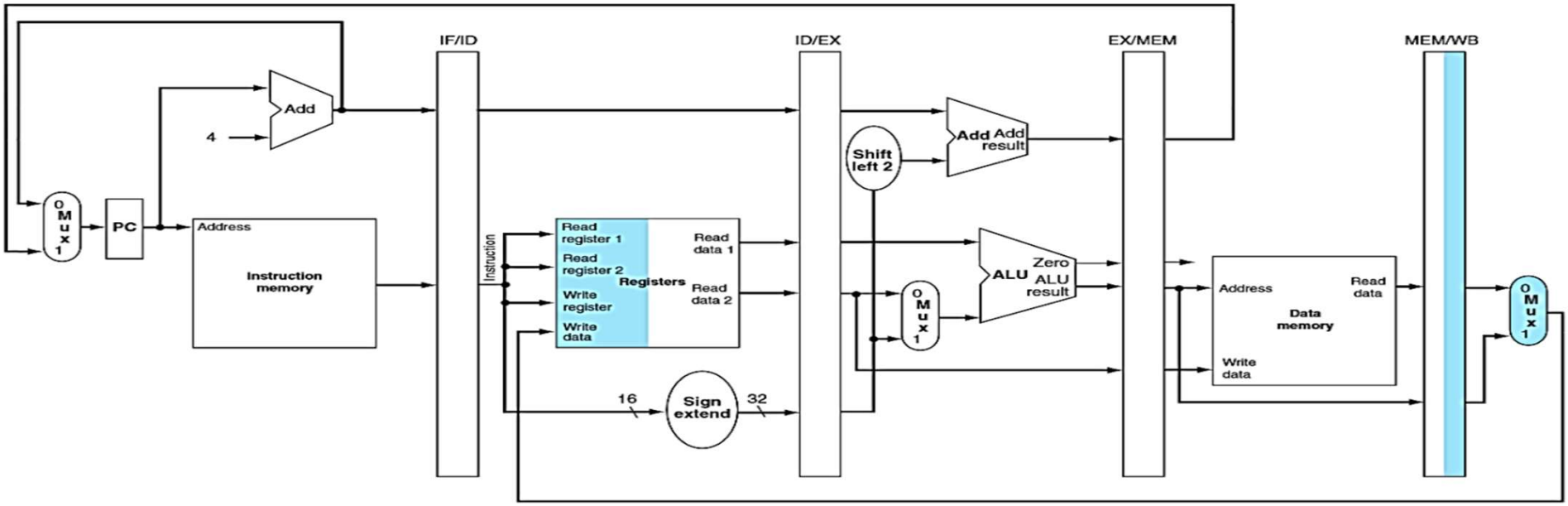
Take the address stored in the EX/MEM pipeline register and use it to access data memory. The data read from memory is stored in the MEM/WB pipeline register.



PIPELINED DATAPATH

Write Back (WB)

Read the data from the MEM/WB register and write it back to the register file in the middle of the datapath.



PIPELINED DATAPATH

In the previous datapath we can notice that every element is used in only one pipelined stage. If this wasn't the case ,we'd have a structure hazard.

so, in order to overcome this issue we add redundancy to our datapath in order to gain the speed improvements from pipelining.

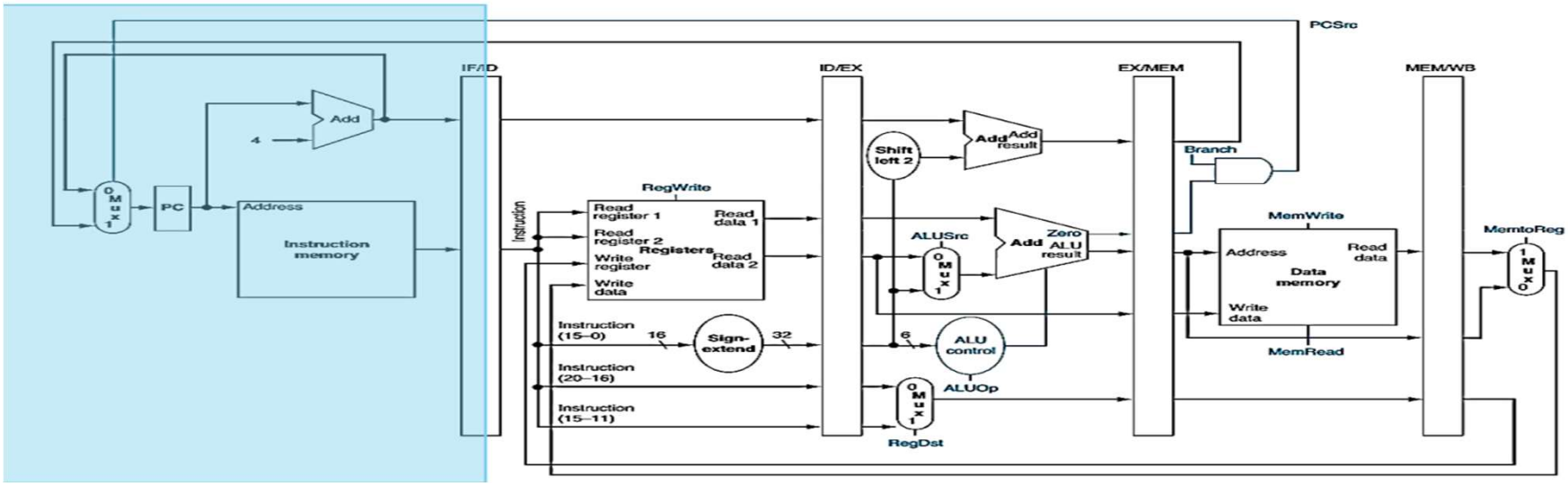
PIPELINED CONTROL

This was our control lines in the single cycle process

Instruction operation	Funct field	Desired ALU action	ALU control input
load word	XXXXXX	add	0010
store word	XXXXXX	add	0010
branch equal	XXXXXX	subtract	0110
add	100000	add	0010
subtract	100010	subtract	0110
AND	100100	and	0000
OR	100101	or	0001
set on less than	101010	set on less than	0111

PIPELINED CONTROL:IF

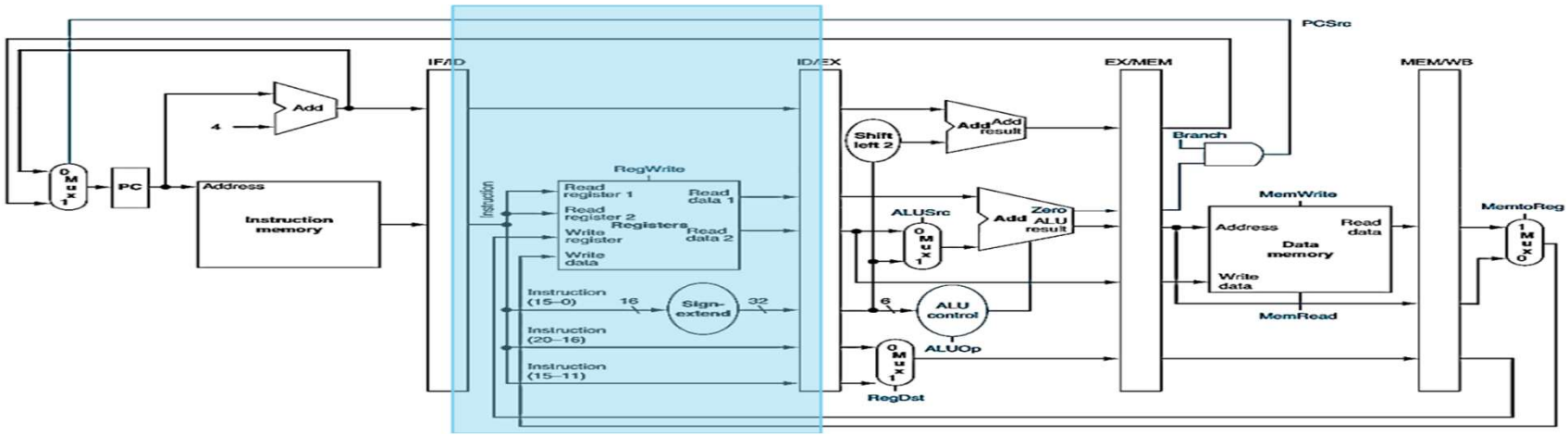
The usage of control signal in this stage is to read the instruction memory and write to PC



PIPELINED CONTROL:ID

As in the previous stage, the same actions take place during every cycle so we do not have to be concerned with any optional control lines.

the RegWrite signal physically resides in this component of the datapath but it is a control line of the WB stage.

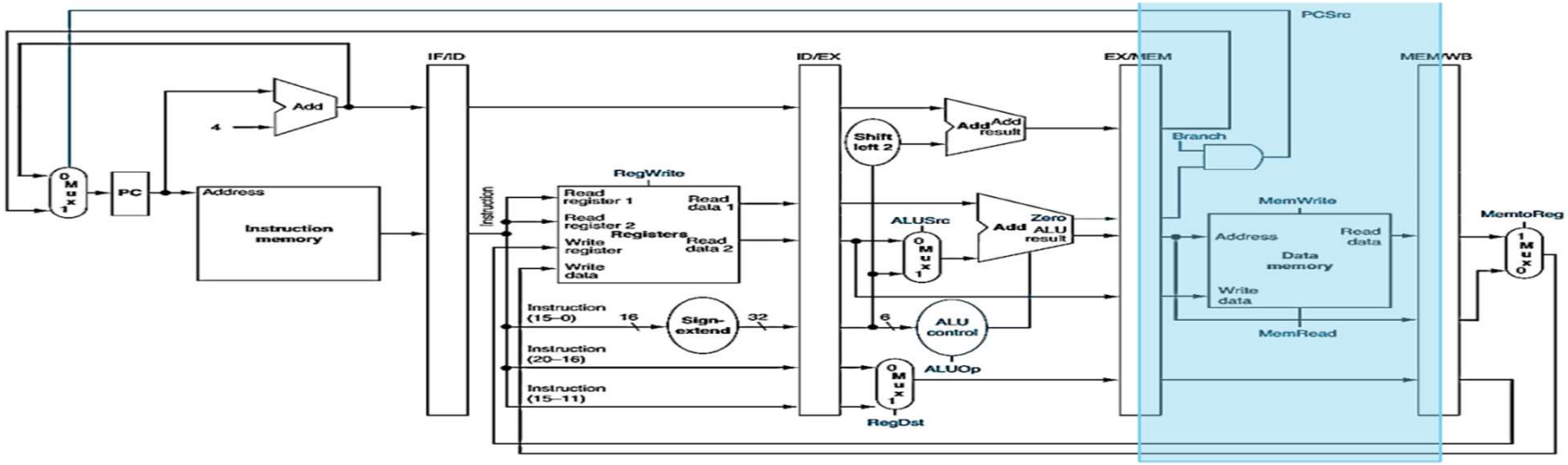


PIPELINE CONTROL: MEM

The relevant control signals in this stage are:

- Branch
- MemRead
- MemWrite

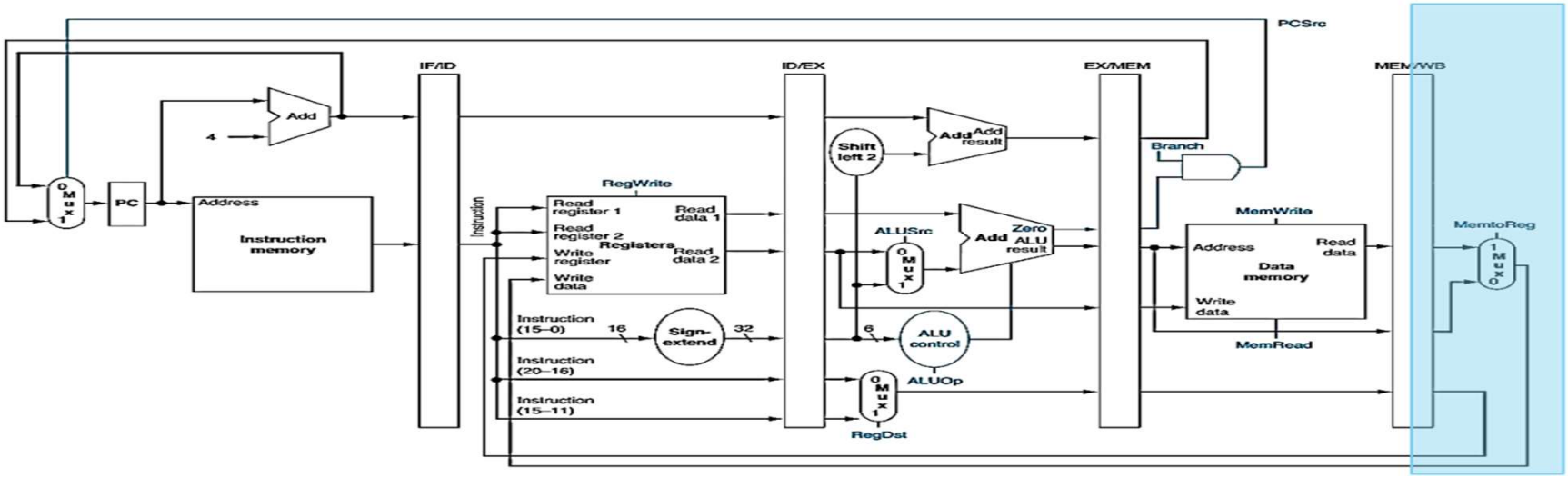
PCSrc is also computed in this stage but it is a product of two other control signals, so we don’t worry about it when designing our Control unit.



PIPELINE CONTROL: WB

The relevant control signals in this stage are:

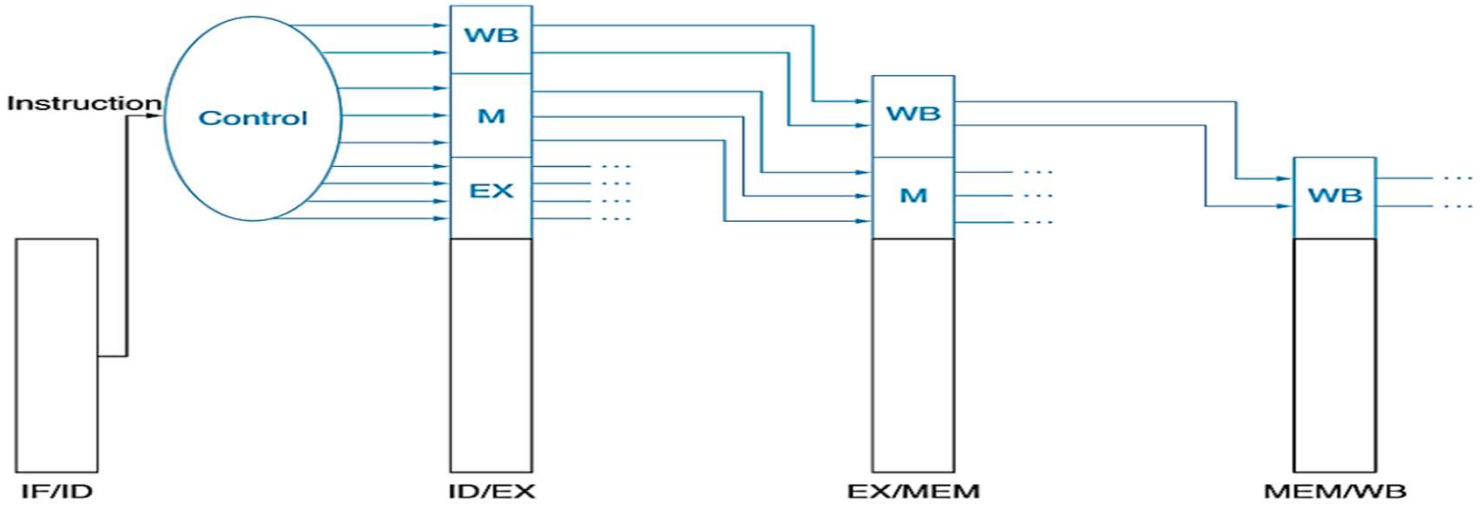
- MemtoReg
- RegWrite



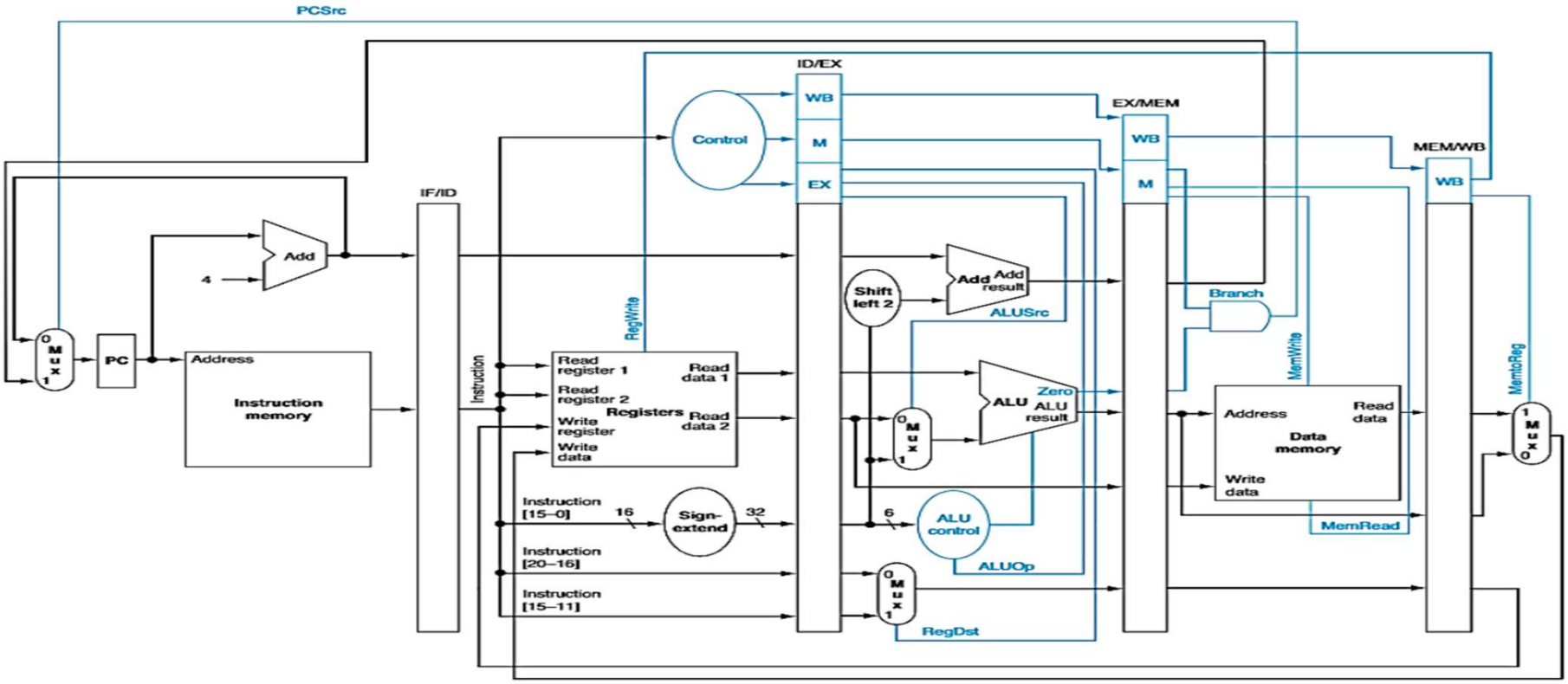
PIPELINED CONTROL

we can’t reach back the datapath for information about how to execute the instructions. it will have already been overwritten.

So we must pass the control line values for the last three stages



this is our datapath after implementing the control unit



PIPELINED Hazards

The hazards of the pipelined processor weren't included in the previous datapath as we have:

Data hazard: is when the instruction is unable to continue the cycle because of dependency of some data that hasn't been committed.

Control hazard: when we do not know which instruction needs to be executed next.

DATA HAZARDS

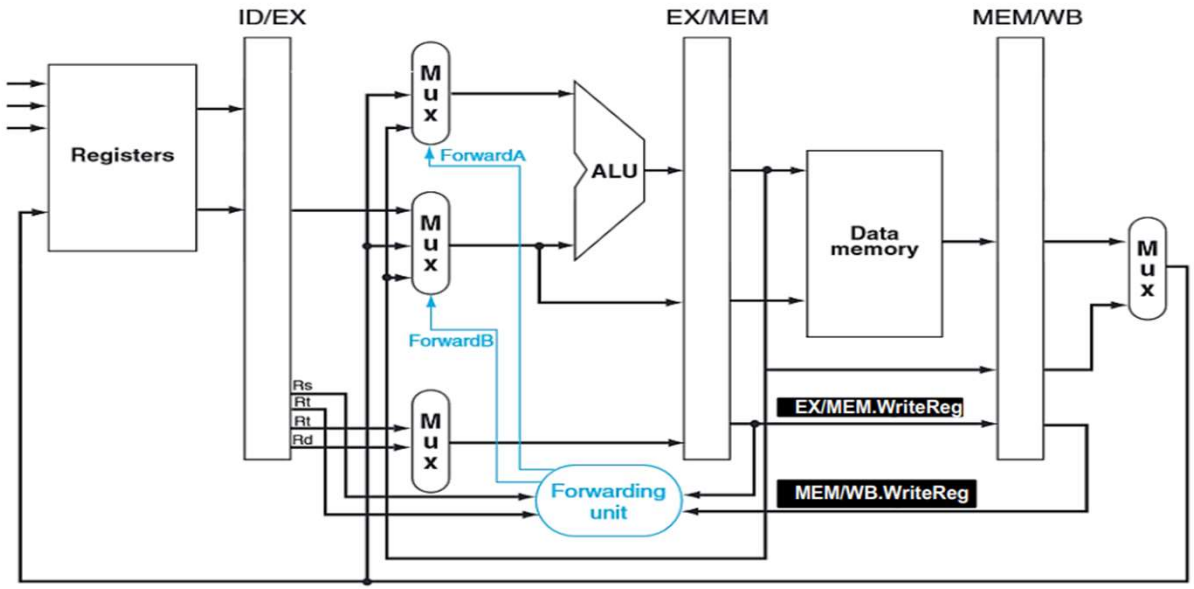
We have two solutions to overcome the problem of the data hazard:

Forwarding: the needed data is forwarded as soon as possible to the instruction which depends on it.

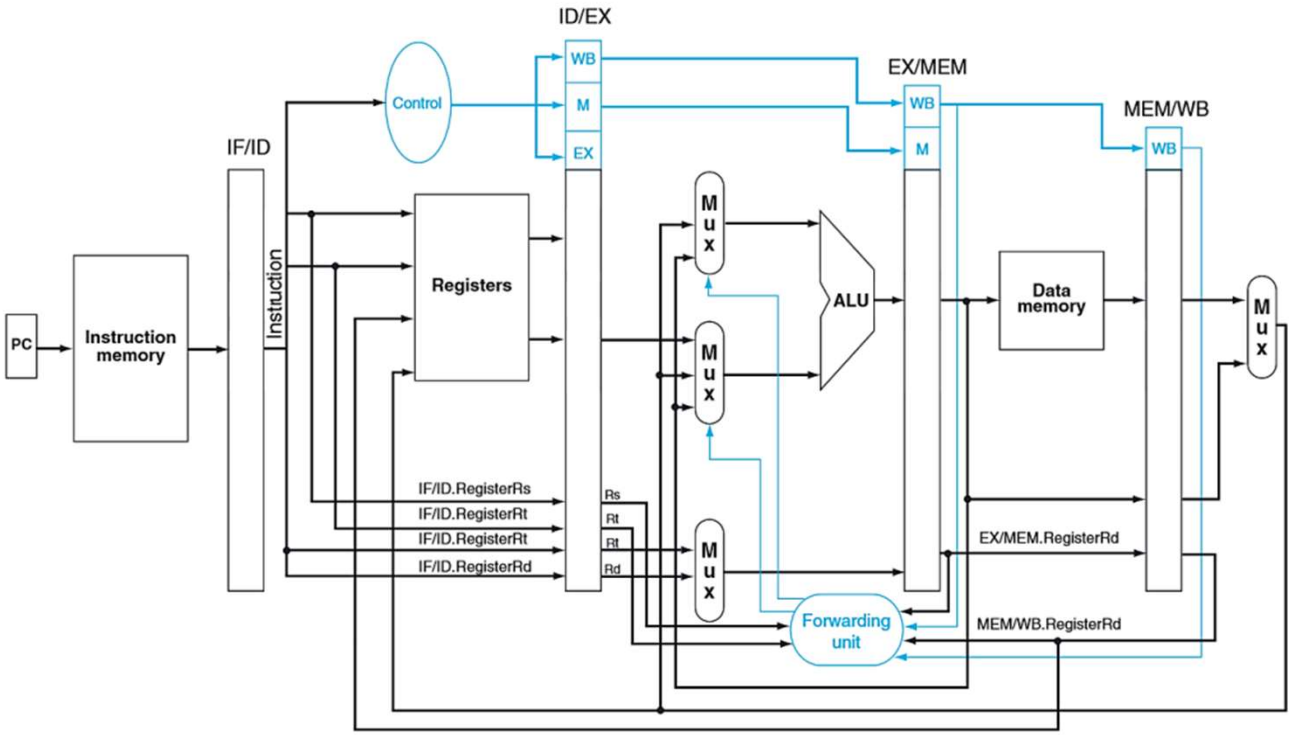
Stalling: the dependent instruction is “pushed back” for the needed amount of clock cycles. It’s like a nop for one or more clock cycles.

DATA HAZARDS

We will implement a forwarding unit instead of directly piping the ID/EX pipeline register values into the ALU, we now have multiplexors that allow us to choose where the input should come from.



DATAPATH/CONTROL WITH FORWARDING

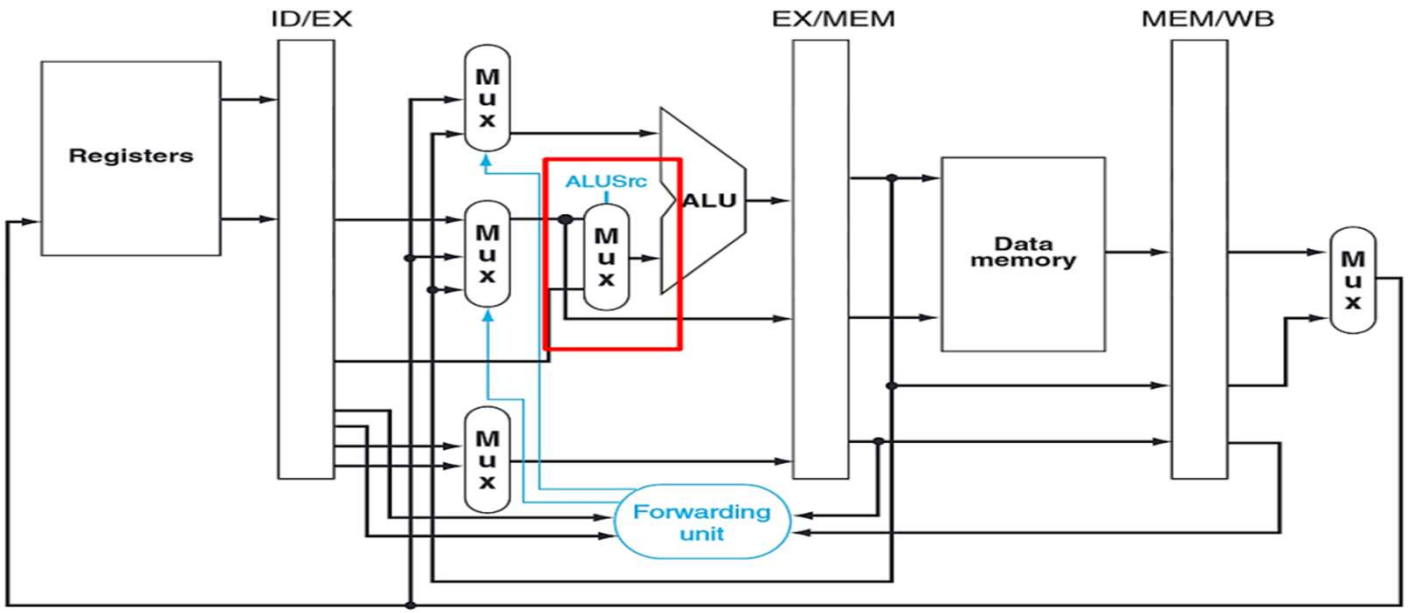


FORWARDING UNIT CONTROL VALUES

Mux Control	Source	Explanation
ForwardA = 00	ID/EX	First ALU input comes from register file.
ForwardA = 10	EX/MEM	First ALU input is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	First ALU input is forwarded from the data memory or a prior ALU result.
ForwardB = 00	ID/EX	Second ALU input comes from register file.
ForwardB = 10	EX/MEM	Second ALU input is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	Second ALU input is forwarded from the data memory or a prior ALU result.

DATAPATH/CONTROL WITH FORWARDING

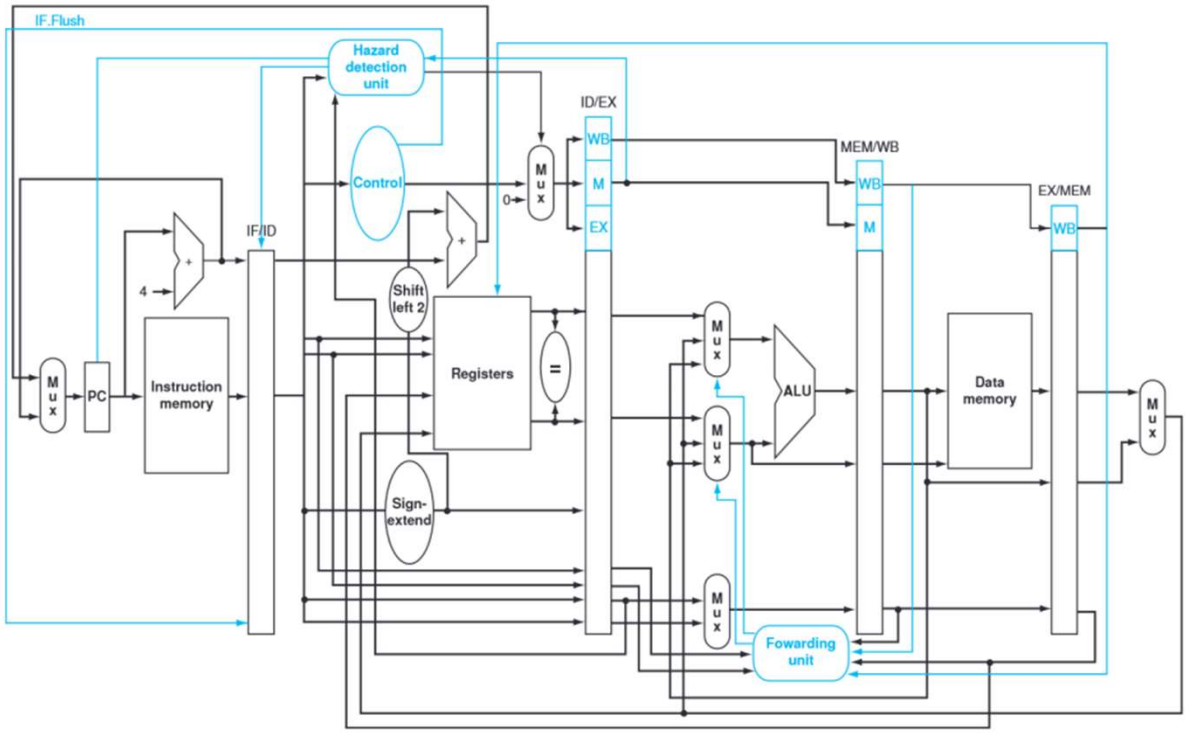
we will need an additional mux before the alu to decide between the rt register contents and the sign-ext immediate value.



DATA HAZARDS AND STALLS

Stalls is implemented by basically inserting an instruction which does nothing. To perform a stall we have the control unit output all nine control lines as 0’s.

FINAL DATAPATH AND CONTROL



Design

Starting with the standalone modules

Design procedure is commented in code

Disclaimer

The following design codes are inserted in the slides as pictures to look more representable

Adder

adder.v

```
1   //adder module for doing the branch and PC incrementation for the system
2
3   module adder ( a, b, y);
4
5   input [31:0] a,b;  //a and b are the input to be added
6   output [31:0] y;   //Y is the result
7
8   assign y=a+b;      //assigning of addition
9
10  endmodule
```

General ALUcontrol signal definition (4-bits)

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

Instructions ALU needed operations

Instruction operation	Funct field	Desired ALU action	ALU control input
load word	XXXXXX	add	0010
store word	XXXXXX	add	0010
branch equal	XXXXXX	subtract	0110
add	100000	add	0010
subtract	100010	subtract	0110
AND	100100	and	0000
OR	100101	or	0001
set on less than	101010	set on less than	0111

ALU

alu.v

```
1   //ALU module for the instructions we implemented in our system
2   //The ALU control bits are based on the general table in the text book
3   //The table will be found somewhere around this page
4
5   `timescale 1ps/1ps
6   //for adjusting of delay
7   //where it means {delay time unit/delay rounding}
8   //delay is adjusted by multiplying the delay time with the delay time unit
9   //and rounding of the output to the nearest delay rounding
10
11  module alu( input [31:0] rsdata,          //Inputs are the register file values (rs and/or rt)
12              input [31:0] rtdataOrextimm,  //or the immediate instruction incase of I-type
13              input [3:0] ALUctrl,          //ALU control from the control unit to determine the fucntion to be done
14              output reg [31:0] ALUResult,  //result of ALU, either address for data memory or data result
15              output reg zero);             //the zero for branching determination
16
17
18  always @ (*) begin
19      #100; //delay time remove for a proper alutestbench result
20      case (ALUctrl)
21          4'b0000: ALUResult = rsdata & rtdataOrextimm;                          // AND 0
22          4'b0001: ALUResult = rsdata | rtdataOrextimm;                          // OR  1
23          4'b0010: ALUResult = rsdata + rtdataOrextimm;                          // ADD 2
24          4'b0110: ALUResult = rsdata - rtdataOrextimm;                          // SUB 6
25          4'b0111: ALUResult = $signed(rsdata) < $signed(rtdataOrextimm) ? 1 : 0;   // SLT 7
26      endcase
27          zero = (ALUResult==8'b0);              //assigning of branching zero,
28                                                 //if ALUresult which is the sub function (diffrence)
29                                                 //is zero then they are equal so zero = 1, otherwise 0.
30
31          end
32  endmodule
```

Control signal: MemRead Modification

- Looking at the 2 control signals MemRead & MemtoReg
- We see there is a redundancy and they only both are equal to 1 if we encounter a load word instruction
- We can simplify the design by making MemRead always equal to one, hence always read the memory
- And have to value ready to be fetched at the MemtoReg mux, if there is a load word instruction MemtoReg will be equal to 1 and we will read data from memory

Operation	MemRead	MemWrite	MemtoReg
R-type	0	0	0
ADDI	0	0	0
SLTI	0	0	0
ANDI	0	0	0
ORI	0	0	0
XORI	0	0	0
LW	1	0	1
SW	0	1	X
BEQ	0	0	X
BNE	0	0	X

ALU control merge into main control

- The whole ALU control seems like a redundant component, as we enter the function and 2 signals that define each instruction, ALUop to take a 4 bit ALUcontrol signal to tell the ALU what operation to do
- Instead, we can enter the function into the main control unit and have it define the 4-bit ALUcontrol signal for us and directly take it into the ALU
- This can be done using a nested case statement for the R-type instructions and for anything that needs a special ALU operation
- Further explanation of what was done to code is in the next slides

Control

ctrlmain.v

```
1   //the main control unit along with the ALU control implemented in it,
2   //check the further shortcut notes for more understanding
3
4   module ctrlmain(input [5:0] Opcode,       //we will inoput the opcode, function and
5                   input [5:0] Func,         //and the zero for brnaching to determine control signals
6                   input Zero,
7               output reg  [3:0] ALUctrl,//all outputs are the control signals of
8                   output reg MemtoReg,      //the whole system
9               output reg  Jump,
10                  output reg  MemWrite,
11              output reg  RegDst,
12                  output reg  ALUSrc,
13                  output reg  RegWrite,
14                  output PCSrc              //PCSrs is the output of the and gate after determining
15                                            //if we will branch
16              //output[6:0] tempp for ctrltestbench
17                  );
18
19  // we have inputs as the opcode, fuction and the zero branch element
20  //then the poutputs are the 7 control signals and the ALUcontrol
21
22  //the OP code and function values are based on the MIPS greensheet
23  //ALUcontrol signal is explained in ALU
24
25  reg[6:0] temp; //for defining the control signal values later
26  reg Branch;    //internal register for branching and determining if we branch or not
27
```

Control

continued

```
28  always @(*) begin
29
30      case (Opcode)
31          6'b000000: begin                            // R-type 0
32                      temp <= 7'b1100000;             //96
33
34                      case (Func)
35                      6'b100000: ALUctrl <= 4'b0010;  // ADD 32 2
36                      6'b100010: ALUctrl <= 4'b0110;  // SUB 34 6
37                      6'b100100: ALUctrl <= 4'b0000;  // AND 36 0
38                      6'b100101: ALUctrl <= 4'b0001;  // OR  37 1
39                      6'b101010: ALUctrl <= 4'b0111;  // SLT 42 7
40                  endcase
41              end
42
43          6'b100011: begin                            // LW 35
44                      temp <= 7'b1010010;             //82
45                      ALUctrl <= 4'b0010;
46                  end
47
48          6'b101011: begin                            // SW 43
49                      temp <= 7'b0010100;             //20
50                      ALUctrl <= 4'b0010;
51                  end
52
53          6'b000100: begin                            // BEQ 4
54                      temp <= 7'b0001000;             //8
55                      ALUctrl <= 4'b0110;
56                  end
```

Control

continued

```
56                  end
57
58          6'b001000: begin                            // ADDI 8
59                      temp <= 7'b1010000;             //80
60                      ALUctrl <= 4'b0010;
61                  end
62
63          6'b000010: begin                            // JUMP 2
64                      temp <= 7'b0000001;             //1
65                      ALUctrl <= 4'b0000;
66                  end
67      default:   temp <= 7'bxxxxxxx;                // NOP
68      endcase
69
70
71
72      {RegWrite,RegDst,ALUSrc,Branch,MemWrite,MemtoReg,Jump} = temp; //setting the control signals
73
74  end
75
76  assign PCSrc = Branch & (Zero); //AND gate output is PCSrc, for determining
77                                  //if we are branching or not
78
79  //assign tempp=temp; for ctrltestbench
80
81  endmodule
```

Data Memory

dmem.v

```
1   //Data memory module
2
3   `timescale 1ps/1ps
4   //for adjusting of delay
5   //where it means {delay time unit/delay rounding}
6   //delay is adjusted by multiplying the delay time with the delay time unit
7   //and rounding of the output to the nearest delay rounding
8
9
10  module dmem(clk,MemWrite,ALUResult,din,dout);
11
12  //we here have defined the data memory to be a 2D array
13  //which we can address each instruction by its row number
14  //where its row number will be the address of the memory unit
15  //the address is the cakculated ALU result
16
17
18
19  parameter depth =128;
20  parameter width = 32;
21
```

Data Memory

continued

```
22   input clk, MemWrite;
23   input [31:0] ALUResult;
24   input [width-1:0] din;            //data into the Dmem
25   output [width-1:0] dout;          //data out of the Dmem
26
27   reg [width-1:0] Dmem [depth-1:0]; //initialization of the 2D array
28
29   assign dout = Dmem[ALUResult];    // as there is no MemRead we will always
30                                     //read and hold the read value
31
32   always @ (posedge clk) begin      //MemRead permission for writing the memory
33       #500; //delay time
34
35       if (MemWrite)       //writing of the data input to the data memory
36       Dmem[ALUResult] <= din;
37
38   end
39
40   endmodule
```

Instruction memory

imem.v

```
1   //instruction memory module
2
3   module imem(PC,Inst);
4
5   //we here have defined the instruction memory to be a 2D array
6   //which we can address each instruction by its row number
7   //where its row number will be the address of each instruction
8   //the address is the calculated ALU result
9
10
11
12  parameter depth =256;
13  parameter width = 32;
14
15
16  input [31:0] PC;
17  output [width-1:0] Inst;
18
19  reg [width-1:0] Imem[depth-1:0];     //initialization of 2D array
20
21  initial
22      $readmemh("memfile.txt", Imem);  //reads the instruction as HEX from
23                                       //from a textfile and inputs the hex as binary values
24                                  //inside the Imem row by row with a max of 32 bits defined up
25
26    assign Inst= Imem[PC/4];           //The PC is incremented by 4 while the instruction memory has
27                                       //values starting from 1 so we revert by dividing PC by 4 to
28                                       //address correctly
29
30
31
32  endmodule
```

Multiplexer

mux.v

```
1   //Multiplexers module
2
3   module mux(inp0,inp1,signal,selectedinp);
4
5   parameter n=32;
6
7   input [31:0] inp0;   //the 2 inputs that we will be selecting from
8   input [31:0] inp1;
9   input signal;        //the control signal for determing the output signal
10  output [31:0] selectedinp;  //output
11
12  assign selectedinp = signal ? inp1 : inp0;  //if statement for determing the
13                                              //selected input based on signal
14
15  endmodule
```

Program Counter

pc.v

```
1   //program counter module
2
3   `timescale 1ps/1ps
4   //for adjusting of delay
5   //where it means {delay time unit/delay rounding}
6   //delay is adjusted by multiplying the delay time with the delay time unit
7   //and rounding of the output to the nearest delay rounding
8
9
10
11  module PC (clk, rst, PC,PCNext);
12
13  input clk,rst;
14  input [31:0] PCNext;   //PCnext is the next value of PC
15  output reg [31:0] PC;  //PC is the current value of PC
16
17  always @ (posedge clk) //clock and reset always block
18  begin
19      if(!rst) PC<=PCNext;  //on initial run where reset=1, we will set PC to zero,
20      else PC<=0;           //otherwise when clock signal is input and reset=0,
21                            //we will set the PC
22                            //to the next value
23
24  #500; //delay time of instruction memory is included here as PC is directly
25        //related to instruction memory
26
27  end
28
29  endmodule
```

Register file

regfile.v

```
1   //Register file module
2
3   `timescale 1ps/1ps
4   //for adjusting of delay
5   //where it means {delay time unit/delay rounding}
6   //delay is adjusted by multiplying the delay time with the delay time unit
7   //and rounding of the output to the nearest delay rounding
8
9
10
11  module regfile (input clk,
12                  input RegWrite,
13                  input reset,
14                  input [4:0] rs, //address of rs
15                  input [4:0] rt, //address of rt
16                  input [4:0] RdOrRt, //address of rd or rt based on instruction type
17                  input [31:0] WriteData, // data to be input to registers
18                  output [31:0] rsdata, //output data of registers
19                  output [31:0] rtdata);
20
21  //we here have defined the register file to be a 2D array
22  //which we can address each register by its row number
23  //where its row number will be the address of each register
24  //the 2D array is 32 bits wide for the register file max size
25
26
27
28  reg [31:0] register [31:0]; //2D array initialization
29
30  assign rsdata = register[rs]; //assigning of the registers data to an output
31  assign rtdata = register[rt]; //in case we need the register contents
32
33  //No delay time as we have no RegrRead signal
```

Register file

continued

```
34
35
36  integer i;
37
38  initial begin //initialization of all registers to be default by zero
39      for (i=1; i<32; i=i+1) begin
40          register[i] <= 32'd0;
41          end
42      end
43
44  always @(posedge clk) //clock and resit always block
45
46  begin
47      register[0]=0;
48      if(reset) for(i = 0; i < 32; i = i + 1) register[i] = 32'd0; // if reset=1
49                                                                   //reset the whole reg file
50
51      else if (RegWrite) //permission to write the register
52          if(RdOrRt != 0) register[RdOrRt]= WriteData; //if you are wondering what happens of both reset and regwrite =1,
53                                                       //and thats the case for the first run incase we encounter an immediate instruction
54                                                       //that is explained in the next slide and how these 2 if statements merge
55      #200; //Delay time
56  end
57
58  endmodule
```

Shift left

shiftl2.v

```
1   //shift left module which multiplies
2   //the input value by 4
3
4   module shiftl2(input [31:0] extdimm, //input value
5                  output [31:0] extdimmt4);//output value times 4
6
7   assign extdimmt4 = extdimm << 2; //using the shift operator to shift
8
9   endmodule
```

Sign Extension

signext.v

```
1   //sign extend module, this takes the leftmost sign
2   //of the input and extends it to 32 bits
3
4   module signext( input  [15:0] imm,    //16-bit input
5                   output [31:0] extdimm //32-bit extended output
6                   );
7
8       assign extdimm = { {16{imm[15]}} , imm };
9                   //taking of the left most bit and extending
10  endmodule
```

Wiring and top module design

- Now that we have all our standalone components done we need to connect everything together in our top module
- Our strategy will be as follows
  - Start from the instruction memory and move along all modules
  - Check the inputs of each module and from there see which module should the input be output from, move to this instruction and try to fetch its inputs as well
  - We do this till be end the loop and return to the instruction memory again
  - When we encounter a module output we already checked it before leave it as then the data does not need to flow in order
  - This will be a continuous flow inputs and sorting where it will be output from.

Wiring and top module design

continued

- Instruction memory
  - `imem(PC,Instr)`
    - Inst is output
    - PC is input from PC module and we need to adjust the following:
      - Adjusting of starting PC
      - Making PC increments
      - Jump and branch MUXs

Wiring and top module design

continued

- Program counter
  - `PC(clk, reset, PC, PCNext)`
    - Clock, reset and PC are output
    - PCnext is input from the Jump and branch MUXs
  - `adder(PC, 32'd4 ,PCplus4)`
- Branch
  - `signext(Instr[15:0], extendedimm)`
  - `shiftl2(extendedimm, extendedimmafter);`
  - `adder(extendedimmafter, PCplus4, PCbeforeBranch)`
  - `branchmux(PCplus4 , PCbeforeBranch, PCSrc, PCBranch)`

Wiring and top module design

continued

- Jump
  - `jumpmux(PCBranch, {PCplus4[31:28], Instr[25:0], 2'b00}, Jump,PCNext)`
    - But for the jumpmux and branchmux we need the control signals, so we need to fetch the control unit
- Control
  - `ctrlmain(Instr[31:26], Instr[5:0], ZeroFlag,ALUControl,MemtoReg,Jump,MemWrite, RegDst, ALUSrc, RegWrite, PCSrc)`
    - We have Instr as input, fetched before in instruction memory
    - ALUControl, MemtoReg, Jump, MemWrite, RegDst, ALUSrc, RegWrite, PCSrc are output
    - The Zeroflag is input from ALU so we will fetch ALU next

Wiring and top module design

continued

- ALU
  - `alu(rsdata, rtdataOrextimm, ALUControl, ALUResult, ZeroFlag)`
    - ALUcontrol is input from control before
    - ALUresult and Zeroflag are outputs
    - rtdataOrextimm is from mux which we should fetch
    - Rsdata is from register file so we will fetch register file
  - `ALUSrcmux(rtdata,extendedimm, ALUSrc, rtdataOrextimm)`
    - We have extendedimm, ALUSrc from branch and control respectively
    - We need rtdata from register file

Wiring and top module design

continued

- Register file
  - `regfile(clk,RegWrite, reset, Instr[25:21], Instr[20:16], RdOrRt, WriteData, rsdata,rtdata)`
    - Clock, RegWrite, Reset, Instr are input from control and instruction memory
    - Rsdata and Rtdata are outputs
    - RdorRt is input from mux which we will fetch
    - WriteData is input from mux which we will fetch
  - `RegDstmux(Instr[20:16],Instr[15:11],RegDst, RdOrRt)`
    - Inst, RegDst are input in instruction memory and control respectively
  - `MemtoRegmux(ALUResult, dout, MemtoReg,WriteData)`
    - ALUresult,MemtoReg is input from ALU and control
    - Dout is input from data memory which we will fetch now

Wiring and top module design

continued

- Data memory
  - `dmem(clk,MemWrite,ALUResult, rtdata, dout)`
    - Clock, Memwrite, ALUresult, rtdata are inputs from control, ALU and register file respectively
    - Dout is the output we need for the register file MUX so we will return to the MUX
- Now that we have dout we return to the write data mux
  - `MemtoRegmux(ALUResult, dout, MemtoReg,WriteData)`
- Now that we have WriteData and RdorRt we can return to register file
  - `regfile(clk,RegWrite, reset, Instr[25:21], Instr[20:16], RdOrRt, WriteData, rsdata,rtdata)`
- Now that we have rsdata and rtdata we can return to the ALU
  - `alu(rsdata, rtdataOrextimm, ALUControl, ALUResult, ZeroFlag)`

Wiring and top module design

continued

- Now that we have zeroflag we return to the control
  - `ctrlmain(Instr[31:26], Instr[5:0], ZeroFlag, ALUControl, MemtoReg, Jump, MemWrite, RegDst, ALUSrc, RegWrite,PCSrc)`
- Now that we have the control signals we can return to the branch and jump MUXs
  - `jumpmux(PCBranch, {PCplus4[31:28], Instr[25:0], 2'b00}, Jump,PCNext)`
- Now that we have Pcnext we can return to the PC module
  - `PC(clk, reset, PC, PCNext)`
- Now that we have PC we can return to our starting point, the instruction memory
  - `imem(PC,Instr)`
- Like this we have completed out wiring stages and have done our top module

Top module

Click here for wiring procedure

top2.v

```
1   //top module for wiring everything together
2
3   `include "adder.v"
4   `include "alu.v"
5   `include "pc.v"
6   `include "mux.v"
7   `include "regfile.v"
8   `include "signext.v"
9   `include "shiftl2.v"
10  `include "dmem.v"
11  `include "imem.v"
12  `include "ctrlmain.v"
13
14  //including of all modules to define them later for wiring in
15  //this top module
16
17  //check the design procedure for wiring the components toghether
18
19  module top(input clk,
20             input reset,
21            //Outputs for testbench
22            output [31:0] PCC,
23            output[31:0] WriteDataa,
24            output[4:0] RdOrRtt,
25            output [31:0] doutt,
26            output[31:0] dinn
27            );
28
29  //all components are wired as we have no geneal input or output
30  //as all components are output from one place and input to another
31  //simultneously across the loop everytime
```

Top module

continued – click here for wiring procedure

```
33  wire [31:0] PCNext, PCplus4, PCbeforeBranch, PCBranch, ALUResult,dout,rtdata,PC,Instr,
34              extendedimm, extendedimmafter, WriteData, rsdata, rtdataOrextimm;
35  wire [4:0] RdOrRt;
36  wire[3:0] ALUControl;
37  wire RegDst,RegWrite,ALUSrc,Jump,MemtoReg,PCSrc,MemWrite,ZeroFlag;
38
39  //Outputs for testbench
40  assign WriteDataa=WriteData;
41  assign RdOrRtt=RdOrRt;
42  assign dinn=rtdata;
43  assign doutt=dout;
44  assign PCC=PC;
45
46  // PC
47
48  PC PCmod(clk,reset, PC,PCNext); //calling of our PC module to determing our first PC which will be zero
49  adder pcadd4(PC, 32'd4 ,PCplus4);//first step is doing PC=PC+4
50  shiftl2 shiftl2(extendedimm,extendedimmafter);//next is extending the branch-imm value to 32 for use in adder, this is the 'A' value
51  adder pcaddsigned(extendedimmafter,PCplus4,PCbeforeBranch);//determining the branching address of there is one
52  mux #(32) branchmux(PCplus4 , PCbeforeBranch, PCSrc, PCBranch);//what we did was (PC+4)+(4*A), which gives us the branching address
53  mux #(32) jumpmux(PCBranch, {PCplus4[31:28],Instr[25:0],2'b00 }, Jump,PCNext);
54  //for jumping we take concatinationn of the first 4 bits of PC along with the 25 jump bits and multiply it by 4
55  //to determing the jump address (4*jump)
56
57
58  // Register File
59
60  regfile regfile(clk,RegWrite, reset, Instr[25:21], Instr[20:16], RdOrRt, WriteData, rsdata,rtdata);
61  mux #(5) RegDstmux(Instr[20:16],Instr[15:11],RegDst, RdOrRt);
62  mux #(32) MemtoRegmux(ALUResult, dout, MemtoReg,WriteData);
63  //setting the register file inputs along with its control signals
64  //setting up of the MUX for determing of I or R type instruction
65  //setting up of the mux for determing which data to be written into register
66  //if R or I type instruction
```

Top module

continued – click here for wiring procedure

```
69   // ALU
70
71   alu alu(rsdata, rtdataOrextimm, ALUControl, ALUResult, ZeroFlag);
72   signext immextention(Instr[15:0],extendedimm);
73   mux #(32) ALUSrcmux(rtdata,extendedimm, ALUSrc, rtdataOrextimm);
74   //setting up of the ALU inputs along with its output zero branching signal
75   //extension of immediate value to be used by register
76   //setting up the MUX to determing input value if we have R or I type instruction
77
78
79   //Data memory
80
81   dmem dmem(clk,MemWrite,ALUResult, rtdata, dout);
82   //pretty simple assigning of out datamemory element with rtdata as out input
83   //for store word immediate instructions and dout as output for load word
84   //immediate instructions
85
86
87   //Instruction memory
88
89   imem imem(PC,Instr);
90   //setting of instruction memory to input PC and give us corresponding instruction
91
92
93   //Control
94
95   ctrlmain ctrlmain(Instr[31:26], Instr[5:0] ,ZeroFlag,ALUControl,MemtoReg,Jump,MemWrite,
96                     RegDst, ALUSrc, RegWrite,PCSrc);
97   //control which we input OP and Fucntion and we get the corrsponding control signals
98
99   endmodule
```

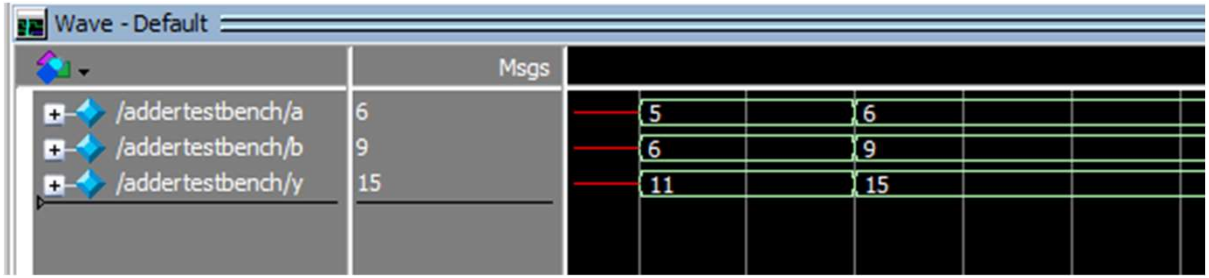
Testbench-Adder

addertb.v

```
//adder testbench

`timescale 1ps/1ps
//for adjusting of delay
//where it means {delay time unit/delay rounding}
//delay is adjusted by multiplying the delay time with the delay time unit
//and rounding of the output to the nearest delay rounding

module addertestbench;

    //Inputs
    reg[31:0] a;
        reg[31:0] b;

    //Outputs for testbench
    wire[31:0] y;

    //Instantiation of Unit Under Test
    adder uut (
        .a(a),
        .b(b),

        //Outputs for testbench
        .y(y)
           );

initial
begin
#10;
a=5;
b=6;

#10
a=6;
b=9;
end

endmodule
```

Testbench - Adder results

continued



Testbench - Control

continued

controltb.v

```
//control testbench

`timescale 1ps/1ps
//for adjusting of delay
//where it means {delay time unit/delay rounding}
//delay is adjusted by multiplying the delay time with the delay time unit
//and rounding of the output to the nearest delay rounding

module controltestbench;

        //Inputs
        reg [5:0] Opcode;
                reg [5:0] Func;
                reg Zero;


        //Outputs for testbench

                wire PCSrc;
        wire[3:0] ALUctrl;
        wire[6:0] tempp;


    //Instantiation of Unit Under Test
    ctrlmain uut (
        .Opcode(Opcode),
        .Func(Func),
        .Zero(Zero),

        //Outputs for testbench
        .PCSrc(PCSrc),
        .tempp(tempp),
        .ALUctrl(ALUctrl)
        );

initial
begin

#200; //ADD
Opcode=0;
Func=16;
Zero=0;

#200; //SUB
Opcode=0;
Func=18;
Zero=0;

#200; //AND
Opcode=0;
Func=20;
Zero=0;

#200; //OR
Opcode=0;
Func=21;
Zero=0;

#200; //SLT
Opcode=0;
Func=42;
Zero=0;

#200; //LW
Opcode=35;
Func=0;
Zero=0;

#200; //SW
Opcode=43;
Func=0;
Zero=0;

#200; //BEQ
Opcode=4;
Func=0;
Zero=1;

#200; //ADDI
Opcode=8;
Func=0;
Zero=0;

#200; //JUMP
Opcode=2;
Func=0;
Zero=0;


end

endmodule
```

Testbench – Control results

continued

Testbench - Multiplexer

continued

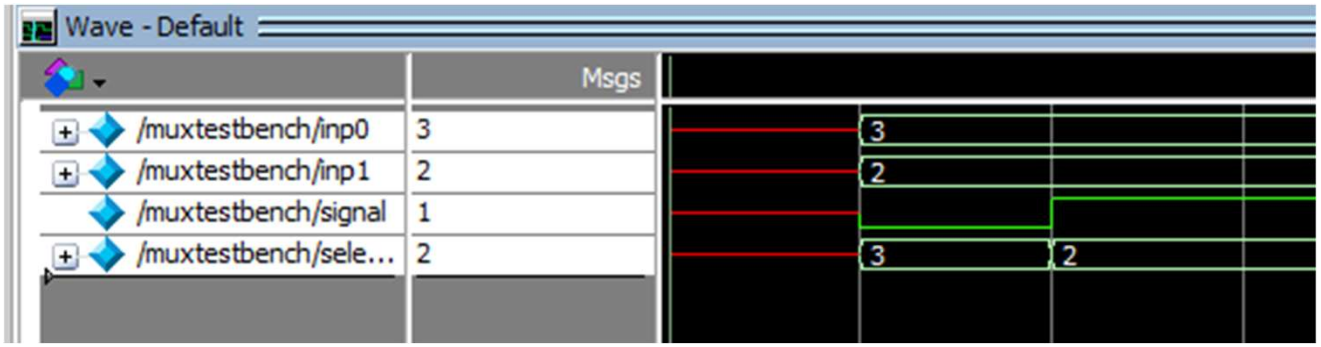
muxtb.v

```
1   //mux testbench
2
3   `timescale 1ps/1ps
4   //for adjusting of delay
5   //where it means {delay time unit/delay rounding}
6   //delay is adjusted by multiplying the delay time with the delay time unit
7   //and rounding of the output to the nearest delay rounding
8
9   module muxtestbench;
10
11      //Inputs
12      reg[31:0] inp0;
13          reg[31:0] inp1;
14      reg signal;
15
16      //Outputs for testbench
17      wire[31:0] selectedinp;
18
19
20      //Instantiation of Unit Under Test
21      mux uut (
22          .inp0(inp0),
23          .inp1(inp1),
24          .signal(signal),
25
26          //Outputs for testbench
27          .selectedinp(selectedinp)
28            );
```

```
30  initial
31  begin
32  #10;
33  inp0=3;
34  inp1=2;
35  signal=0;
36
37  #10;
38  inp0=3;
39  inp1=2;
40  signal=1;
41
42  end
43
44  endmodule
```

Testbench – Multiplexer results

continued



Testbench – Shiftleft

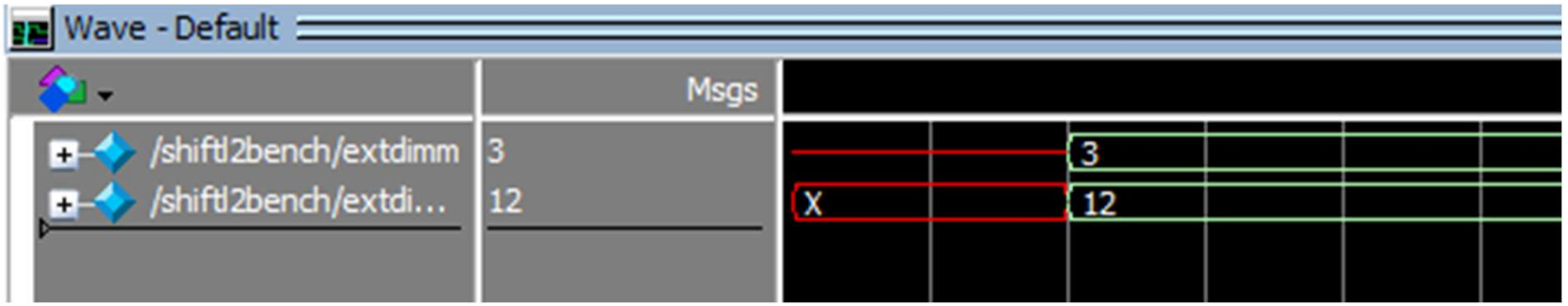
continued

shiftl2tb.v

```
1   //shift testbench
2
3   `timescale 1ps/1ps
4   //for adjusting of delay
5   //where it means {delay time unit/delay rounding}
6   //delay is adjusted by multiplying the delay time with the delay time unit
7   //and rounding of the output to the nearest delay rounding
8
9   module shiftl2bench;
10
11      //Inputs
12      reg[31:0] extdimm;
13
14      //Outputs for testbench
15      wire[31:0] extdimmt4;
16
17
18      //Instantiation of Unit Under Test
19      shiftl2 uut (
20          .extdimm(extdimm),
21
22          //Outputs for testbench
23          .extdimmt4(extdimmt4)
24            );
25
26  initial
27  begin
28  #10;
29  extdimm=3;
30  end
31  endmodule
```

Testbench – Shiftleft results

continued



Testbench – Sign extension

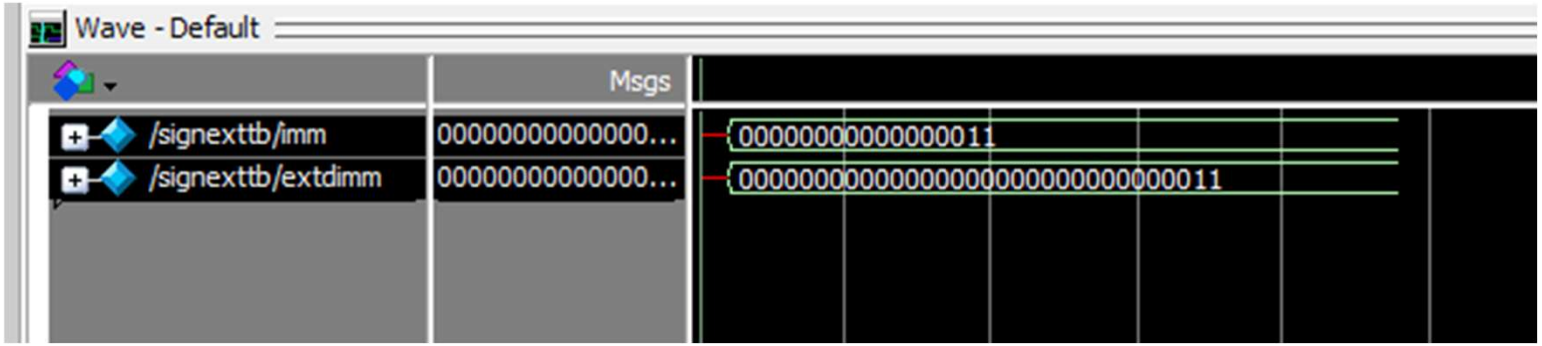
continued

signexttb.v

```
1   //signext testbench
2
3   `timescale 1ps/1ps
4   //for adjusting of delay
5   //where it means {delay time unit/delay rounding}
6   //delay is adjusted by multiplying the delay time with the delay time unit
7   //and rounding of the output to the nearest delay rounding
8
9   module signexttb;
10
11      //Inputs
12      reg[16:0] imm;
13
14      //Outputs for testbench
15      wire[31:0] extdimm;
16
17
18      //Instantiation of Unit Under Test
19      signext uut (
20          .imm(imm),
21
22          //Outputs for testbench
23          .extdimm(extdimm)
24              );
25
26  initial
27  begin
28  #10;
29  imm=3;
30  end
31  endmodule
```

Testbench – Sign extension results

continued



General testbench hand analysis

```
addi $5, $0, 3    (0)
beq  $0, $0, next (4)
jmp: add $6, $5, 0    (8)
ext: sw  $5, 4($5)    (12)
     lw  $7, 4($5)    (16)
     j   jump         (20)
```

→ PC: 0 , 4 , 12 , 16 , 20 , 8 , 12 , 16 , 20 ---

→ WriteData: Data getting written in Registers 32-bits

: 3 , 0 , 3+4=7 , 3 , 0 , 3 , 7 , 3 , 0 , 3 , 7 ---

(3+4=7: Base + offset)

→ Rs or Rt : Selection of I or R-type Registers

: 5 , 0 , 5 , 7 , 0 , 5 ---

General testbench hand analysis

continued

→ d_in (data into dmem): 0, 0, 3, 0, 0, 0, 3, —

→ d_out (data out of dmem): 0, 0, 0, 3, 0, 0, 3, —

✓ Agrees with test bench from modelsim

→ If this test-bench succedes then we have tested all components of :- PC, imem
- Branch
- Jump
- dmem

Testing file and encoding

This text file contains our instructions in hex format ready to be imported into the instruction memory, check instruction memory for more illustration of what happens.

memfile.txt

```
0 addi $5,$0,3          // 5 has 3
4 beq $0, $0, next      // true
8 jump: add $6, $5, $0  // never reached
12 next: sw $5,4($5)    // sw from reg5 in memory
16 lw $7, 4($5)         // lw from memory to reg 7
20 j jump               // 8/4=2


20050003
10000001
00a03020
aca50004
8ca70004
08000002
```

Testbench – PC, Register file, Data memory, Instruction memory

continued – [for clock period calculations click here]

testbench.v

```
1   //test bench module for testing of the main intructions we have
2   //and monitoring of the hand analysis elements
3   //this testbench tests everything that needs a clock, which are
4   //Instruction memory,Data memory, PC, and instruction memory together
5   //we input or use the instrictions in our memfile.txt
6
7   `timescale 1ps/1ps
8   //for adjusting of delay
9   //where it means {delay time unit/delay rounding}
10  //delay is adjusted by multiplying the delay time with the delay time unit
11  //and rounding of the output to the nearest delay rounding
12
13  module testbench;
14
15      //Inputs
16      reg clk;
17          reg reset;
18
19      //Outputs for testbench
20      wire[31:0] PCC;
21      wire[31:0] WriteDataa;
22      wire[4:0]RdOrRtt;
23      wire[31:0] dinn;
24      wire[31:0] doutt;
25
26
27      //Instantiation of Unit Under Test
28      top uut (
29          .clk(clk),
30          .reset(reset),
31
32          //Outputs for testbench
33          .PCC(PCC),     //for instruction memory and PC
34          .WriteDataa(WriteDataa), //for register file test
35          .RdOrRtt(RdOrRtt), //for register file test
36          .dinn(dinn), //for register file and data memory test
37          .doutt(doutt) //for register file and data memory test
38      );
```

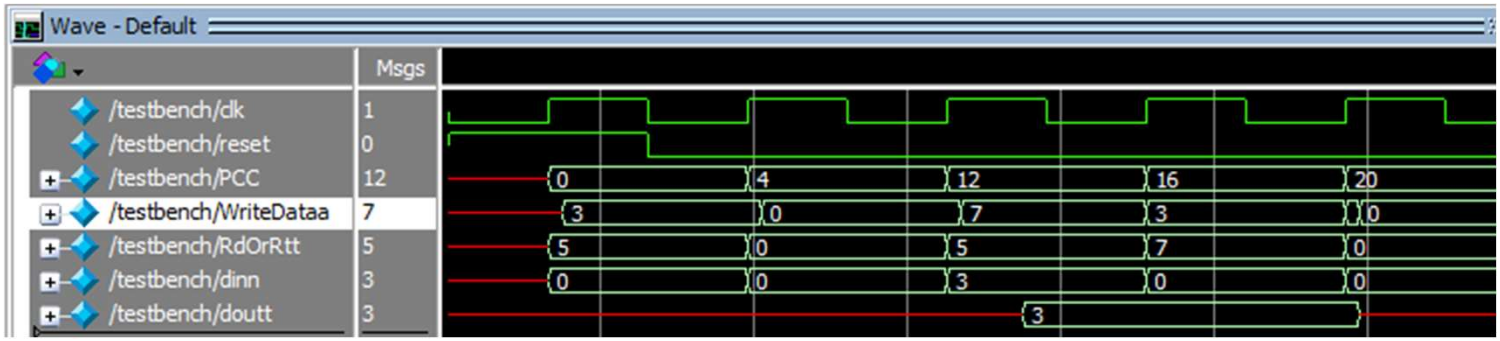
Testbench – PC, Register file, Data memory, Instruction memory

continued – [for clock period calculations click here]

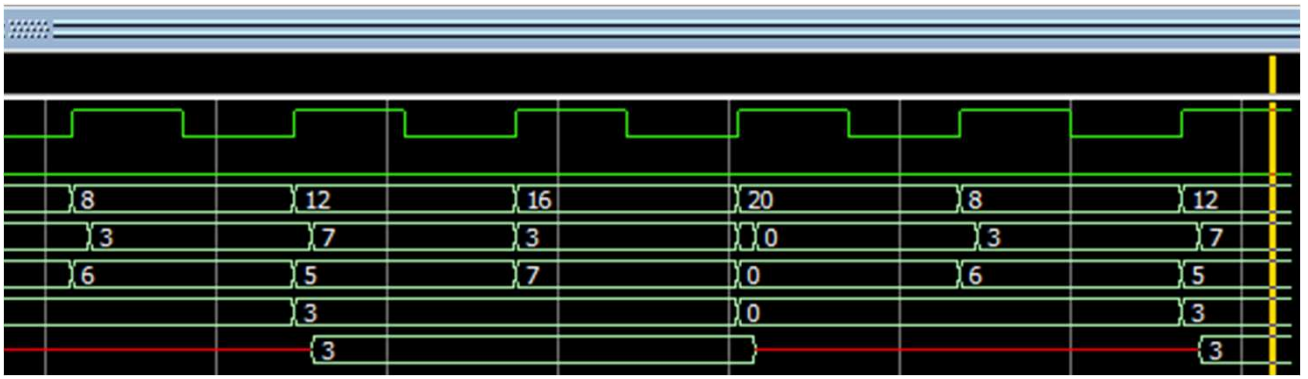
```
39
40      //our clock period is 1300ps based on the slowest instruction
41      //calculated in the next slide
42
43      always #650 clk=!clk; //adjustment of a general clock with half cycle=clock cycle/2
44              //reset=1 for the first loop
45      initial begin
46
47          $dumpfile("dump.vcd"); $dumpvars;
48
49      clk=0;
50      reset=1;
51      #1300; //reset=1 for the first clock cycle
52
53        repeat(10) begin
54      reset=0;//reset=0 for the rest cycles
55      #1300;
56        end
57          $finish; //we will have 11 clock cycles assuming we have
58          //max of 11 instructions for our program to terminate
59          //the instruction set I made has endless number of instructions
60          //so I used finish to terminate my program
61          //if we need more clock cycles set the 'repeat' value above higher
62      end
63
64  endmodule
```

Testbench – PC, Instruction memory, Register file, Data memory results

continued



PCC – for testing PC and instruction memory
WriteDataa - To test Register file
dinn - to test data memory
doutt - to test data memory



Clock period Calculation

Instruction Class	Instruction memory	Instruction Decode/Register read	ALU	Data memory	Register write	Total Delay [ps]
Load	500	200	100	500	200	1500
Store	500	200	100	500	--	1300
ALU	500	200	100	--	200	1000
Branch	500	200	100	--	--	800
Jump	500	200	--	--	--	700

- Here we can see that the longest operation is the load word operation which takes 1500ps
- But we do not have a RegRead signal so there should be no delay in reading from the register file
- So our new clock period will be 1500-200=1300ps
- And our half cycle will be 650ps

Pipelined Processor

- For pipelining we added 5 stage pipeline registers
  - Instruction Fetch
  - Instruction Decode
  - Execute stage
  - Memory stage
  - Writeback stage

Pipelined Processor

- Main changes:
  - Addition of the register stages
  - Slight modification of control unit
  - Addition of Forwarding Unit
  - Addition of hazard detection unit
  - Top module wiring modification to fit these changes

Instruction Fetch Decode Register file

IFtoID.v

```
1   module IFtoID(input clk,
2                 input reset,
3                 input stall,
4                 input [31:0]PCplus4_IF,
5                 output reg [31:0]PCplus4_ID,
6                 input [31:0]Instr_IF,
7                 output reg [31:0]Instr_ID);
8
9     always@(posedge clk)
10      begin
11
12        if (reset) begin //Resetting the whole system
13          PCplus4_ID <= 0;
14          Instr_ID <= 0;
15        end
16
17        else if(stall) begin        //If we encounter a branch or LW satll,
18          PCplus4_ID <= PCplus4_ID; //we need everything to stay the same
19          Instr_ID <= Instr_ID;
20          end
21
22        else begin
23    PCplus4_ID <= PCplus4_IF;       //if no stall or resets we need the registers to
24          Instr_ID <= Instr_IF;     //continue normally
25        end
26
27      end
28
29  endmodule
```

Instruction Decode Execute Register

IDtoEX.v

```
1   module IDtoEX(input clk,
2                 input reset,
3                 input[31:0] rsdata_ID,
4                 output reg [31:0] rsdata_Ex,
5                 input[31:0] rtdata_ID,
6                 output reg [31:0] rtdata_Ex,
7                 input[31:0] extendedimm_ID,
8                 output reg [31:0] extendedimm_Ex,
9                 input [31:0] Instr_ID,
10                output reg [31:0] Instr_Ex,
11                input RegWrite_ID,
12                output reg RegWrite_Ex,
13                input MemtoReg_ID,
14                output reg MemtoReg_Ex,
15                input MemWrite_ID,
16                output reg MemWrite_Ex,
17                input [3:0]ALUControl_ID,
18                output reg [3:0]ALUControl_Ex,
19                input ALUSrc_ID,
20                output reg ALUSrc_Ex,
21                input RegDst_ID,
22                output reg RegDst_Ex);
23
24    always@(posedge clk)
25      begin
26        if (reset) begin
27
28          rsdata_Ex <= 0;
29          rtdata_Ex <= 0;
30          extendedimm_Ex <= 0;
31          Instr_Ex <= 0;
32          RegWrite_Ex <= 0;
33          MemtoReg_Ex <= 0;
34          MemWrite_Ex <= 0;
35          ALUControl_Ex <= 0;
36          ALUSrc_Ex <= 0;
37          RegDst_Ex <= 0;
38
39        end
40
41      else   begin
42
43          rsdata_Ex <= rsdata_ID;
44          rtdata_Ex <= rtdata_ID;
45          extendedimm_Ex <= extendedimm_ID;
46          Instr_Ex <= Instr_ID;
47          RegWrite_Ex <= RegWrite_ID;
48          MemtoReg_Ex <= MemtoReg_ID;
49          MemWrite_Ex <= MemWrite_ID;
50          ALUControl_Ex <= ALUControl_ID;
51          ALUSrc_Ex <= ALUSrc_ID;
52          RegDst_Ex <= RegDst_ID;
53
54        end
55      end
56
57  endmodule
```

Execute to Memory stage

EXtoMEM.v

```
module EXtoMEM(input clk,
               input reset,
               input [31:0] ALUResult_Ex,
               output reg [31:0] ALUResult_M,
               input [31:0] fdwmux2out,
               output reg [31:0] din_M,
               input [4:0] RdOrRt_Ex,
               output reg [4:0] RdOrRt_M,
               input RegWrite_Ex,
               output reg RegWrite_M,
               input MemtoReg_Ex,
               output reg MemtoReg_M,
               input MemWrite_Ex,
               output reg MemWrite_M);

  always@(posedge clk)
    begin
      if (reset)  begin
        ALUResult_M <= 0;
        din_M <= 0;
        RdOrRt_M <= 0;
        RegWrite_M <= 0;
        MemtoReg_M <= 0;
        MemWrite_M <= 0;
        end

      else begin
        ALUResult_M <= ALUResult_Ex;
        din_M <= fdwmux2out;
        RdOrRt_M  <= RdOrRt_Ex;
        RegWrite_M  <= RegWrite_Ex;
        MemtoReg_M  <= MemtoReg_Ex;
        MemWrite_M  <= MemWrite_Ex;
      end
    end

endmodule
```

Memory to Writeback stage

MEMtoWB.v

```
module MEMtoWB(input clk,
               input reset,
               input [31:0] dout_M,
               output reg [31:0] dout_WB,
               input [31:0] ALUResult_M,
               output reg [31:0]ALUResult_WB,
               input [4:0]RdOrRt_M,
               output reg [4:0]RdOrRt_WB,
               input RegWrite_M,
               output reg RegWrite_WB,
               input MemtoReg_M,
               output reg MemtoReg_WB);

 always@(posedge clk )
   begin
     if (reset) begin
       dout_WB <= 0;
       ALUResult_WB <= 0;
       RdOrRt_WB <= 0;
       RegWrite_WB <= 0;
       MemtoReg_WB <= 0;
     end
     else begin
       dout_WB <= dout_M;
       ALUResult_WB <= ALUResult_M;
       RdOrRt_WB <= RdOrRt_M;
       RegWrite_WB  <= RegWrite_M;
       MemtoReg_WB  <= MemtoReg_M;

     end
   end
endmodule
```

3 input Multiplexer

mux3.v

```
1   module mux3 (inp0,inp1,inp2,signal,selectedinp);
2
3   parameter n=32;
4
5   input [n-1:0] inp0,inp1,inp2;
6   input [1:0] signal;
7   output reg [n-1:0] selectedinp;
8
9   always @* begin
10
11      case(signal)
12          2'b00: selectedinp<=inp0;
13          2'b01: selectedinp<=inp1;
14          2'b10: selectedinp<=inp2;
15      endcase
16
17  end
18
19  endmodule
```

Control Unit modification

```
4   module ctrlmain(input [5:0] Opcode,       //we will inoput the opcode, function and
5                   input [5:0] Func,         //and the zero for brnaching to determine control signals
6
7                   output reg  [3:0] ALUctrl,//all outputs are the control signals of
8                   output reg MemtoReg,      //the whole system
9                   output reg  Jump,
10                  output reg  MemWrite,
11                  output reg  RegDst,
12                  output reg  ALUSrc,
13                  output reg  RegWrite,
14                  output reg Branch
15              //output[6:0] tempp for ctrltestbench
16                  );
63          6'b000010: begin                            // JUMP 2
64                         temp <= 7'b0000001;          //1
65                         ALUctrl <= 4'b0000;
66                     end
67      default:   temp <= 7'bxxxxxxx;                  // NOP
68      endcase
69
70
71  {RegWrite,RegDst,ALUSrc,Branch,MemWrite,MemtoReg,Jump} = temp; //setting the control signals
72
73
74  end
75
76  //assign tempp=temp; for ctrltestbench
77
78  endmodule
```

Forwarding Unit

ForwardingUnit.v

```
module ForwardingUnit(  input [4:0] Rs_EX,
                        input [4:0] Rt_EX,
                        input [4:0] Rs_ID,
                        input [4:0] Rt_ID,
                        input [4:0] RdOrRt_M,
                        input [4:0] RdOrRt_WB,
                        input RegWrite_M,
                        input RegWrite_WB,
                        output reg[1:0] ForwardAE,
                        output reg[1:0] ForwardBE,
                        output reg ForwardAD,
                        output reg ForwardBD );

//The forwarding Unit strategy of detecting the depencies is taken from the refrence and
//has been directly implemented in here with respect to the same signals that goes to the multiplexers
//Forward A is the first MUX and forward B is the second MUX
//Forward AE is Forward A around the EXEcute
//Forward AD is forward A around the Decode
always @(*)
    begin
        // Forward around EX hazards
        if (RegWrite_M
            && (RdOrRt_M != 0)
            && (RdOrRt_M == Rs_EX))
            ForwardAE = 2'b10;
        // Forward around MEM hazards
        else if (RegWrite_WB
            && (RdOrRt_WB != 0)
            && (RdOrRt_WB == Rs_EX))
            ForwardAE = 2'b01;
        // No hazards, use the value from ID/EX
        else
            ForwardAE = 2'b00;

        // Forward around EX hazards
        if (RegWrite_M
            && (RdOrRt_M != 0)
            && (RdOrRt_M == Rt_EX))
            ForwardBE = 2'b10;
        // Forward around MEM hazards
        else if (RegWrite_WB
            && (RdOrRt_WB != 0)
            && (RdOrRt_WB == Rt_EX))
            ForwardBE = 2'b01;
        // No hazards, use the value from ID/EX
        else
            ForwardBE = 2'b00;


        //Forwarding for the branch in the decode stage in case we encounter an R-type instruction before
        //the BEQ
        ForwardAD = (RdOrRt_M !=0) && (Rs_ID == RdOrRt_M) && RegWrite_M;
        ForwardBD = (RdOrRt_M !=0) && (Rt_ID == RdOrRt_M) && RegWrite_M;


    end

endmodule
```

Hazard Detection Unit

hazarddetect.v

```
module hazarddetect(    input [4:0] Rt_EX,
                        input [4:0] Rs_D,
                        input [4:0] Rt_D,
                        input [4:0] RdOrRt_M,
                        input [4:0] RdOrRt_Ex,
                        input MemtoReg_ID,
                        input MemtoReg_M,
                        input RegWrite_EX,
                        input Branch_ID,
                        input Jump_ID,
                        output reg stall_IF_ID,
                        output reg stall_ID_EX,
                        output reg flush_EX_Mem);
reg lwstall, branchstall;
always @(*) begin
  //stall after the lw instruction
  lwstall= ((Rs_D==Rt_EX) || (Rt_D==Rt_EX)) && MemtoReg_ID;


  //stall after the branch instruction
  branchstall =Branch_ID &
            (RegWrite_EX ||
            (RdOrRt_Ex == Rs_D | RdOrRt_Ex == Rt_D) |
            MemtoReg_M ||
            (RdOrRt_M == Rs_D | RdOrRt_M == Rt_D));

   stall_ID_EX = lwstall | branchstall | Jump_ID;
   stall_IF_ID = lwstall | branchstall | Jump_ID;
   flush_EX_Mem = lwstall;

   end

endmodule
```

Top module modification

```
59  // Fetch stage
60
61  PC PCmod(clk,reset,Stall_IF, PC_IF,PCNext); //calling of our PC module to determine our first PC which will be zero
62  adder pcadd4(PC_IF, 32'd4 ,PCplus4_IF);//first step is doing PC=PC+4
63  mux #(32) branchmux(PCplus4_IF , PCbeforeBranch, Branchdet, PCBranch);//what we did was (PC+4)+(4*A), which gives us the branching address
64  mux #(32) jumpmux(PCBranch, {PCplus4_ID[31:28],Instr_ID[25:0],2'b00 }, Jump,PCNext);
65  //for jumping we take concatinationn of the first 4 bits of PC along with the 25 jump bits and multiply it by 4
66  //to determing the jump address (4*jump)
67  imem imem(PC_IF,Instr_IF);
68  //setting of instruction memory to input PC and give us corresponding instruction
69
70
71  //IF ID registers
72  IFtoID IFtoID(clk,reset|Branchdet|Jump,Stall_ID,PCplus4_IF,PCplus4_ID,Instr_IF,Instr_ID);
73  //branchdet is with reset signal in order to reset the current instruction that is
74  //after the branch incase the branch is executed
75
76  //Decode stage
77  signext immextention(Instr_ID[15:0],extendedimm_ID);
78  shiftl2 shiftl2(extendedimm_ID,extendedimmafter);//next is extending the branch-imm value to 32 for use in adder, this is the 'A' value
79  regfile regfile(clk,RegWrite_WB, reset, Instr_ID[25:21], Instr_ID[20:16], RdOrRt_WB, WriteData, rsdata_ID,rtdata_ID);
80
81  // for add and beq dependencies, the add instruction will be in the Memory stage and the branch needs in the Decode stage
82  //so we prepare a forwarding mux to do this dependency and execute the branch as it is not in the decode stage
83  mux #(32) equalmux1(rsdata_ID,ALUResult_M,ForwardAD,equalone);
84  mux #(32) equalmux2(rtdata_ID,ALUResult_M,ForwardBD,equaltwo);
85
86  //comparator for the branch instruction instead of using the ALU separately
87  //Branchdet is for determining if we should branch or not
88  assign Equal_ID = (equalone==equaltwo);
89  assign Branchdet = (Equal_ID & Branch_ID);
```

Top module modification

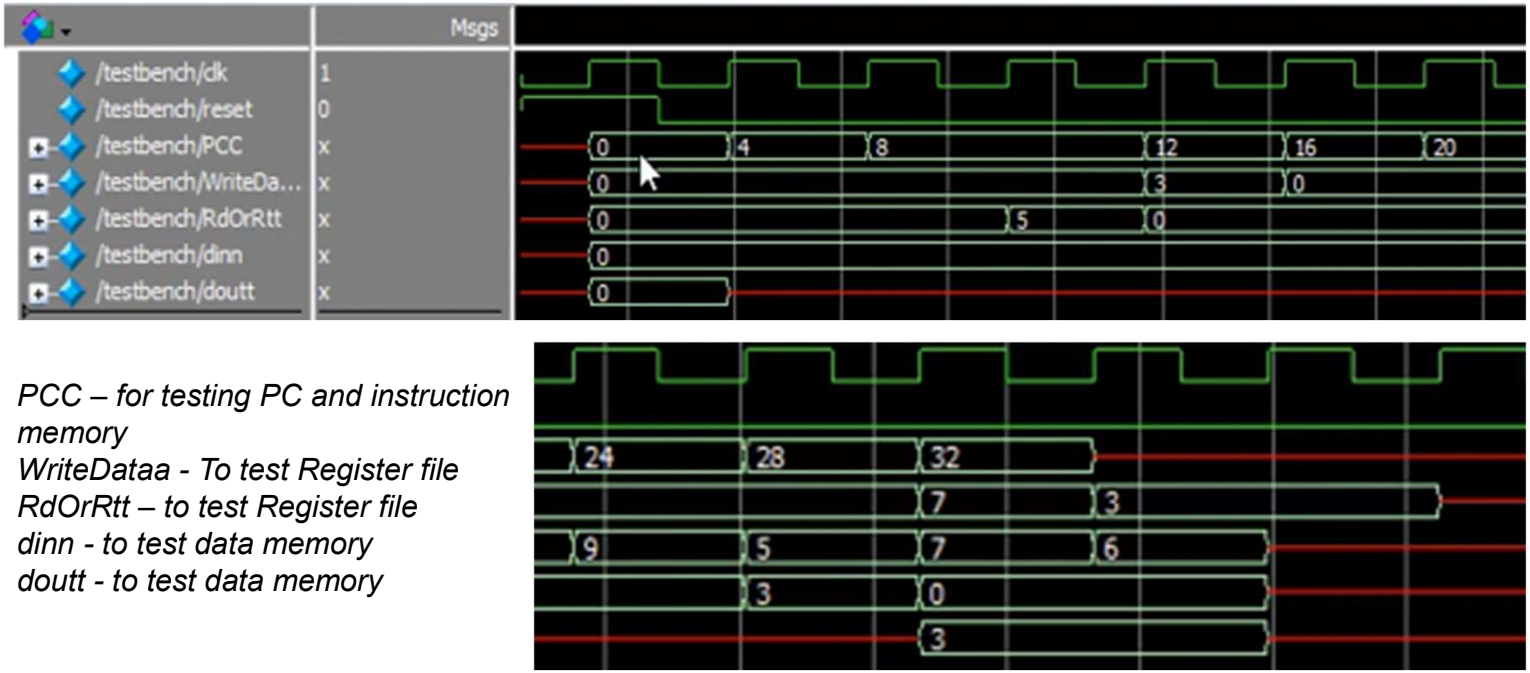
```
89  adder pcaddsigned(extendedimmafter,PCplus4_ID,PCbeforeBranch);//determining the branching address of there is one
90  ctrlmain ctrlmain(Instr_ID[31:26], Instr_ID[5:0],ALUControl_ID,MemtoReg_ID,Jump,MemWrite_ID,
91                    RegDst_ID, ALUSrc_ID, RegWrite_ID,Branch_ID);
92  //control which we input OP and Function and we get the corrsponding control signals
93
94  //flush of memtoreg signal in case of stall
95  mux #(32) flushmux(MemtoReg_ID,32'b0,Stall_IF,MemtoReg_ID_flush);
96
97  //ID Ex reg
98  IDtoEX IDtoEx(clk, reset , rsdata_ID, rsdata_Ex,rtdata_ID,rtdata_Ex, extendedimm_ID,extendedimm_Ex, Instr_ID,Instr_Ex, RegWrite_ID, RegWrite_Ex,
99                MemtoReg_ID_flush, MemtoReg_Ex, MemWrite_ID,MemWrite_Ex, ALUControl_ID, ALUControl_Ex, ALUSrc_ID, ALUSrc_Ex, RegDst_ID, RegDst_Ex);
100
101 //ex stage
102 mux3 forwardmuxA (rsdata_Ex, WriteData, ALUResult_M, ForwardAE, fwdmux1out);
103 mux3 forwardmuxB (rtdata_Ex, WriteData, ALUResult_M, ForwardBE, fwdmux2out);
104 alu alu(fwdmux1out, fwdmux2OrExtimm, ALUControl_Ex, ALUResult_Ex, ZeroFlag);
105 mux #(32) ALUSrcmux(fwdmux2out,extendedimm_Ex, ALUSrc_Ex, fwdmux2OrExtimm);
106 mux #(5) RegDstmux(Instr_Ex[20:16],Instr_Ex[15:11],RegDst_Ex, RdOrRt_Ex);
107 //setting up of the ALU inputs along with its output zero branching signal
108 //extension of immediate value to be used by register
109 //setting up the MUX to determing input value if we have R or I type instruction
110
111
112 //ex_m reg
113 EXtoMEM EXtoMEM(clk, reset, ALUResult_Ex, ALUResult_M, fwdmux2out, din_M, RdOrRt_Ex, RdOrRt_M,
114                 RegWrite_Ex, RegWrite_M, MemtoReg_Ex, MemtoReg_M, MemWrite_Ex, MemWrite_M );
115
116 //memory stage
117 dmem dmem(clk,MemWrite_M,ALUResult_M, din_M, dout_M);
118 //pretty simple assigning of out datamemory element with rtdata as out input
119 //for store word immediate instructions and dout as output for load word
120 //immediate instructions
121
122
123 //forwarding and hazard
124 ForwardingUnit ForwardingUnit( Instr_Ex [25:21], Instr_Ex [20:16], Instr_ID [25:21], Instr_ID [20:16], RdOrRt_M, RdOrRt_WB, RegWrite_M, RegWrite_WB, ForwardAE, ForwardBE, ForwardAD, ForwardBD);
125 hazarddetect hazarddetect(Instr_Ex [20:16], Instr_ID [25:21], Instr_ID [20:16], RdOrRt_M,RdOrRt_Ex,MemtoReg_ID_flush,MemtoReg_M,RegWrite_Ex,Branch_ID,Jump,
126 Stall_IF,Stall_ID,Flush_Ex );
127
128 //mem_WB
129 MEMtoWB MEMtoWB(clk,reset, dout_M, dout_WB, ALUResult_M, ALUResult_WB, RdOrRt_M, RdOrRt_WB,
130                 RegWrite_M, RegWrite_WB, MemtoReg_M, MemtoReg_WB);
131
132 //Wb stage
133 mux #(32) MemtoRegmux(ALUResult_WB, dout_WB, MemtoReg_WB,WriteData);
```

Pipelined processor test file and encoding

memfile.txt

```
20050003
10000001
00000020
00A0482A
aca50004
8ca70004
00a03020

0 addi a1,zero,3
4 beq zero,zero,1
8 add zero,zero,zero
12 slt t1,a1,zero
16 sw a1,4(a1)
20 lw a3,4(a1)
24 sub a2,a1,zero
```

Pipelined Testbench - PC, Instruction memory, Register file, Data memory results



PCC – for testing PC and instruction memory
WriteDataa - To test Register file
RdOrRtt – to test Register file
dinn - to test data memory
doutt - to test data memory