

University of Jordan  
Computer Engineering Department  
CPE439: Computer Design Lab

Experiment 1: Introduction to Verilogger Pro

**Objective:**

The objective of this experiment is to introduce the student to the environment of the Verilog simulator, and write simple programs.

**The VeriLogger Pro Environment:**

When you start the VeriLogger Pro program, you will notice that there are four windows. The upper left is the **project window**; in this window you select the HDL source files to be simulated. The upper right window enables the programmer to add a free **parameter**. The lower left window is the place where you will see the **timing diagram** that shows the waveforms of the signals monitored throughout the simulation. The lower right window is the place where the contents of the **log file** can be seen, and the **errors of compilation** are displayed.

**How to write a program that describes the operation of AND and NAND gates?**  
**Perform the following steps:**

1. **Open a new project** file by selecting “*New HDL Project*” from the *Project menu*. Name the project “**AND\_project.hpj**”. The name is given when you select “*Save HDL Project As...*” from the *Project menu*.
2. **Open a new source file** by selecting “*New HDL File*” from the *Editor menu*. A new window should appear in which you should copy the following Verilog code.

```
// This module describes 2-input NAND gate behaviorally
module NAND (out, in1, in2);
  input in1, in2;
  output out;

  assign #2 out = ~ (in1 & in2);

endmodule
```

3. **Save this new HDL file** as “**NAND.v**” by selecting “*Save HDL File As...*” from the *Editor menu*.
4. **Add NAND.v to your HDL project** by selecting the project window, right click in the workspace of this window, and select “*Add HDL File(s)...*”.
5. Similar to Steps 2 through 4, **add to your project a new file named AND.v** that contains following code.

```
// This module describes 2-input AND gate structurally
module AND (out, in1, in2);
  input in1, in2;
  output out;
  wire w1;

  NAND N1 (w1, in1, in2);
  NAND N2 (out, w1, w1);

endmodule
```

6. Now you need to test your AND and NAND modules and verify that they operate properly. Similar to Steps 2 through 4, **add to your project a new file named test.v** that contains following code.

```
module test;
  reg in1,in2;           //declaring in1 and in2 as registers for inputs
  wire andout;          //declaring andout as wire for output

  AND n1(andout,in1,in2); //Creating an instance of the module AND

  initial begin: stop_at //This initial statement selects
    #100; $finish;      //an appropriate simulation period
  end                   //We choose it here to be 250 time units

  initial begin :init
    in1=0;
    in2=0; //Initially set in1 and in2 to zero

    /* The $display statement prints the sentence between quotations in the
    log file. It Operates in the same way the printf function does in the C
    language.*/
    $display("*** Table of changes ***");
    $display("Time    in1    in2    andout");

    /* The monitor statement prints the values of the different parameters
    whenever a change in the value of one of them or more occurs.*/
    $monitor("%0d    %b    %b    %b", $time, in1, in2, andout);
  end

    /* We use this always construct to continuously vary the values of
    the input registers in1 and in2, in order to have a simulation whose
    output continuously changes.*
  always #10 in1 = ~in1;
  always #20 in2 = ~in2;

endmodule
```

7. After you have added the required files **start the program simulation** by clicking on the **green arrow**  in the center of the Tool bar. The results should appear in the log file and the waveforms should appear in the timing diagram.

**University of Jordan**  
**Computer Engineering Department**  
**CPE439: Computer Design Lab**

**Experiment 2: 32-Bit ALU**

**Description**

In this experiment, students have to design and test a 32-bit ALU with the block diagram shown in Figure 1 and the operations listed in Table I. The design should be done using Verilog structural programming by utilizing the modules available in the library *Library439.v* that is available online. It is advised that you follow the modular approach in your design, in which you start by designing small modules from which you build the larger modules.

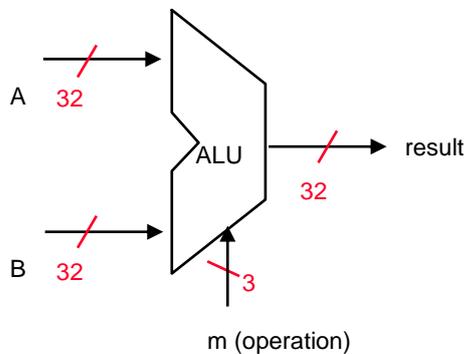


Figure 1. 32-bit ALU block diagram

m (operation)	Function
000	Or
001	And
010	Xor
011	Add
100	Nor
101	Nand
110	Slt (Set on less than)
111	Subtract

Table I. Arithmetic and logic operations supported by the ALU

**Procedure**

- Using modular design, you may start the design of the 32-bit ALU by considering the implementation of a 1-bit ALU shown in Figure 2. In order to build this circuit, most of the primitive and basic gates are available in the library *Library439.v*. However, you have to design the 1-bit full adder and the 8-to-1 multiplexer according the following specifications. Keep in your mind that your Verilog modules for these units should be structural.

a) **(Prelab.)** 1-bit FA

The block diagram and truth table for the full adder are shown in Figure 3. You should write a Verilog structural module to implement this logic circuit using the following template.

```

module FullAdder(Cout, sum, a, b, Cin);
    output sum, Cout;
    input Cin, a, b;

    // implementation details are left to the student
endmodule

```

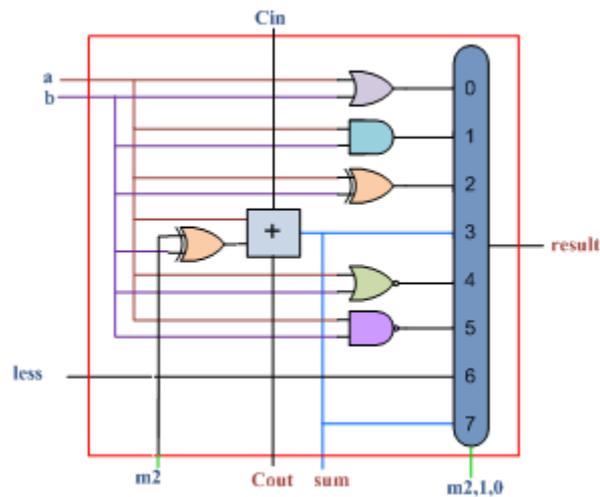


Figure 2. 1-bit ALU.

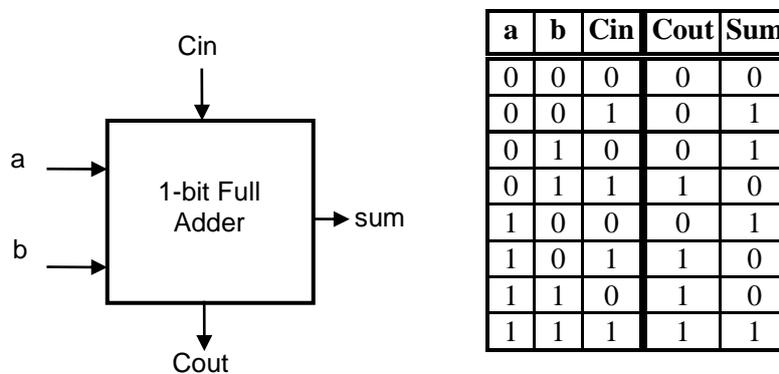


Figure 3. 1-bit FA block diagram and truth table.

b) **(Prelab.)** 8-to-1 Multiplexor

You should write a Verilog module that implements this multiplexor using structural modeling. Your module should use the following template.

```

module Mux_8_to_1(result, s, in);
  output result;
  input [2:0] s;
  input [7:0] in;

  // implementation details are left to the student...
endmodule

```

2) Once you have built the full adder and the multiplexor, you can now move to the next level by writing the Verilog module that implements the 1-bit ALU using the following template.

```

module ALU_1(result, sum, Cout, Cin, a, b, less, m);
  output result, sum, Cout;
  input Cin, a, b, less;
  input [2:0] m;

  // implementation details are left to the student...
endmodule

```

- 3) After you have designed the 1-bit ALU, you may choose to use 32 copies of this module to build the large 32-bit ALU. However, such approach is time consuming and requires a lot of effort in wiring-up these instances. Instead, consider building the 32-bit ALU using 8-bit ALUs. In this case you need to wire only 4 instances. So, consider writing a Verilog module for an 8-bit ALU using the 1-bit ALU designed in the previous step. Use the following template.

```

module ALU_8(result, sum, Cout, Cin, a, b, less, m);
    output [7:0]result, sum;
    output Cout;
    input Cin;
    input [7:0]a, b, less;
    input [2:0] m;

    // implementation details are left to the student...
endmodule

```

- 4) Once you have built the 8-bit ALU, it is time to construct the 32-bit ALU. Use the following template for this purpose.

```

module ALU_32(result, a, b, m);
    output [31:0]result;
    input [31:0]a, b;
    input [2:0] m;

    // implementation details are left to the student...
endmodule

```

## Testing

Write a Verilog module to test your 32-bit ALU. The module should use the data given in Table II as a benchmark. Generate the timing diagram and estimate the maximum delay in your design.

a	b	m
00000102 <sub>h</sub>	00000c0f <sub>h</sub>	000
00000102 <sub>h</sub>	00000c0f <sub>h</sub>	001
00000102 <sub>h</sub>	00000c0f <sub>h</sub>	010
00000102 <sub>h</sub>	00000c0f <sub>h</sub>	100
00000102 <sub>h</sub>	00000c0f <sub>h</sub>	101
00000102 <sub>h</sub>	00000c0f <sub>h</sub>	110
000f0001 <sub>h</sub>	00000024 <sub>h</sub>	110
000f0001 <sub>h</sub>	00000024 <sub>h</sub>	011
000f0001 <sub>h</sub>	00000024 <sub>h</sub>	111

**University of Jordan**  
**Computer Engineering Department**  
**CPE439: Computer Design Lab**  
**Experiment 3: Register File**

- **Description**

In this experiment, students have to design and test a register file with 32 32-bit registers to be used in the design of the MIPS like processor by the end of the semester. The register file to be designed is shown in Figure 1. It consists of 32 32-bit negative edge-triggered registers, one write port, and two read ports. The write port requires a decoding circuit in order to determine which register is enabled to receive the data available on the WriteData input based on the 5-bit address supplied on WriteReg port. This is done through the 5-to-32 decoder.

For the read ports, they are essentially built using 32-bit wide 32-to-1 multiplexers. The 5-bit read address ports, ReadReg1 and ReadReg2, are connected to the selection lines of the multiplexers to select the contents of the addressed registers.

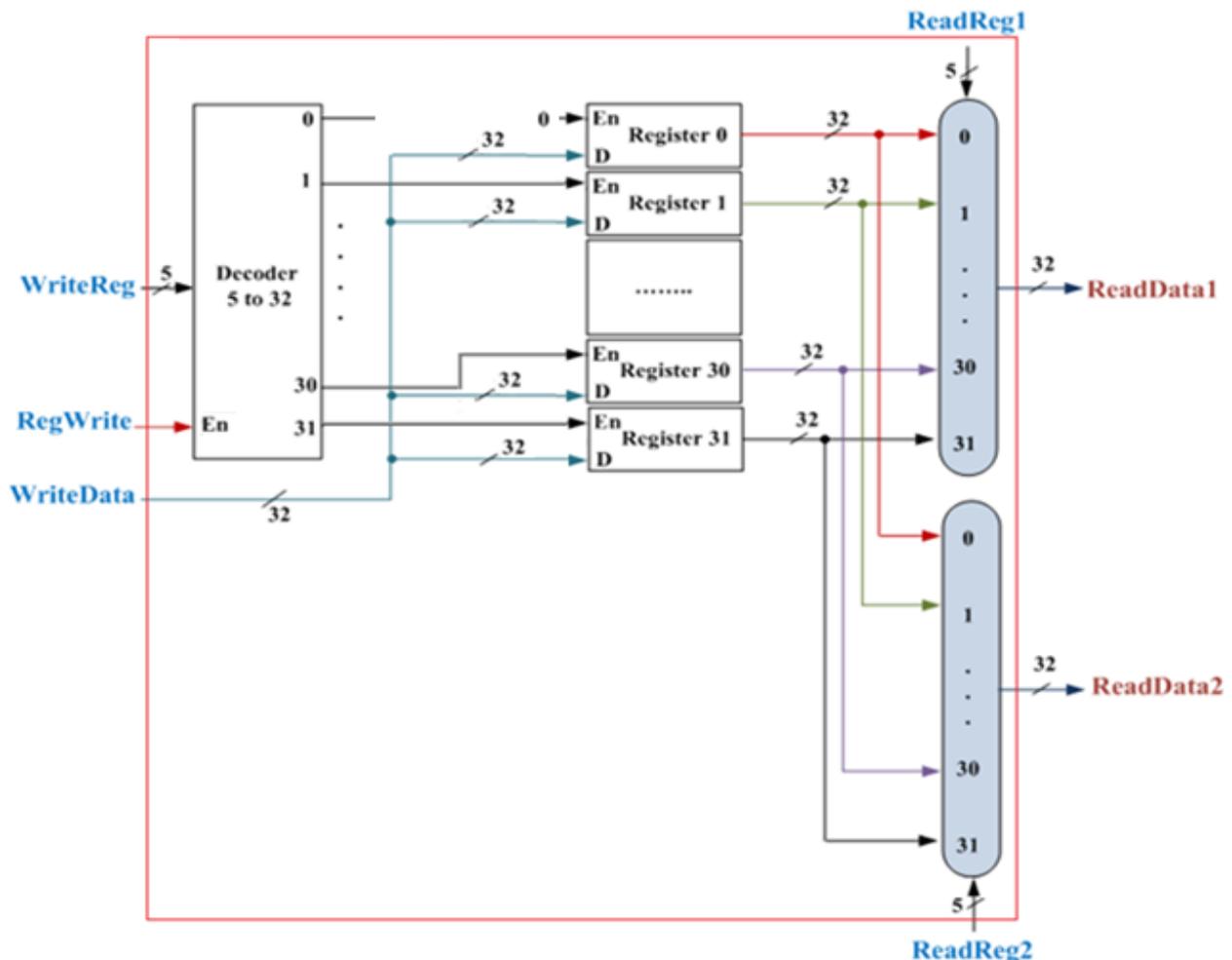


Figure 1. Layout of the register file.

- **Procedure**

The required register file is to be built using Verilog structural programming, unless otherwise stated, by utilizing the modules available in the library *Library439.v* that is available online. This has to be done in a modular fashion. We suggest that you follow the following steps in your design.

1) ***(Prelab.) 32-Bit Register***

Instead of combining 32 negative edge-triggered flip-flops to build this unit, you may consider using 4 instances of the 8-bit register module *REG8negclk* that is available in the library. Your module should use the following template.

```
module REG32negclk (Q, D, clk, reset, enable);
    input  clk, reset, enable;
    input  [31:0] D;
    output [31:0] Q;
    // implementation details are left to the student...
endmodule
```

2) ***(Prelab.) 32-Bit Multiplexor***

Due to the complexity of designing and wiring-up a multiplexor of this size, we suggest building it using Verilog behavioral modeling. Your module should use the following template.

```
module Mux_32_to_1_32bit(out, s, in);
    input  [1023:0] in;
    input  [4:0]s;
    output [31:0]out;
    reg    [31:0]out;

    always @(in or s)
        #6 case (s)
            5'd0 : out = in[31:0];
            5'd1 : out = in[63:32];
            // The student should complete all cases
            5'd30 : out = in[991:960];
            5'd31 : out = in[1023:992];
        endcase
endmodule
```

3) ***5-to-32 Decoder***

Building a decoder with this size could be cumbersome. Instead, consider building small decoders and then cascading them to obtain the 5-to-32 decoder as follows:

a) **2-to-4 Decoder**

You should write a Verilog module that implements this decoder using structural modeling. Your module should use the following template.

```
module Decoder2to4 (out, in, enable);
    input  enable; //active high enable
    input  [1:0]in;
    output [3:0]out;
    // implementation details are left to the student.....
endmodule
```

**b) 3-to-8 Decoder with enable**

You should write a Verilog module that implements this decoder using structural modeling. Your module should use the following template.

```
module Decoder3to8 (out, in, enable);
  input  enable; //active high enable
  input  [2:0]in;
  output [7:0]out;
  // implementation details are left to the student.....
endmodule
```

**c) 5-to-32 Decoder**

You should write a Verilog module that implements this decoder using one instance of Decoder2to4 module and four instances of Decoder3to8 module only. Your module should use the following template.

```
module Decoder5to32 (out, in, enable);
  input  enable; //active high enable
  input  [4:0]in;
  output [31:0]out;
  // implementation details are left to the student..
endmodule
```

**4) The Register File**

Once the previous modules have been implemented, it is time now to combine them into one block that implements the register file. Use the following template for this purpose.

```
module RegFile(readdata1 ,readdata2, readreg1, readreg2,
               writedata, writereg, regwrite, clk, reset);
  input  regwrite, clk, reset;
  input  [4:0]readreg1, readreg2, writereg;
  input  [31:0]writedata;
  output [31:0]readdata1, readdata2;
  // implementation details are left to the student.....
endmodule
```

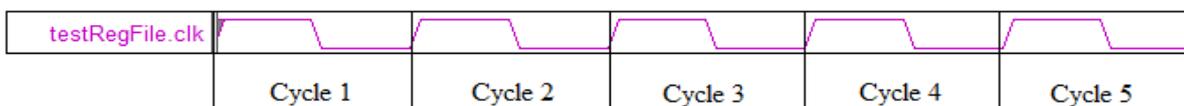
• **Testing**

Write a Verilog module to test your register file module. The test module should use the data given in Table I as a benchmark. Generate the timing diagram and estimate the maximum delay in your design.

**Table I. Data to be used in design testing and verification.**

Cycle #	Clock	writedata	writereg	regwrite	readreg1	readreg2	reset
1	1 to 0 to 1	00000ff <sub>h</sub>	00011 <sub>b</sub>	0	00000 <sub>b</sub>	00011 <sub>b</sub>	1
2	1 to 0 to 1	00000150 <sub>h</sub>	00011 <sub>b</sub>	1	00011 <sub>b</sub>	00100 <sub>b</sub>	0
3	1 to 0 to 1	00000066 <sub>h</sub>	00100 <sub>b</sub>	1	00011 <sub>b</sub>	00100 <sub>b</sub>	0
4	1 to 0 to 1	00000008 <sub>h</sub>	00011 <sub>b</sub>	0	00011 <sub>b</sub>	01000 <sub>b</sub>	0
5	1 to 0 to 1	00000040 <sub>h</sub>	01000 <sub>b</sub>	0	00001 <sub>b</sub>	00101 <sub>b</sub>	0

The waveform for the clock signal should similar to the following one:



**University of Jordan**  
**Computer Engineering Department**  
**CPE439: Computer Design Lab**  
**Experiment 4: Instruction and Data Memories**

• **Description**

In this experiment, students have to design and test the instruction memory in addition to the data memory in order to use them in the design of the MIPS like processor by the end of the semester. The block diagrams and specifications for these units are shown Figure 1.

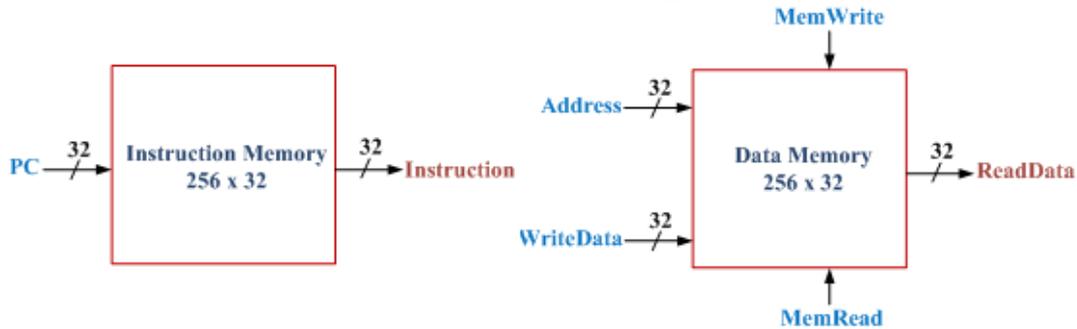


Figure 1. The Block Diagram for Instruction and Data Memories.

• **Procedure**

The required memories are to be built using Verilog behavioral programming.

1) ***Instruction Memory***

We just read from the instruction memory and we don't write it, and we read an instruction every cycle so we don't need an explicit read signal. Write a Verilog module to implement this memory and initialize it as given in the following module. You don't have to add further statements. *Pay attention that the memory is 32 bit wide, i.e. it is word-addressed, while the PC which contains the byte address. So, the contents of the program counter should be divided by 4.*

```
module Instruction_Memory(PC, instruction);
  input  [31:0] PC;
  output [31:0] instruction;
  reg     [31:0] instruction;
  reg     [31:0] IM [255:0];

  initial begin
    IM[0] = 32'h00000010;
    IM[1] = 32'h00000020;
    IM[2] = 32'h00000030;
    IM[3] = 32'h00000040;
    IM[4] = 32'h00000050;
  end

  //MIPS instruction is 4 Byte, Processor counts bytes not words
  always @ (PC )
    #15 instruction = IM[PC>>2]; //To get the correct
                                //address, we divide by 4

endmodule
```

## 2) Data Memory

We write and read from the data memory, and we neither read nor write every cycle so we need explicit read and write signals. *Note that this data memory is also 32-bit wide, thus it is word-addressed. However, the memory address formed in LW and SW instructions is the byte address.* The data memory should be initialized such that each location has a number greater than the previous location by 1. For example, word 0 is initialized to 0x00000000, word 1 is 0x00000001, word 2 is 0x00000002 and so on. Use **for loop to do this initialization**. Based on this description, use the following template to implement this memory.

```
module Data_Memory(readdata, address, writedata, memwrite,
                  memread, clk);

    input [31:0] address , writedata ;
    input memwrite , memread , clk;
    output [31:0] readdata;

    // implementation details are left to the student.....
endmodule
```

### • Testing

Write the Verilog modules to test your instruction and data memory modules. The test module for the instruction memory should use the data given in Table I as a benchmark, and the test module for the data memory should use the data given in Table II as a benchmark.

**Table I. Test data for Instruction Memory**

PC
00000000 <sub>h</sub>
00000004 <sub>h</sub>
00000008 <sub>h</sub>
0000000C <sub>h</sub>
00000010 <sub>h</sub>
00000014 <sub>h</sub>

**Table II. Test data for Data Memory**

Cycle #	Clock	writedata	address	memread	memwrite
1	1 to 0 to 1	00000000 <sub>h</sub>	00000014 <sub>h</sub>	0	0
2	1 to 0 to 1	000000e5 <sub>h</sub>	00000014 <sub>h</sub>	1	0
3	1 to 0 to 1	00000f14 <sub>h</sub>	00000014 <sub>h</sub>	0	1
4	1 to 0 to 1	0000000a <sub>h</sub>	00000018 <sub>h</sub>	0	1
5	1 to 0 to 1	0000009e <sub>h</sub>	00000014 <sub>h</sub>	1	0
6	1 to 0 to 1	0000007f <sub>h</sub>	00000018 <sub>h</sub>	1	0

**University of Jordan**  
**Computer Engineering Department**  
**CPE439: Computer Design Lab**  
**Experiment 5: The Control Unit**

• **Description**

In this experiment, students have to design and test the control unit to use it in the design of the MIPS like processor. The control unit is responsible for generating all the signals required to control different elements of the processor datapath that will be designed in the next experiment. The values of control signals are determined based on the opcode and function fields of the MIPS instructions. The block diagram and specifications for this unit is shown in Figure 1.

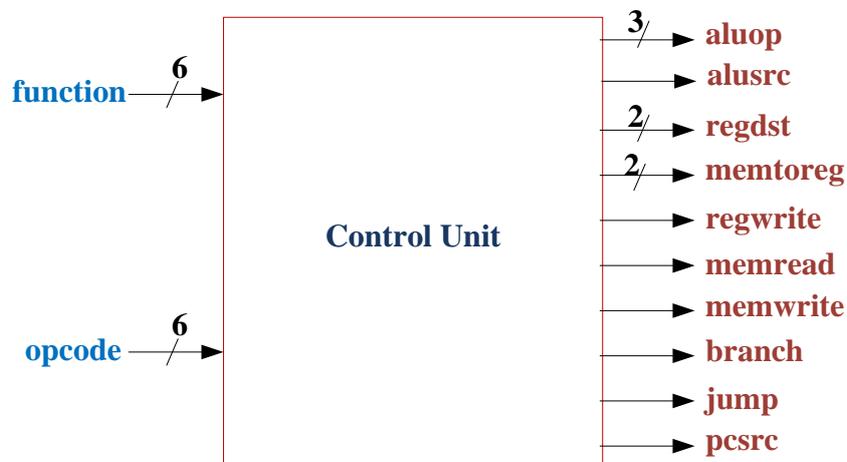


Figure 1. The Block Diagram for the Control Unit.

• **Procedure**

In order to build this control unit you need to find the equations for the output signals which are shown on Table 1, then build these equations using structural modeling. Don't attempt to use logic minimization as the hardware has 12 inputs. Instead, try finding a logic expression for each signal by inspecting the truth table.

For example, the  $aluop[0]$  signal equals to 1 in one of the following cases:

- 1) The left most two bits of the opcode are zero and the right most bit of the function field is 1. (R-type instructions)
- 2) Bit 4 of the opcode is 1 and the right most bit is 1. (Immediate instructions)
- 3) The left most two bits of the opcode are 01. (Memory instructions)

This makes the logic expression for this signal as follows

$$aluop[0] = \overline{op5} \cdot \overline{op4} \cdot func0 + \overline{op5} \cdot op4 \cdot op0 + op5 \cdot \overline{op4}$$

You can follow the same approach in deriving the expressions for the remaining signals. Note that the assignment of the opcode and function fields for the instructions in Table 1 is tailored to simplify your design and they are not the same as those available in the MIPS instructions datasheet.

*Your module should use the following template.*

```

module ControlUnit(aluop, alusrc, regdst, memtoreg, regwrite,
                  memread, memwrite, branch, jump, pcsrc,
                  opcode, func);

    input [5:0] opcode, func;
    output [2:0]aluop;
    output [1:0]regdst, memtoreg;
    output alusrc, regwrite, memread, memwrite, branch, jump,
           pcsrc;
    // implementation details are left to the student.....

endmodule

```

instruction	opcode	function	aluop[2]	aluop[1]	aluop[0]	alusrc	regdst[1]	regdst[0]	memtoreg[1]	memtoreg[0]	regwrite	memread	memwrite	branch	jump	pcsrc
OR	000000	000000	0	0	0	0	0	1	0	0	1	0	0	0	0	0
AND	000000	000001	0	0	1	0	0	1	0	0	1	0	0	0	0	0
XOR	000000	000010	0	1	0	0	0	1	0	0	1	0	0	0	0	0
ADD	000000	000011	0	1	1	0	0	1	0	0	1	0	0	0	0	0
NOR	000000	000100	1	0	0	0	0	1	0	0	1	0	0	0	0	0
NAND	000000	000101	1	0	1	0	0	1	0	0	1	0	0	0	0	0
SLT	000000	000110	1	1	0	0	0	1	0	0	1	0	0	0	0	0
SUB	000000	000111	1	1	1	0	0	1	0	0	1	0	0	0	0	0
JR	000000	001000	x	x	x	x	x	x	x	x	0	0	0	0	0	1
ORI	010000	-	0	0	0	1	0	0	0	0	1	0	0	0	0	0
ANDI	010001	-	0	0	1	1	0	0	0	0	1	0	0	0	0	0
XORI	010010	-	0	1	0	1	0	0	0	0	1	0	0	0	0	0
ADDI	010011	-	0	1	1	1	0	0	0	0	1	0	0	0	0	0
NORI	010100	-	1	0	0	1	0	0	0	0	1	0	0	0	0	0
NANDI	010101	-	1	0	1	1	0	0	0	0	1	0	0	0	0	0
SLTI	010110	-	1	1	0	1	0	0	0	0	1	0	0	0	0	0
SUBI	010111	-	1	1	1	1	0	0	0	0	1	0	0	0	0	0
LW	100011	-	0	1	1	1	0	0	0	1	1	1	0	0	0	0
SW	101011	-	0	1	1	1	x	x	x	x	0	0	1	0	0	0
BEQ	110000	-	x	x	x	x	x	x	x	x	0	0	0	1	0	0
J	110001	-	x	x	x	x	x	x	x	x	0	0	0	0	1	1
JAL	110011	-	x	x	x	x	1	0	1	0	1	0	0	0	1	1

**Table 1. Truth Table for the Control Unit**

- **Testing**

Write the Verilog modules to test your control unit module. The test module should use the data given in Table 2 as a benchmark. Generate the timing diagram for the control signals.  
Estimate the maximum delay in your design.

opcode	func
000000 <sub>b</sub>	000000 <sub>b</sub>
000000 <sub>b</sub>	000001 <sub>b</sub>
000000 <sub>b</sub>	000010 <sub>b</sub>
000000 <sub>b</sub>	000011 <sub>b</sub>
000000 <sub>b</sub>	000100 <sub>b</sub>
000000 <sub>b</sub>	000101 <sub>b</sub>
000000 <sub>b</sub>	000110 <sub>b</sub>
000000 <sub>b</sub>	000111 <sub>b</sub>
000000 <sub>b</sub>	001000 <sub>b</sub>
010000 <sub>b</sub>	001000 <sub>b</sub>
010001 <sub>b</sub>	001000 <sub>b</sub>
010010 <sub>b</sub>	001000 <sub>b</sub>
010011 <sub>b</sub>	001000 <sub>b</sub>
010100 <sub>b</sub>	001000 <sub>b</sub>
010101 <sub>b</sub>	001000 <sub>b</sub>
010110 <sub>b</sub>	001000 <sub>b</sub>
010111 <sub>b</sub>	001000 <sub>b</sub>
100011 <sub>b</sub>	001000 <sub>b</sub>
101011 <sub>b</sub>	001000 <sub>b</sub>
110000 <sub>b</sub>	001000 <sub>b</sub>
110001 <sub>b</sub>	001000 <sub>b</sub>
110011 <sub>b</sub>	001000 <sub>b</sub>

**Table 2. Test data for the Control Unit**

**University of Jordan**  
**Computer Engineering Department**  
**CPE439: Computer Design Lab**  
**Experiment 6: Single Cycle Implementation**

### Description

In this experiment, students have to construct a Verilog module for a single cycle implementation of the MIPS like processor that they have been working on since the beginning of the semester. This module should include the five modules that they have constructed in the previous experiments, namely: **ALU**, **RegFile**, **Instruction\_Memory**, **Data\_Memory**, and **ControlUnit** modules. Additionally, few small modules that required to support specific instructions are to be designed and implemented.

### Procedure

The single cycle implementation to be designed is shown in Figure 1. In order to build this implementation, you need to design the following components and then connect them with the modules constructed in previous experiments. To simplify the design, these new modules are to be implemented using *behavioral modeling*.

- **Secondary modules**

- 1) **32-bit Adder**

Your module should use the following template. (The delay = 50 ns)

```
module Adder32bit (out, a, b);  
    input [31:0]a, b;  
    output [31:0]out;  
  
    // implementation details are left to the student..  
endmodule
```

- 2) **Sign Extend Unit**

Your module should use the following template.

```
module SignExtend (out, in);  
    input [15:0]in;  
    output [31:0]out;  
  
    // implementation details are left to the student..  
endmodule
```

- 3) **Comparator**

Your module should use the following template. (The delay = 10 ns)

```
module Comparator32bit (equal, a, b);  
    input [31:0]a, b;  
    output equal;  
  
    // implementation details are left to the student..  
endmodule
```

4) **26-Bit Shift Left by 2 Unit**

Your module should use the following template.

```
module ShiftLeft26_by2(out, in);
    input  [25:0]in;
    output [27:0]out;

    // implementation details are left to the student...
endmodule
```

5) **32-Bit Shift Left by 2 Unit**

Your module should use the following template.

```
module ShiftLeft32_by2(out, in);
    input  [31:0]in;
    output [31:0]out;

    // implementation details are left to the student...
endmodule
```

6) **(Prelab.) 5 Bit 3-to-1 Multiplexor**

Your module should use the following template. (The delay = 6 ns)

```
module Mux_3_to_1_5bit(out, s, i2, i1, i0);
    input  [4:0] i2, i1, i0;
    input  [1:0] s;
    output [4:0] out;

    // implementation details are left to the student...
endmodule
```

7) **(Prelab.) 32 Bit 3-to-1 Multiplexor**

Your module should use the following template. (The delay = 6 ns)

```
module Mux_3_to_1_32bit(out, s, i2, i1, i0);
    input  [31:0] i2, i1, i0;
    input  [1:0] s;
    output [31:0] out;

    // implementation details are left to the student...
endmodule
```

8) **(Prelab.) 32 Bit 2-to-1 Multiplexor**

Your module should use the following template. (The delay = 6 ns)

```
module Mux_2_to_1_32bit(out, s, i1, i0);
    input  [31:0] i1, i0;
    input  s;
    output [31:0] out;

    // implementation details are left to the student...
endmodule
```

9) **The Program Counter**

The program counter is a 32 bit register so we can use *REG32negclk* module which we have built in register file experiment.

- **The Processor Module**

Once you have implemented the previous modules, you can proceed and connect them to the modules you have built in earlier experiments. Your module should use the following template.

```

module Processor(clk, reset, enable);
input clk, reset, enable;
//REG32negclk ProgramCounter(Q, D, clk, reset, enable);
//Instruction_Memory(PC, instruction);
//Adder32bit (out, a, b); for PC + 4
//Mux_3_to_1_5bit(out, s, i2, i1, i0);
//ControlUnit(aluop, ....., jump, pcsrc, opcode, func);
//ShiftLeft26_by2(out, in);
//SignExtend (out, in);
//RegFile(readdata1, readdata2, ....., clk, reset);
//Mux_2_to_1_32bit(out, s, i1, i0); for the input b of the ALU
//ALU_32(result, a, b, m);
//ShiftLeft32_by2(out, in);
//Adder32bit (out, a, b); to calculate branch target Address
//Comparator32bit (equal, a, b);
//AND (out, in1, in2);
//Mux_2_to_1_32bit(out, s, i1, i0); branch address or PC + 4
//Mux_2_to_1_32bit(out, s, i1, i0); jump address or jr
//Mux_2_to_1_32bit(out, s, i1, i0); select the final address
//Data_Memory(readdata, address, ....., clk);
//Mux_3_to_1_32bit(out, s, i2, i1, i0);
endmodule

```

### Testing

- **(Prelab.)** It is required to test your design for the entire processor by filling the **instruction memory module** by the instructions sequence shown in the following table. You need to determine the machine code for these instructions based on Table 1 of the previous experiment.

**Table 1. The Content of the Instruction Memory**

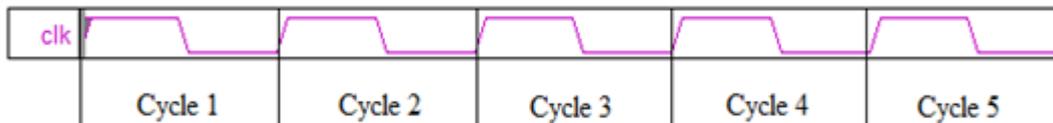
Address	Instruction	Machine Code
00	LW R1, 4(R0)	8C010004 <sub>h</sub>
01	LW R12, 12(R0)	
02	LW R3, 20(R0)	
03	LW R4, 28(R0)	
04	NAND R5, R1, R3	
05	NORI R6, R5, 1023	
06	SUB R8, R4, R12	
07	JAL 11	
08	XOR R7, R5, R6	
09	SW R7, 8(R0)	
10	J 14	
11	ANDI R9, R8, 2047	
12	BEQ R3, R6, 1	
13	JR R31	
14	OR R10, R7, R9	
15	SLT R11, R8, R1	

**Table 2. The Test Data for the Processor**

- **(Prelab.)** Next, write a Verilog **test module** to test your processor module, your test module should run for 17 cycles.

Cycle #	clk	enable	reset
1	1 to 0 to 1	1	1
2	1 to 0 to 1	1	0
3	1 to 0 to 1	1	0
4	1 to 0 to 1	1	0
5	1 to 0 to 1	1	0
6	1 to 0 to 1	1	0
7	1 to 0 to 1	1	0
8	1 to 0 to 1	1	0
9	1 to 0 to 1	1	0
10	1 to 0 to 1	1	0
11	1 to 0 to 1	1	0
12	1 to 0 to 1	1	0
13	1 to 0 to 1	1	0
14	1 to 0 to 1	1	0
15	1 to 0 to 1	1	0
16	1 to 0 to 1	1	0
17	1 to 0 to 1	1	0

The waveform for the clock signal should similar to the following one:



Your timing diagram should contain the following signals:

- Clock, reset, and enable.*
- PC (The output of the program counter).*
- Instruction (The output of the instruction memory).*
- The writedata, readreg1, readreg2, and writereg for the register file.*
- The output for the registers R1, R3, R5, R6, R8, R9.*
- The input and the output of the ALU (a, b, m, result).*
- The output of the data memory.*

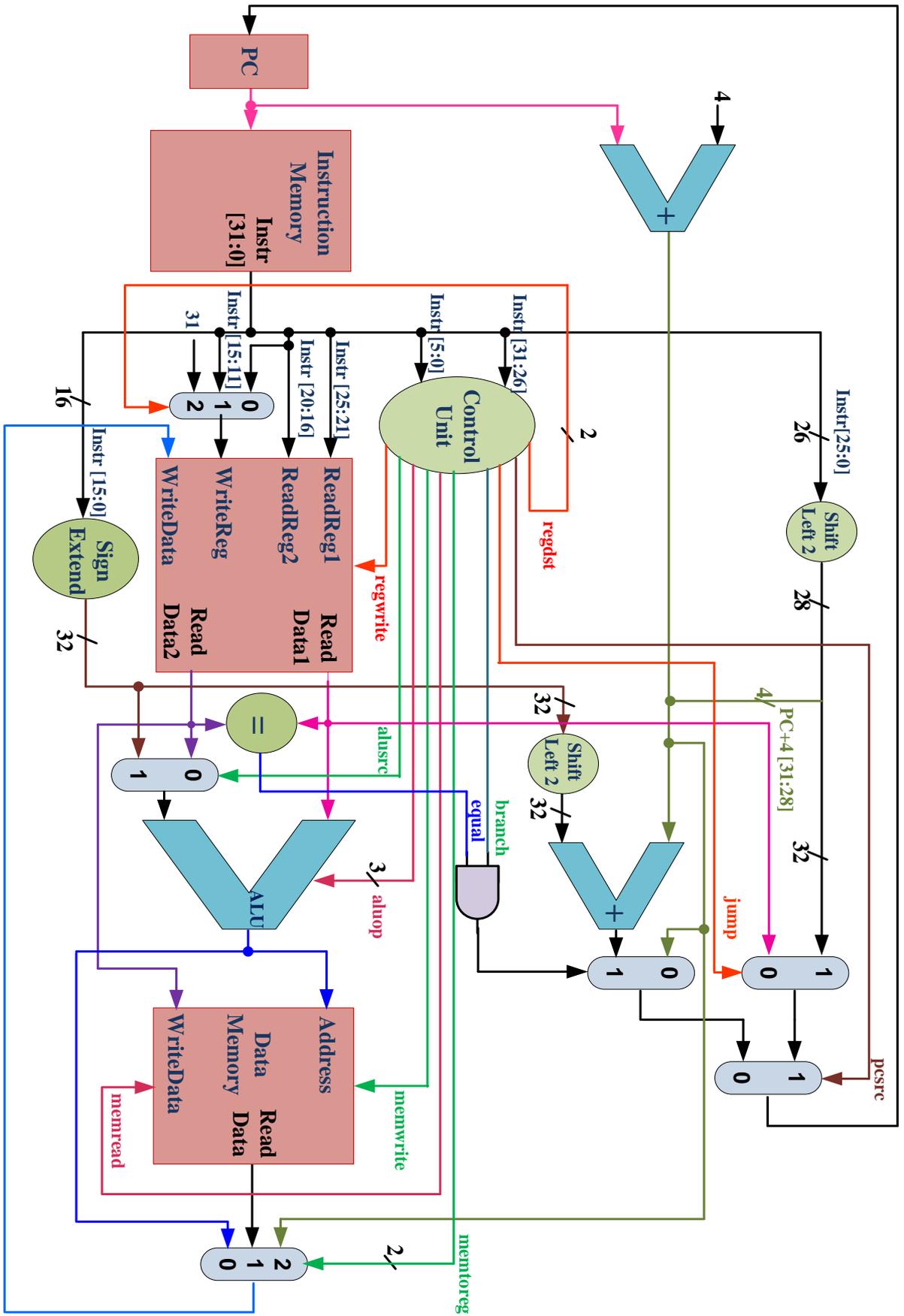


Figure 1. The Datapath for MIPS Like Processor.

**University of Jordan**  
**Computer Engineering Department**  
**CPE439: Computer Design Lab**  
**Experiment 7: Pipelined Implementation**

**Description**

In this experiment, students have to construct a Verilog module for a pipelined implementation of the MIPS like processor. This module should include all modules that they have been used in the implementation of the single cycle processor in addition to few small modules that required to the pipelined processor.

**Procedure**

The pipelined implementation to be designed is shown in Figure 1. In order to build this implementation, you need to design the following components structurally and then add them to the processor module which we built in the previous experiment.

- **Secondary modules**

- 1) ***(Prelab.)* The Program Counter**

We need to modify the program counter to make it a 32 bit register with *positive edge trigger* to enable us to make the pipelining, so you may consider using 4 instances of the 8-bit register module *REG8* that is available in the library *Library439.v*. Your module should use the following template.

```
module ProgramCounter (Q, D, clk, reset, enable);  
    input  clk, reset, enable;  
    input  [31:0] D;  
    output [31:0] Q;  
    // implementation details are left to the student  
endmodule
```

- 2) **IF\_ID Register**

We need to build the pipeline register between fetch and decode stages this register is a 64-bit register with positive edge trigger. Your module should use the following template.

```
module IFID (Q, D, clk, reset, enable);  
    input  clk, reset, enable;  
    input  [63:0] D;  
    output [63:0] Q;  
    // implementation details are left to the student  
endmodule
```

### 3) ID\_EX Register

We need to build the pipeline register between decode and execute stages this register is a 154-bit register with positive edge trigger. Your module should use the following template.

```
module IDEX (Q, D, clk, reset, enable);
  input  clk, reset, enable;
  input  [153:0] D;
  output [153:0] Q;
  // implementation details are left to the student
endmodule
```

### 4) EX\_MEM Register

We need to build the pipeline register between execute and memory stages this register is a 106-bit register with positive edge trigger. Your module should use the following template.

```
module EXMEM (Q, D, clk, reset, enable);
  input  clk, reset, enable;
  input  [105:0] D;
  output [105:0] Q;
  // implementation details are left to the student
endmodule
```

### 5) MEM\_WB Register

We need to build the pipeline register between execute and memory stages this register is a 104-bit register with positive edge trigger. Your module should use the following template.

```
module MEMWB (Q, D, clk, reset, enable);
  input  clk, reset, enable;
  input  [103:0] D;
  output [103:0] Q;
  // implementation details are left to the student
endmodule
```

## ● The Processor Module

Once you have implemented the previous modules, you can proceed and connect them to the modules you have built in earlier experiments. Your module should use the following template.

```
module PipelinedProcessor(clk, reset, enable);
  input  clk, reset, enable;

  // implementation details are left to the student

endmodule
```

## Testing

- **(Prelab.)** It is required to test your design for the entire processor by filling the instruction memory by the instruction sequence shown in the following table. You need to determine the machine code for these instructions based on Table 1 in Experiment 5.

**Table 1. The Content of the Instruction Memory**

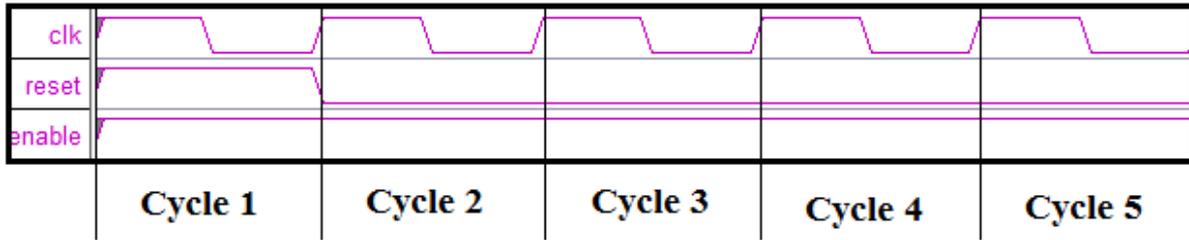
Address	Instruction	Machine Code
00	LW R1, 4(R0)	8C010004 <sub>h</sub>
01	LW R2, 12(R0)	
02	LW R3, 20(R0)	
03	LW R4, 28(R0)	
04	NAND R5, R1, R2	
05	NORI R6, R3, 1023	
06	SUB R7, R4, R2	
07	XOR R8, R5, R4	
08	ANDI R9, R6, 2047	
09	SW R6, 8(R0)	
10	LW R10, 8(R0)	
11	OR R11, R7, R8	
12	SLT R12, R1, R4	

- **(Prelab.)** Next, write a Verilog test module to test your processor module, your test module should run for **18** cycles.

**Table 2. The Test Data for the Processor**

Cycle #	clk	enable	reset
1	1 to 0 to 1	1	1
2	1 to 0 to 1	1	0
3	1 to 0 to 1	1	0
4	1 to 0 to 1	1	0
5	1 to 0 to 1	1	0
6	1 to 0 to 1	1	0
7	1 to 0 to 1	1	0
8	1 to 0 to 1	1	0
9	1 to 0 to 1	1	0
10	1 to 0 to 1	1	0
11	1 to 0 to 1	1	0
12	1 to 0 to 1	1	0
13	1 to 0 to 1	1	0
14	1 to 0 to 1	1	0
15	1 to 0 to 1	1	0
16	1 to 0 to 1	1	0
17	1 to 0 to 1	1	0
18	1 to 0 to 1	1	0

The waveform for the clock, reset and enable signals should be similar to the following one:



**Your timing diagram should contain the following signals:**

- a) *Clock, reset, and enable.*
- b) *PC (The output of the program counter).*
- c) *Instruction (The output of the instruction memory).*
- d) *The writedata, readreg1, readreg2, and writereg for the register file.*
- e) *The output for the registers R5, R6, R7, R8, R9, R10, R11.*
- f) *The input and the output of the ALU (a, b, m, result).*
- g) *The output of the data memory.*

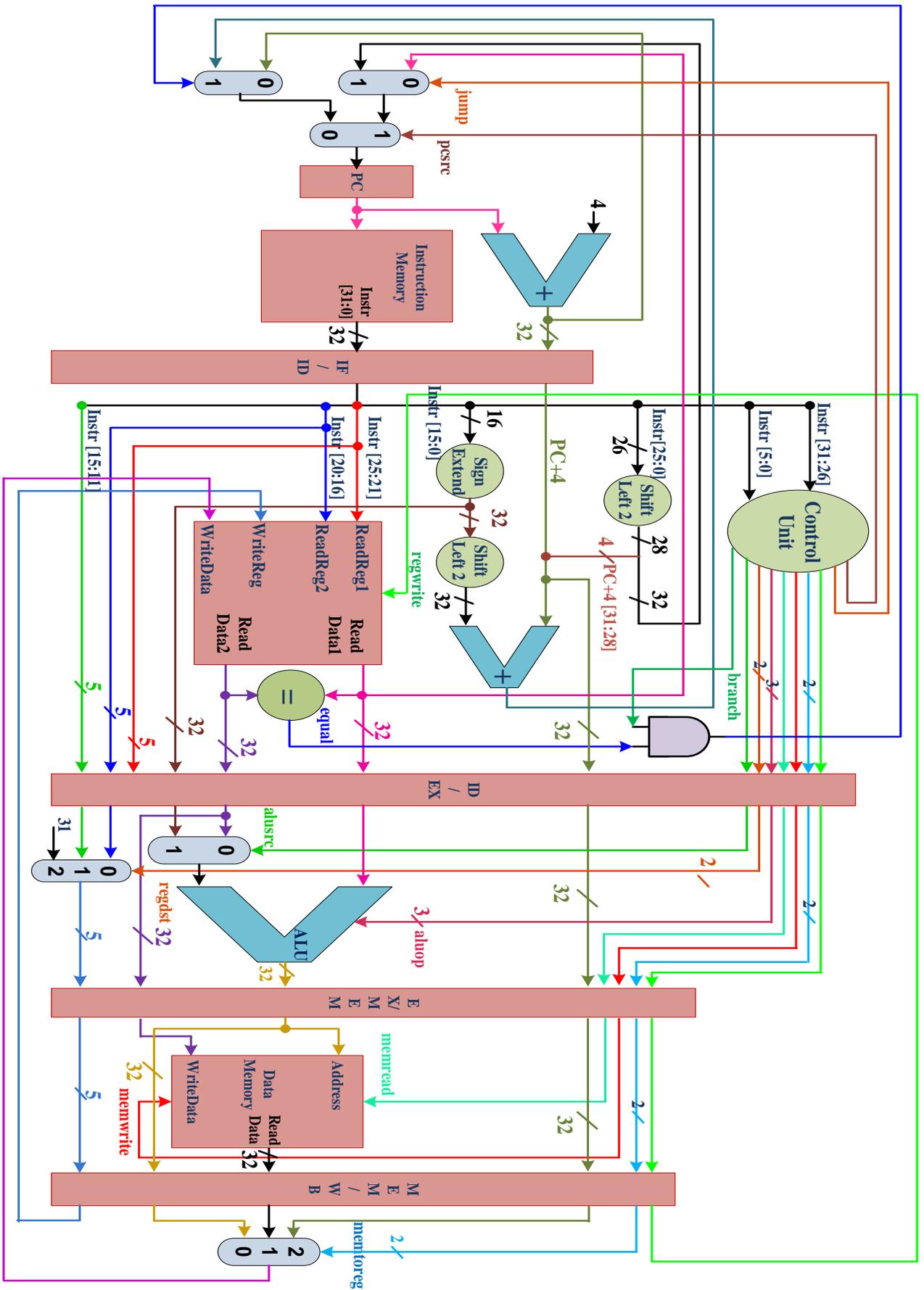


Figure 1. The Datapath for Pipelined Processor.

# University of Jordan

## Computer Engineering Department

### CPE439: Computer Design Lab

#### Experiment 8: Resolving Data Hazards

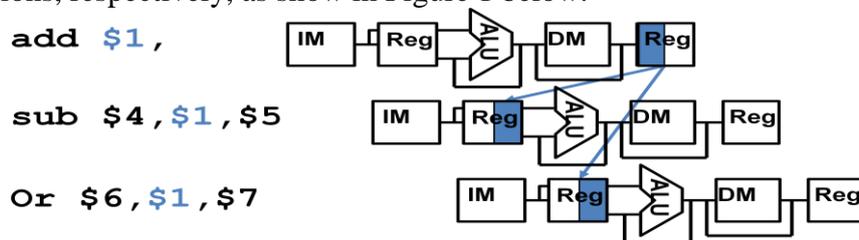
### Description

In this experiment, students have to add a forwarding unit that is capable of resolving register-use data hazards for the pipelined processor that they implemented in experiment 7.

Register-use data hazards occur when there is dependence between consecutive instructions that are being executed in the pipeline. Specifically, when the registers read by a later instruction are effectively the destination for an earlier instruction, data hazards occur. Consider for example

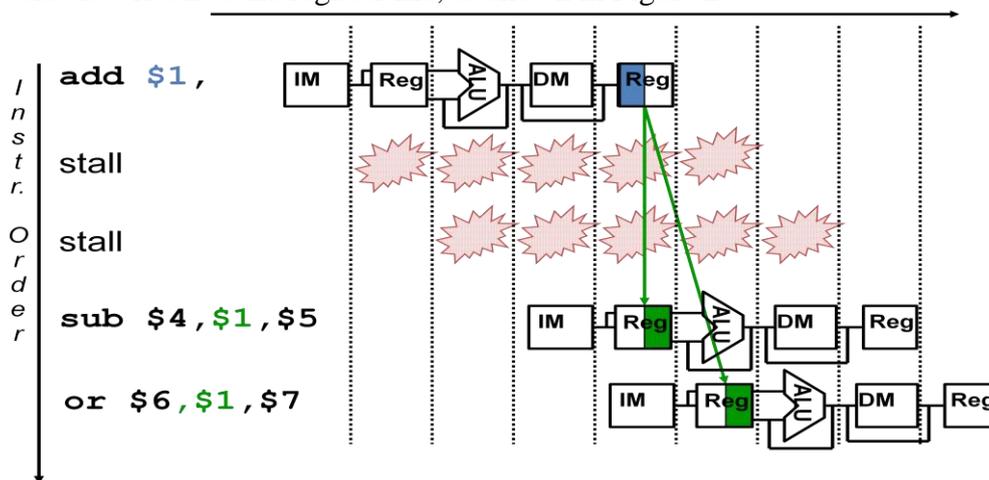
add \$1, \$2, \$3  
 sub \$4, \$1, \$5  
 or \$6, \$1, \$7

The last two instructions need to use the new value of \$1. However, the new value is written by the first instruction in the fifth cycle while it is needed in the second and third cycles for the second and third instructions, respectively, as show in Figure 1 below.



**Figure 1. Illustration of data hazards.**

In order to obtain correct operation, one solution would be to stall the pipeline for two cycles to wait until the value is written to the register file, as shown in Figure 2.



**Figure 2. Solving data hazard by stalling the pipeline.**

However, this solution affects the performance of the pipeline. Alternatively, we know that the new value for \$1 is computed and stored in the EX/MEM register by the end of the third cycle. So, we can use this value before it is written to the register file by forwarding to the ALU input and use it instead of the old value(s). Note how the value should be forwarded from the EX/MEM for the second instruction and from the MEM/WB register for the third instruction to the ALU inputs as shown in Figure 3. In other words, the inputs to the ALU are no longer the values read from the register file when the data hazard exists.

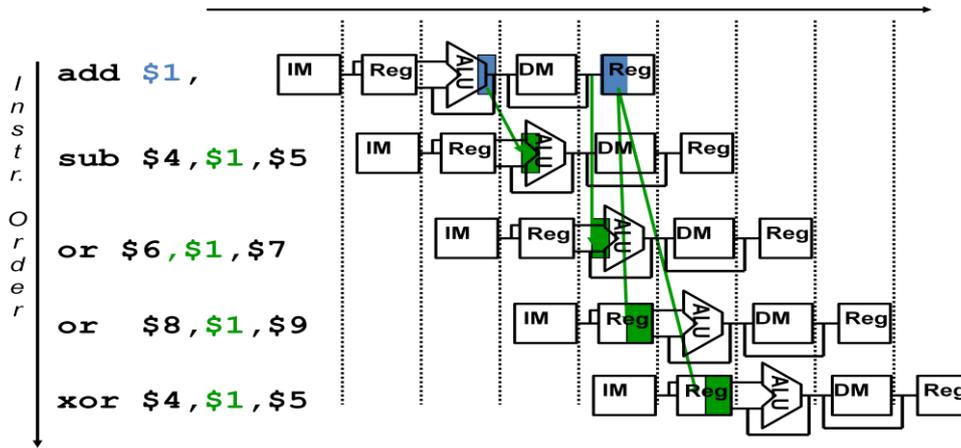


Figure 3. Solving data hazard by forwarding.

The forwarding hardware is essentially a logic circuit that consists of a set of comparators that compare the destination and source registers for consecutive instructions in addition to a set of multiplexors connected to the ALU inputs as shown in Figure 4.

If the source register(s) (rs and/or rt) for some instruction that has been decoded (stored in the ID/EX register) matches the destination register for the instruction that has passed the execute stage (stored in the EX/MEM register), then the input to the ALU should be the ALU result found in the EX/MEM register instead of the values read for the conflicting instruction in the decode stage. The same argument holds for the case when the source register(s) for an instruction matches the destination register for an earlier instruction that has finished the memory stage (stored in the MEM/WB register).

Basically, the forwarding unit hardware should implement the following conditions

1) *Forwarding the memory stage*

- a. **if (EX/MEM.RegWrite && (EX/MEM.RegRd == ID/EX.RegRs) && (EX/MEM.RegRd != 0))**

**ForwardA[0] = 1;**

- b. **if (EX/MEM.RegWrite && (EX/MEM.RegRd == ID/EX.RegRt) && (EX/MEM.RegRd != 0))**

**ForwardB[0] = 1;**

2) *Forwarding the write-back stage*

- a. **if (MEM/WB.RegWrite && (MEM/WB.RegRd == ID/EX.RegRs) && ((EX/MEM.RegRd != ID/EX.RegRs) || (EX/MEM.RegWrite==0)) && (MEM/WB.RegRd != 0))**

**ForwardA[1] = 1;**

- b. **if (MEM/WB.RegWrite and (MEM/WB.RegRd == ID/EX.RegRt) and ((EX/MEM.RegRd != ID/EX.RegRt) || (EX/MEM.RegWrite==0)) && (MEM/WB.RegRd != 0))**

**ForwardB[1] = 1;**

The ForwardA and ForwardB signals are outputs from the forwarding unit and are used to select the proper input to the ALU.

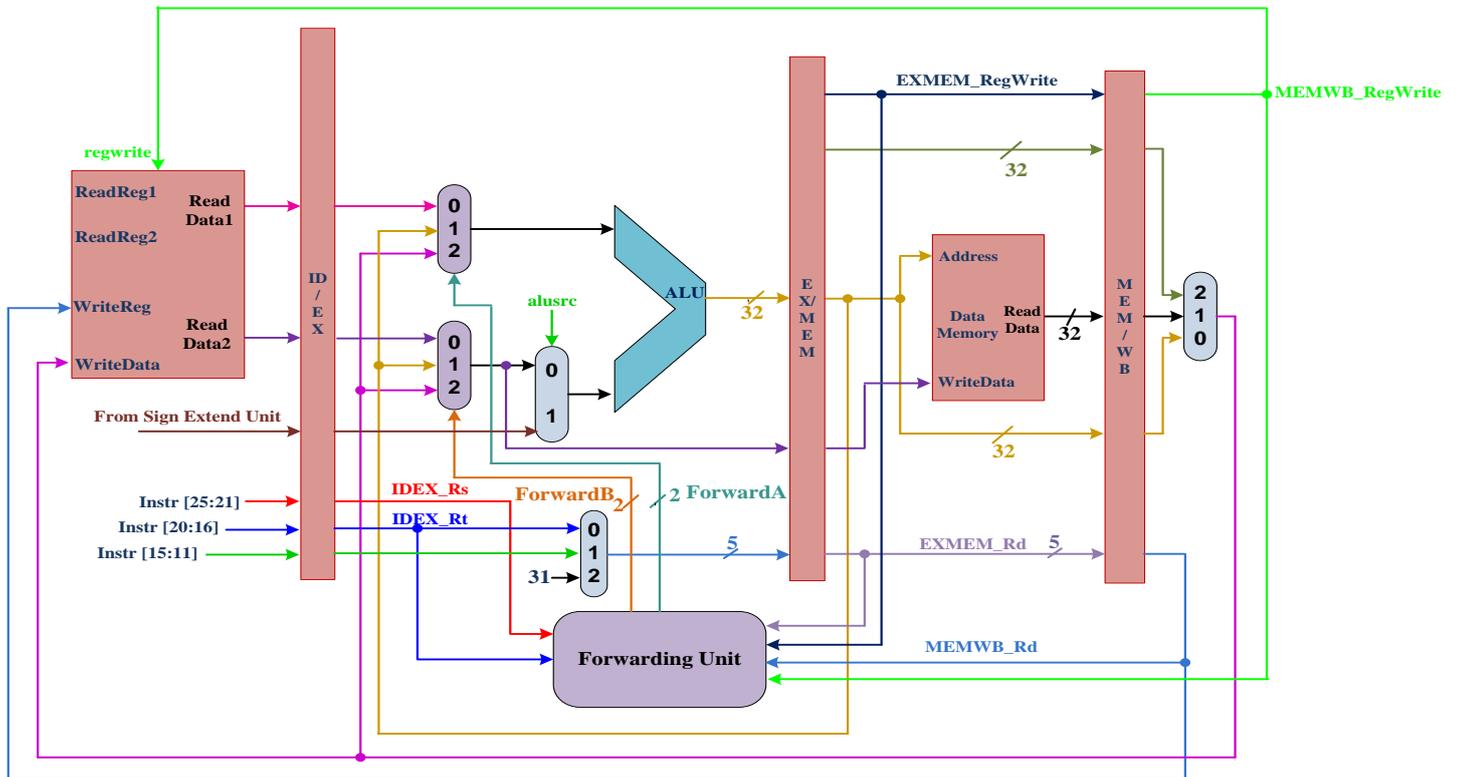


Figure 4. Incorporating forwarding within the pipeline.

## Procedure

In order to incorporate forwarding in your design you need to implement the forwarding unit and use 32-bit 3-to-1 multiplexers at the ALU inputs. Then, you should wire these new modules with the pipelined implementation as shown in Figure 4.

### 1) *(Prelab.)* 5-Bit Comparator

You need to write a structural Verilog module for 5-bit comparator. Your module should use the following template:

```
module Comparator5bit (equal, a, b);
  input [4:0]a, b;
  output equal;
  // implementation details are left to the student
endmodule
```

### 2) The Forwarding Unit

You need to build the forwarding unit structurally using 5-bit comparator and any necessary gates. Your module should use the following template:

```
module ForwardingUnit(ForwardA, ForwardB, EXMEM_Rd,
  MEMWB_Rd, IDEX_Rs, IDEX_Rt,
  EXMEM_RegWrite, MEMWB_RegWrite);

  input [4:0] EXMEM_Rd, MEMWB_Rd, IDEX_Rs, IDEX_Rt;
  input EXMEM_RegWrite, MEMWB_RegWrite;
  output [1:0]ForwardA, ForwardB;
  // implementation details are left to the student
endmodule
```

### 3) The processor module

You need to modify the pipelined processor module by adding the forwarding unit and ALU multiplexers and any needed modifications.

## Testing

- **(Prelab.)** Write the Verilog module to **test your forwarding unit**. The test module for this unit should use the data given in Table 1 as a benchmark,

**Table 1. Test data for Forwarding Unit**

EXMEM_Rd	MEMWB_Rd	IDEX_Rs	IDEX_Rt	EXMEM_RegWrite	MEMWB_RegWrite
5'b00001	5'b00001	5'b00001	5'b00001	0	0
5'b00001	5'b00011	5'b00001	5'b00000	1	0
5'b00001	5'b00001	5'b00001	5'b00001	0	1
5'b00011	5'b00010	5'b00101	5'b00010	1	1
5'b00101	5'b00101	5'b00101	5'b00110	1	1

- **(Prelab.)** Next, it is required to test your design for the pipelined processor by filling the **instruction memory module** by the instruction sequence shown in Table 2.

**Table 2. The Content of the Instruction Memory**

Address	Instruction	Machine Code
00	LW R1, 4 (R0)	8C010004 <sub>h</sub>
01	LW R2, 12 (R0)	
02	LW R3, 20 (R0)	
03	LW R4, 28 (R0)	
04	ADD R5, R2, R1	
05	AND R6, R5, R5	
06	SLTI R6, R5, 8	
07	OR R7, R2, R4	
08	NAND R7, R2, R4	
09	NOR R8, R7, R7	

- **(Prelab.)** Next, write a Verilog **test module** to test your processor module
  - **Your Timing diagram should contain the following signals:**
    - PC (The output of the program counter).*
    - Instruction (The output of the instruction memory).*
    - The writedata, readreg1, readreg2, and writereg for the register file.*
    - The output for the registers R5, R7, R8.*
    - The input and the output of the ALU (a, b, m, result).*
    - The output of forwarding unit (ForwardA, ForwardB).*
  - **Calculate number of cycles needed to execute the above code.**

**University of Jordan**  
**Computer Engineering Department**  
**CPE439: Computer Design Lab**  
**Experiment 9: Resolving Control Hazards**

### **Description**

In the previous experiment, student worked on resolving one out of several cases where data dependencies between instructions may cause data hazards in pipelining. In this experiment, students have to modify their pipelining implementation to accommodate for a new type of pipelining hazards; namely, control hazards.

Control hazards arise when executing program flow control instructions such as beq, j, jr, and jal. When these instructions are being executed (stored in the IF/ID register), the processor is fetching the following instruction (at PC+4). However, when execution is over (the decoding of the flow instruction is over and it is stored in the ID/EX register), the fetched instruction (stored in IF/ID register) might not be correct for conditional flow instructions (beq) if the condition evaluates to true. In this case, the processor should have fetched the instruction pointed-to by the branch address. Similarly, for unconditional flow instructions (j, jr, and jal), the fetched instruction is always wrong since it has to be fetched from the jump address for j and jal instructions, and from the address contained in one of the registers for the jr instruction.

In order to resolve this hazard, the fetched instruction in both cases has to be removed (flushed) from the pipeline. This can be implemented by clearing or flushing the IF/ID register asynchronously after the instruction is stored and the hazard is detected. Note how this affects the performance of the pipeline since it wastes one cycle.

In this experiment, students have to resolve control hazard by designing a hazard detection hardware that is capable of determining the need for flushing the fetched instruction or not based on the type of the instruction in the decode stage, and then incorporate it within the pipelined implementation done in experiment 8, as shown in Figure 1.

### **Procedure**

1) ***(Prelab.)*** Hazard Detection Unit

You need to build the hazard detection unit structurally. Your module should use the following template:

```
module HazardDetectionUnit(Flush, pcsrc, takenbranch);
    output Flush;
    input pcsrc, takenbranch;

    // implementation details are left to the student
endmodule
```

2) **The processor module**

You need to modify the pipelined processor module by **adding the hazard detection unit** and **make the needed modifications**.

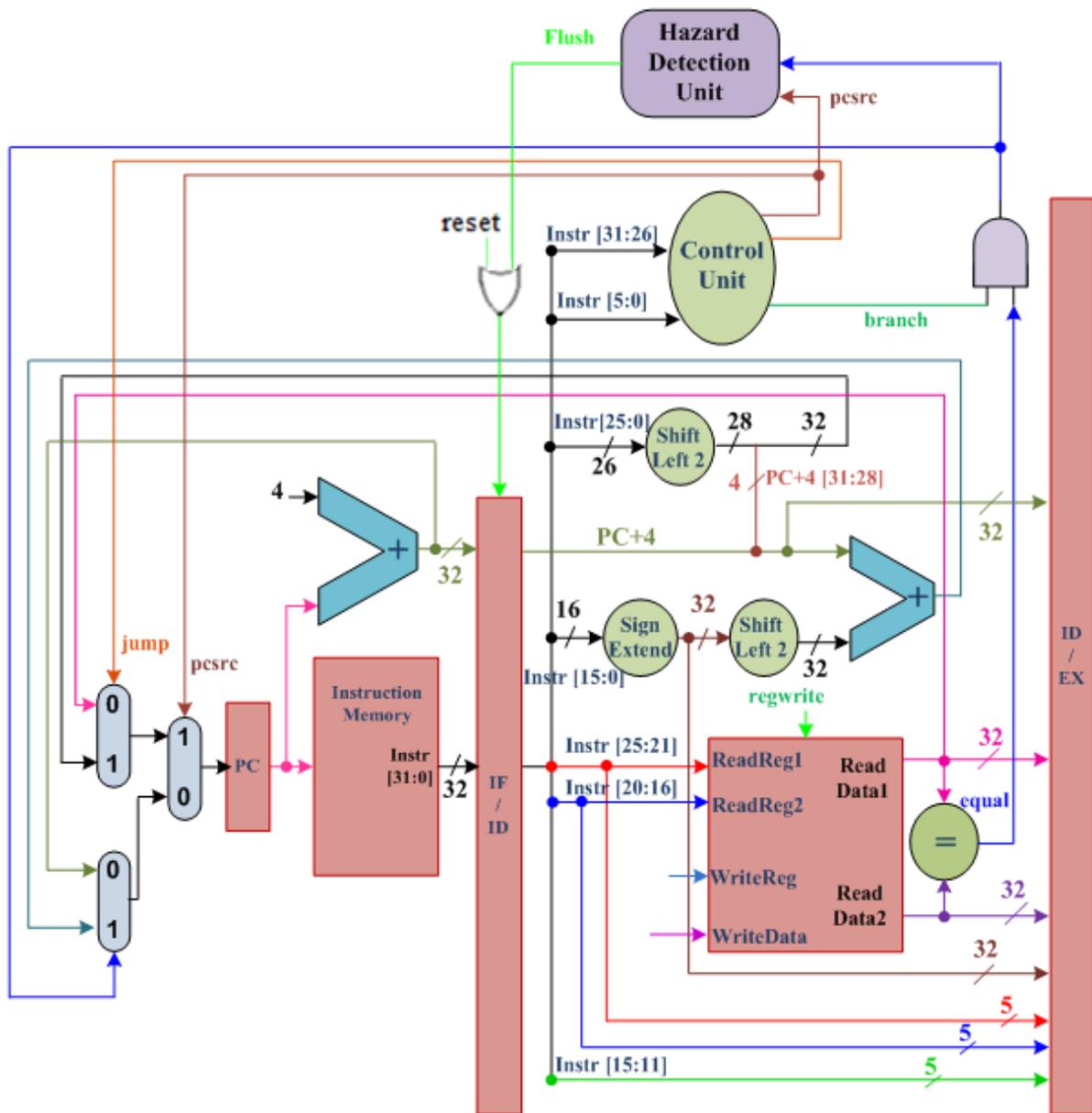


Figure 1

## Testing

- **(Prelab.)** Test your design for the pipelined processor by filling the instruction memory by the instruction sequence shown in Table 1.

**Table 1. The Content of the Instruction Memory**

Address	Instruction	Machine Code
00	LW R1, 4(R0)	8C010004 <sub>h</sub>
01	LW R2, 12(R0)	
02	LW R3, 20(R0)	
03	LW R4, 28(R0)	
04	NAND R5, R1, R3	
05	NORI R6, R5, 1023	
06	SUB R8, R4, R2	
07	JAL 11	
08	XOR R7, R5, R6	
09	SW R7, 8(R0)	
10	J 19	
11	ADDI R8, R8, 2	
12	SW R5, 4(R0)	
13	SW R6, 24(R0)	
14	BEQ R8, R3, -4	
15	SUB R9, R8, R3	
16	JR R31	
17	OR R10, R7, R9	
18	SLT R11, R9, R4	

- **(Prelab.)** Next, write a Verilog test module to test your processor module
  - **Your timing diagram should contain the following signals:**
    - a) PC (The output of the program counter).
    - b) The output of IFID register.
    - c) The output for the registers R5, R6, R7, R8, R9, R10, R11.
    - d) The output of forwarding unit (ForwardA, ForwardB).
    - e) The input and the output of hazard detection unit.
  - Calculate number of cycles needed to execute the above code.