University of Jordan

Computer Engineering Department

CPE 439: Computer Design Lab

Verilog Overview

What is Verilog?

- It is a Hardware Description Language (HDL).
- Two major HDL languages:
  - Verilog
  - VHDL
- Verilog is easier to learn and use (It is like the C language).

Cont.

- Introduced in 1985 by (Gateway Design Systems)

  Now, part of (Cadence Design Systems).

- 1990: OVI (Open Verilog International).

- 1995: IEEE Standard.

- **Simulator:**

  Verilogger Pro, from Synapticad INC.

Why Use Verilog?

- Digital systems are highly complex.

  ➔Schematic is useless (just connectivity)

- Behavioral Constructs:

  - Hide implementation details.
  - Arch. Alternatives through simulations.
  - Detect design bottlenecks.

  (BEFORE DETAILED DESIGN BEGINS)

- ***Automated Tools*** compile behavioral models to actual circuits.

Lexical Conventions

- Close to C / C++.
- **Comments:**

```
// Single line comment
/* multiple
       lines
          comments   */
```

- Case Sensitive
- **Keywords:**

  - Reserved.
  - lower case.
  - Examples: **`module, case, initial, always`**.

- **Numbers:**

<size>'<base format><number>

- **Size:** No. of bits (optional).
- **Base format:** ➔ **b: binary**

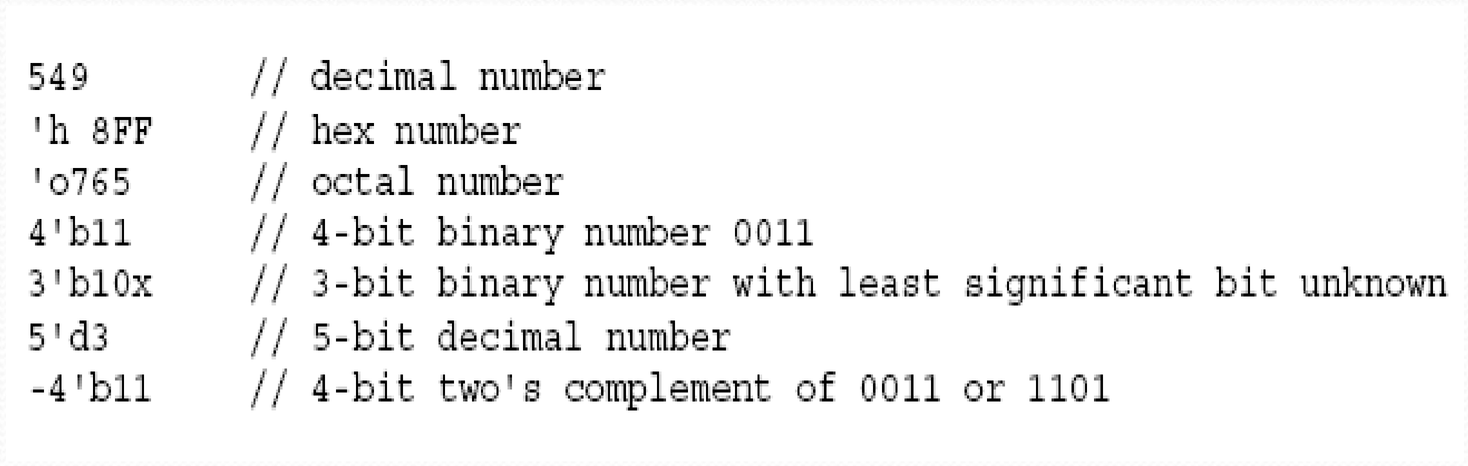
➔ **d : decimal**

➔ **o : octal**

➔ **h : hexadecimal**

(DEFAULT IS DECIMAL)

Examples:



```
549         // decimal number
'h 8FF      // hex number
'o765       // octal number
4'b11       // 4-bit binary number 0011
3'b10x      // 3-bit binary number with least significant bit unknown
5'd3        // 5-bit decimal number
-4'b11      // 4-bit two's complement of 0011 or 1101
```

- **Identifiers:**

  - Start with letter or (_).

  - A combination of letters, digits, ($), and(_).

  - Up to 1024 characters.

Program Structure

- Verilog describes a system as a set of **modules.**
- Each module has an interface to other modules.
- **Usually:**

  - Each module is put in a separate file.

  - One top level module that contains:

    → Instances of hardware modules.

    →Test data.
- **Modules can be specified:**

    - Behaviorally.

    - Structurally.
- Modules are different from subroutines in other languages (never called).

Physical Data Types

- **Registers (reg):**

  - Store values

```
reg X;                  // 1-bit register
reg [7:0] A;            // 8-bit register
```

- **Wires (wire):**

  - Do not store a value.

  - Represent physical connections between entities.

```
wire X;
wire [7:0] A;
```

- **reg** and **wire** data objects may have a value of:

  - 0 : logical zero
  - 1 : logical one
  - x : unknown
  - z : high impedance

- Registers are initialized to **x** at the start of simulation.

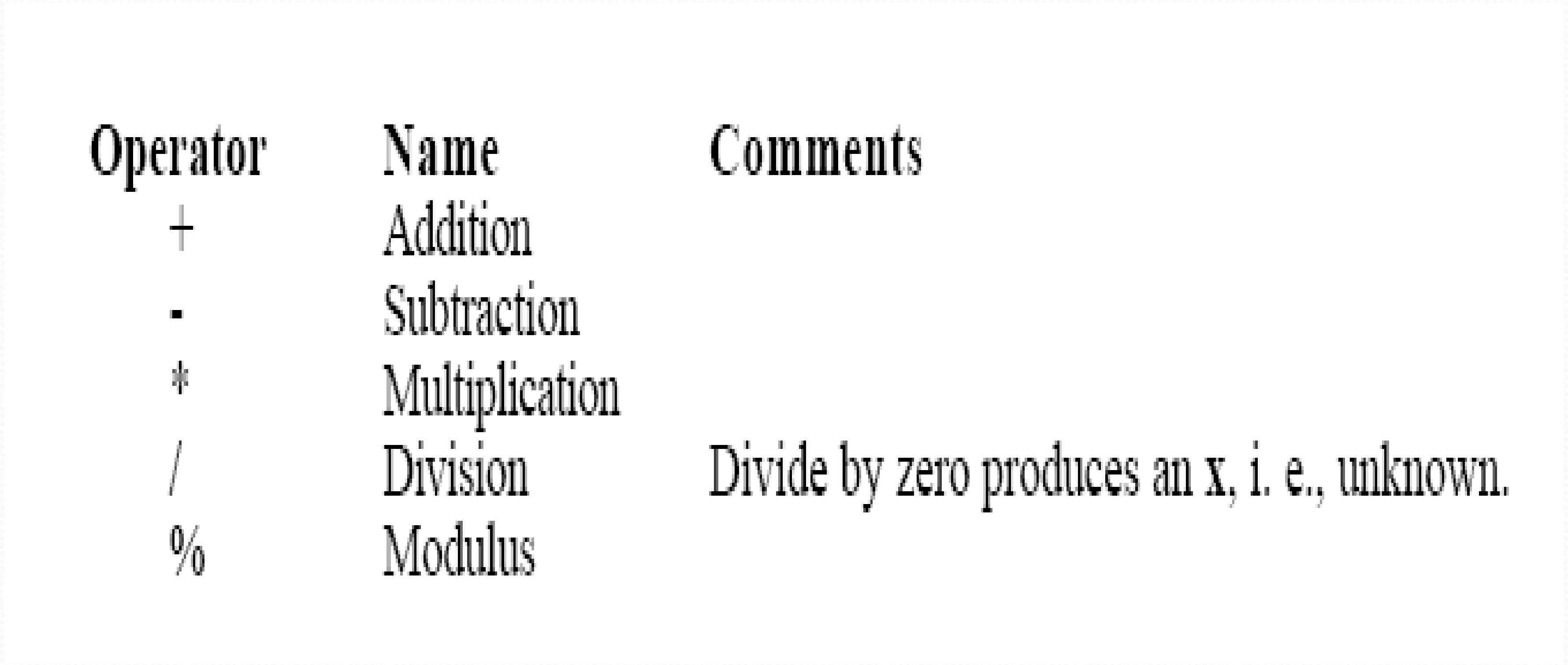
- Any wire not connected to something has the value **x**.

- **How to declare a memory in verilog?**

```
reg [7:0] A [1023:0];// A is 1K words each 8-bits
```

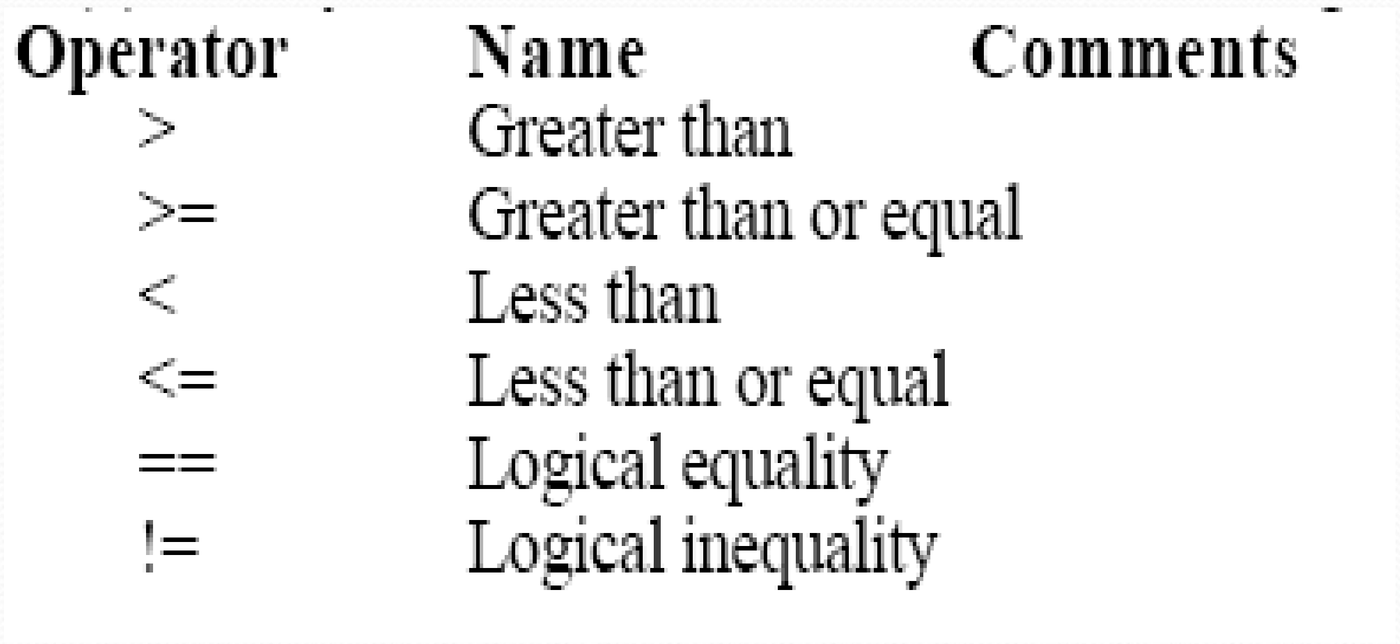
Operators

Binary Arithmetic Operators:



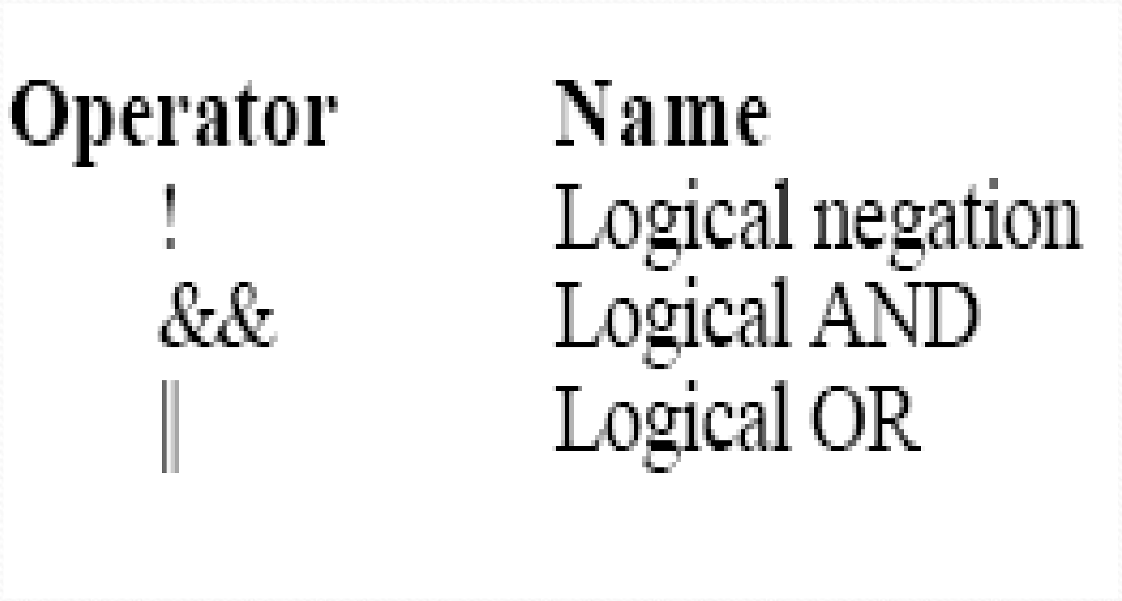
Operator	Name	Comments
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	Divide by zero produces an x, i. e., unknown.
%	Modulus	

Relational Operators:



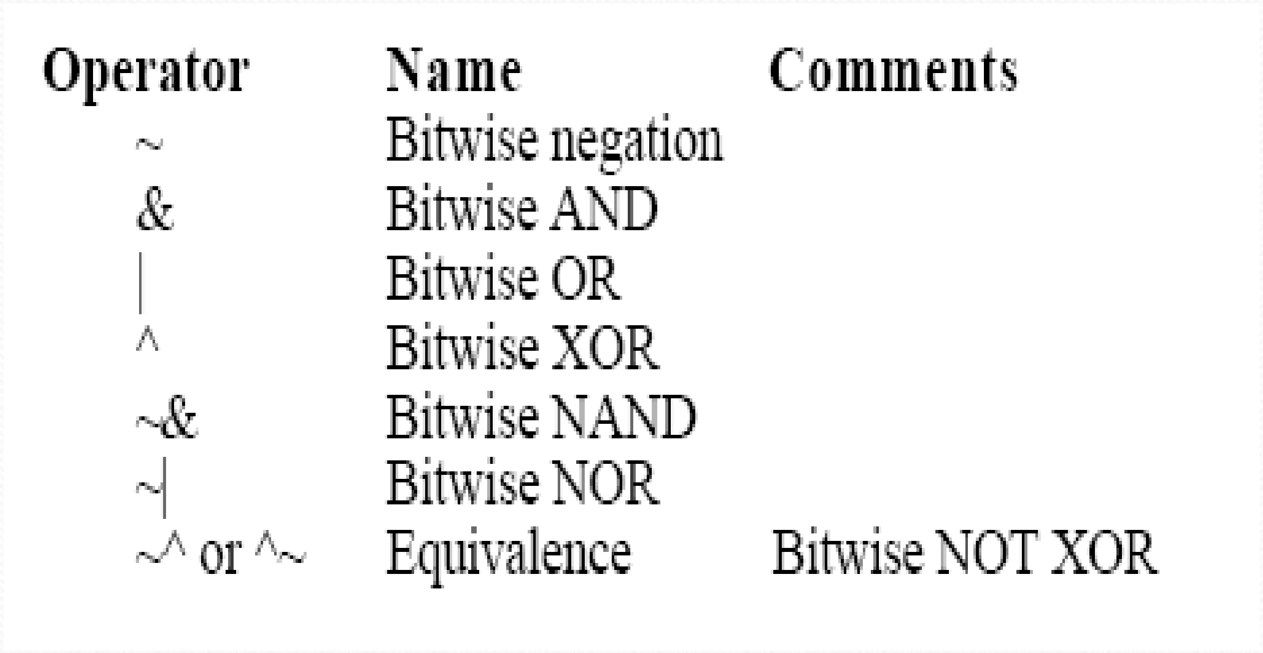
Operator	Name	Comments
>	Greater than	
>=	Greater than or equal	
<	Less than	
<=	Less than or equal	
==	Logical equality	
!=	Logical inequality	

Logical Operators:



Operator	Name
!	Logical negation
&&	Logical AND
||	Logical OR

Bitwise Operators:



Operator	Name	Comments
~	Bitwise negation	
&	Bitwise AND	
|	Bitwise OR	
^	Bitwise XOR	
~&	Bitwise NAND	
~|	Bitwise NOR	
~^ or ^~	Equivalence	Bitwise NOT XOR

Other operators:

Concatenation:

```
{A[0], B[7:1]}
```

Shift Left:

```
A = A << 2 ;        // right   >>
```

Conditional:

```
A = C > D  ?  B + 3 : B - 2 ;
```

IF Statement

- Similar to C/C++.
- Instead of { } , use `begin` and `end`.

```
 if (A == 4)
  begin
      B = 2;
  end
else
  begin
      B = 4;
  end
```

Case Statement

- Similar to C/ C++
- No `break` statements are needed.

```
case ( A )
 1'bz: $display("A is high impedance");
 1'bx: $display("A is unknown");
 default: $display("A = %b", A);
endcase
```

Repetition `for`, `while` and `repeat` Statements

- The `for` statement is very close to C's **for** statement except that the ++ and -- operators do not exist in Verilog. Therefore, we need to use **i = i + 1**.

```
for(i = 0; i < 10; i = i + 1)
   $display("i= %0d", i);
```

- The `while` statement acts in the normal fashion.

```
 i = 0;
while(i < 10)
begin
  $display("i= %0d", i);
  i = i + 1;
 end
```

Example: NAND Gate

```
module   NAND (out, in2, in1);
  input   in1, in2;
  output out;
  assign #2 out = ~(in1 & in2);// Continuous
                               // Assignment
endmodule
```

AND Gate

```
module  AND (out, in2,in1);
  input  in1, in2;
  output  out;
  wire w1;
  NAND n1(w1, in2, in1);
  NAND n2(out, w1, w1) ;
endmodule
```

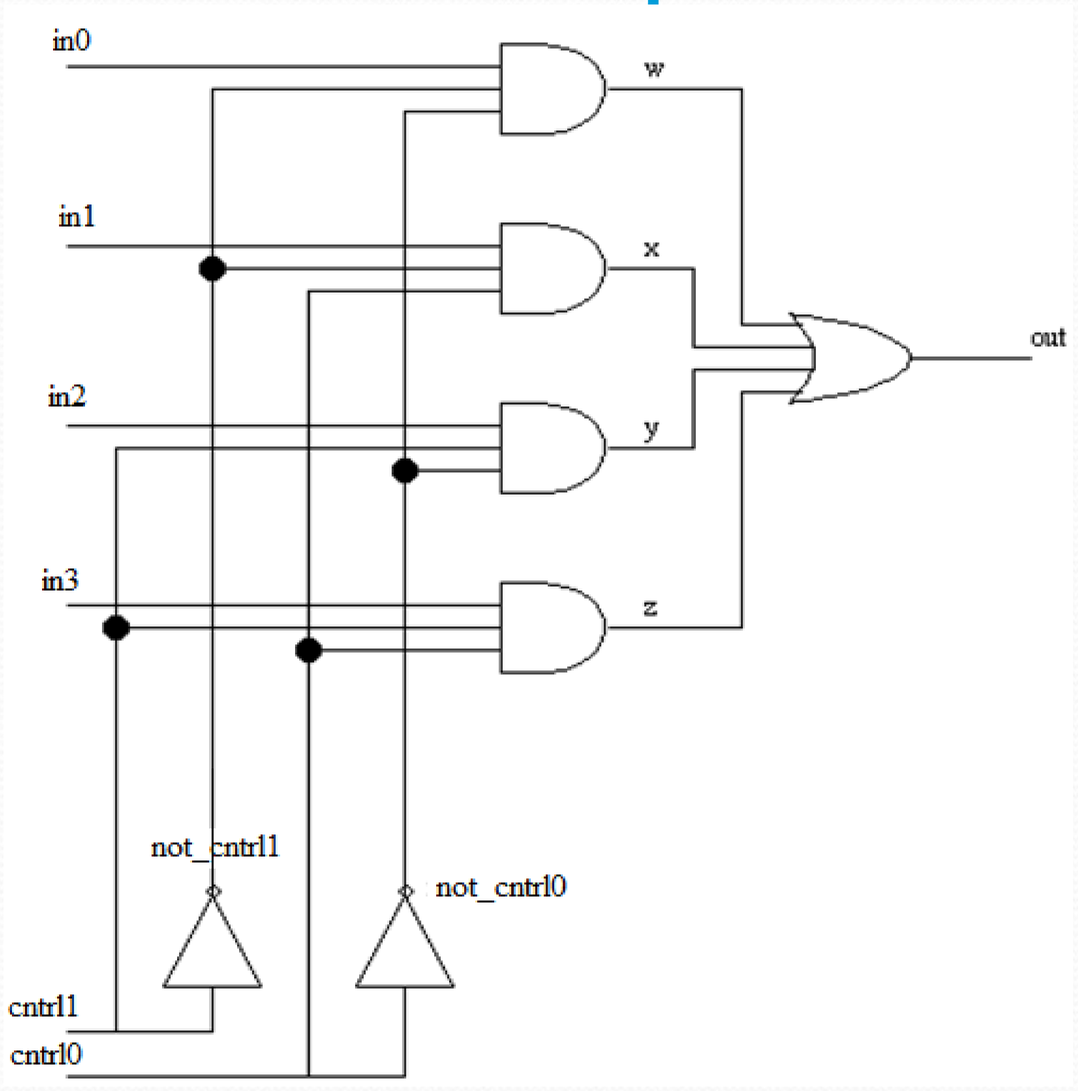
Test

```
module TestAND;
reg in1,in2;
wire out;

AND a1(out, in2, in1);

initial begin :init
     in2=0; in1=0;
#10 in2=0; in1=1;
#10 in2=1; in1=0;
#10 in2=1; in1=1;
#10;
end

initial begin
$display("Time     in2    in1    out");
$monitor("%0d      %b     %b     %b", $time, in2, in1, out);
end

endmodule
```

Output

Time	in2	in1	out
0	0	0	x
4	0	0	0
10	0	1	0
20	1	0	0
30	1	1	0
34	1	1	1

4-to-1 Multiplexor



Structural Gate-level model

```
module  MUX_4x1 (out ,in4 , in3 , in2, in1 , cntrl0, cntrl1);
  output out;
  input in1, in2, in3, in4, cntrl0, cntrl1;
  wire not_cntrl0, not_cntrl1, w, x, y, z;

  INV   n0 (not_cntrl0, cntrl0);
  INV   n1 (not_cntrl1, cntrl1);

  AND3  a0 (w, in0, not_cntrl1, not_cntrl0);
  AND3  a1 (x, in1, not_cntrl1, cntrl0);
  AND3  a2 (y, in2, cntrl1, not_cntrl0);
  AND3  a3 (z, in3, cntrl1, cntrl0);

  OR4   o1 (out, w, x, y, z);
endmodule
```

Behavioral Gate-Level Model (RTL)

```
module MUX_4x1 (out ,in3 , in2 , in1, in0 , cntrl0, cntrl1);
  output out;
  input in0, in1, in2, in3, cntrl1, cntrl0;

   assign #4 out =  (in0 &  ~cntrl1 & ~cntrl0)|
                    (in1 &  ~cntrl1 &  cntrl0)|
                    (in2 &   cntrl1 & ~cntrl0)|
                    (in3 &   cntrl1 &  cntrl0);
endmodule
```

Behavioral Model

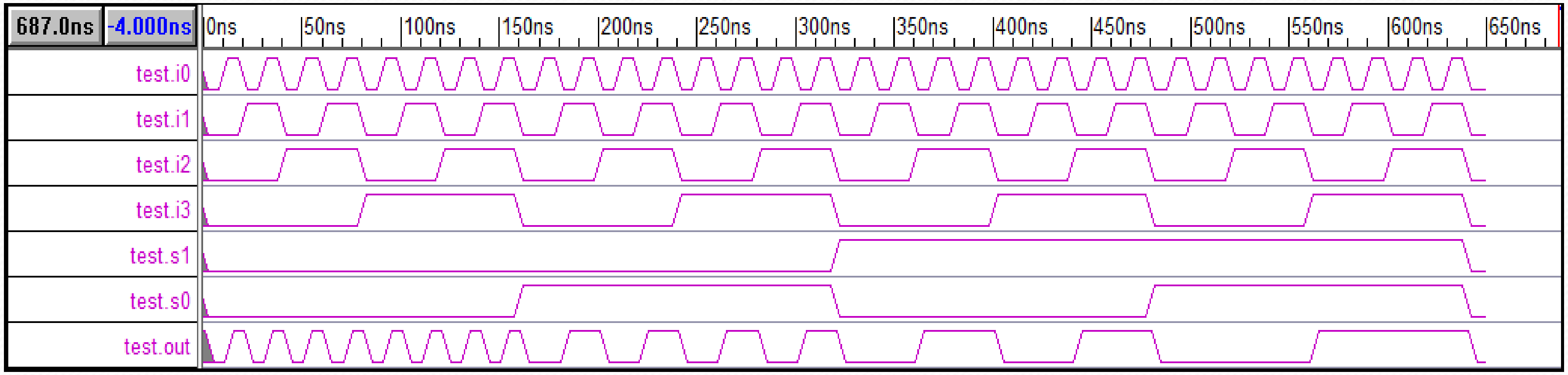
```
module MUX_4x1 (out ,in3 , in2 , in1, in0 , cntrl0, cntrl1);
  output out;
  input in0, in1, in2, in3, cntrl1, cntrl0;
  reg out;

 always @(in0 or in1 or in2 or in3 or cntrl1 or cntrl0)
    #4 case ({cntrl1, cntrl0})
       2'b00 : out = in0;
       2'b01 : out = in1;
       2'b10 : out = in2;
       2'b11 : out = in3;
    endcase
endmodule
```

Behavioral code: output out must now be of type reg as it is assigned values in a procedural block.

Test Module

```
module test;
reg i0,i1,i2,i3,s1,s0;
wire out;

MUX_4x1 mux4_1(out,i3,i2,i1,i0,s0,s1);

initial begin: stop_at
   #650; $finish;
end

initial begin :init
i0=0; i1=0; i2=0; i3=0; s0=0; s1=0;

$display("*** Mulitplexer 4 x 1 ***");
$display("Time    i3   i2   i1   i0    s1   s0      out ");
$monitor("%0d        %b       %b       %b        %b       %b       %b      %b ", $time,i3,i2,i1,i0,s1,s0,out);
end

always #10 i0 = ~i0;
always #20 i1 = ~i1;
always #40 i2 = ~i2;
always #80 i3 = ~i3;
always #160 s0 = ~s0;
always #320 s1 = ~s1;

endmodule
```

The Output


687.0ns
-4.000ns
0ns
50ns
100ns
150ns
200ns
250ns
300ns
350ns
400ns
450ns
500ns
550ns
600ns
650ns
test.i0
test.i1
test.i2
test.i3
test.s1
test.s0
test.out

T - Flip Flop

```
// Asynchronous T Flip-flop with reset (Negative-edge trigger)
module TFF (Q, T, clk, reset);
   input T, clk, reset;
   output Q;
   reg Q;
   always @(negedge clk or posedge reset)
      begin
            #2;
            if (reset == 1)
                  Q = 0;
            else if (T == 1)
                  Q = ~Q;
      end
endmodule
```

2-bit Counter

```
module counter2bit (Q, clk, reset);
output [1:0]Q;
input  clk, reset;

 //TFF (Q,      T   , clk , reset);
 TFF t0(Q[0], 1'b1, clk , reset);
 TFF t1(Q[1], 1'b1, Q[0], reset);
endmodule
```

```
module  project_test ;
reg reset, clk ;
wire [1:0] Q;

counter2bit f1(Q, clk, reset);

initial #200 $finish;

initial begin : init_block
       clk = 1'b0;
       #5 reset = 1'b1 ;
       #10 reset = 1'b0 ;
       $display ("Time                Q ");
       $monitor ("%0t               %0d ", $time, Q);
  end
always #10 clk = ~clk;

endmodule
```

University of Jordan
Computer Engineering Department
CPE439: Computer Organization 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 quatations 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 Organization 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.

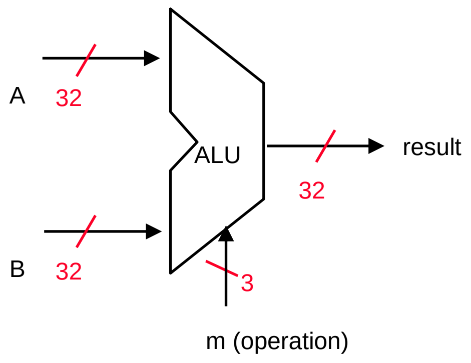


Figure 1. 32-bit ALU block diagram

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

1) 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
```

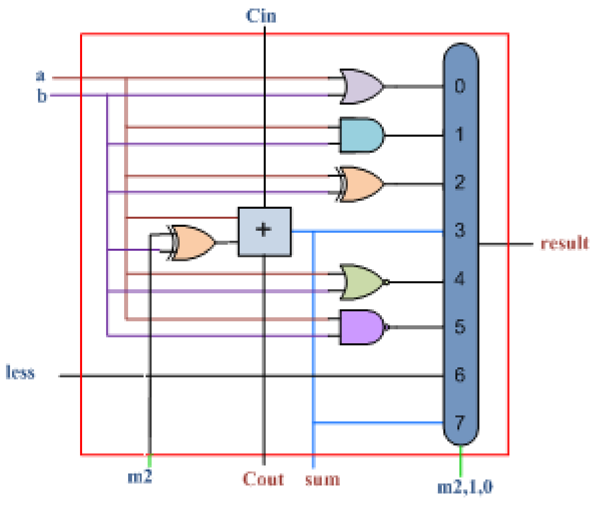
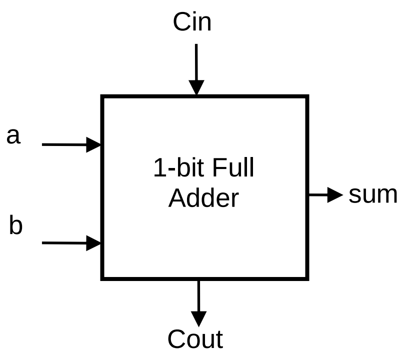


Figure 2. 1-bit ALU.



a	b	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

b) ***(Prelab.)*** *8-to-1 Multiplexor*
You should write a Verliog 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_h	$00000c0f_h$	000
00000102_h	$00000c0f_h$	001
00000102_h	$00000c0f_h$	010
00000102_h	$00000c0f_h$	100
00000102_h	$00000c0f_h$	101
00000102_h	$00000c0f_h$	110
$000f0001_h$	00000024_h	110
$000f0001_h$	00000024_h	011
$000f0001_h$	00000024_h	111

University of Jordan
Computer Engineering Department
CPE439: Computer Organization 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 multiplexors. The 5-bit read address ports, ReadReg1 and ReadReg2, are connected to the selection lines of the multiplexors to select the contents of the addressed registers.

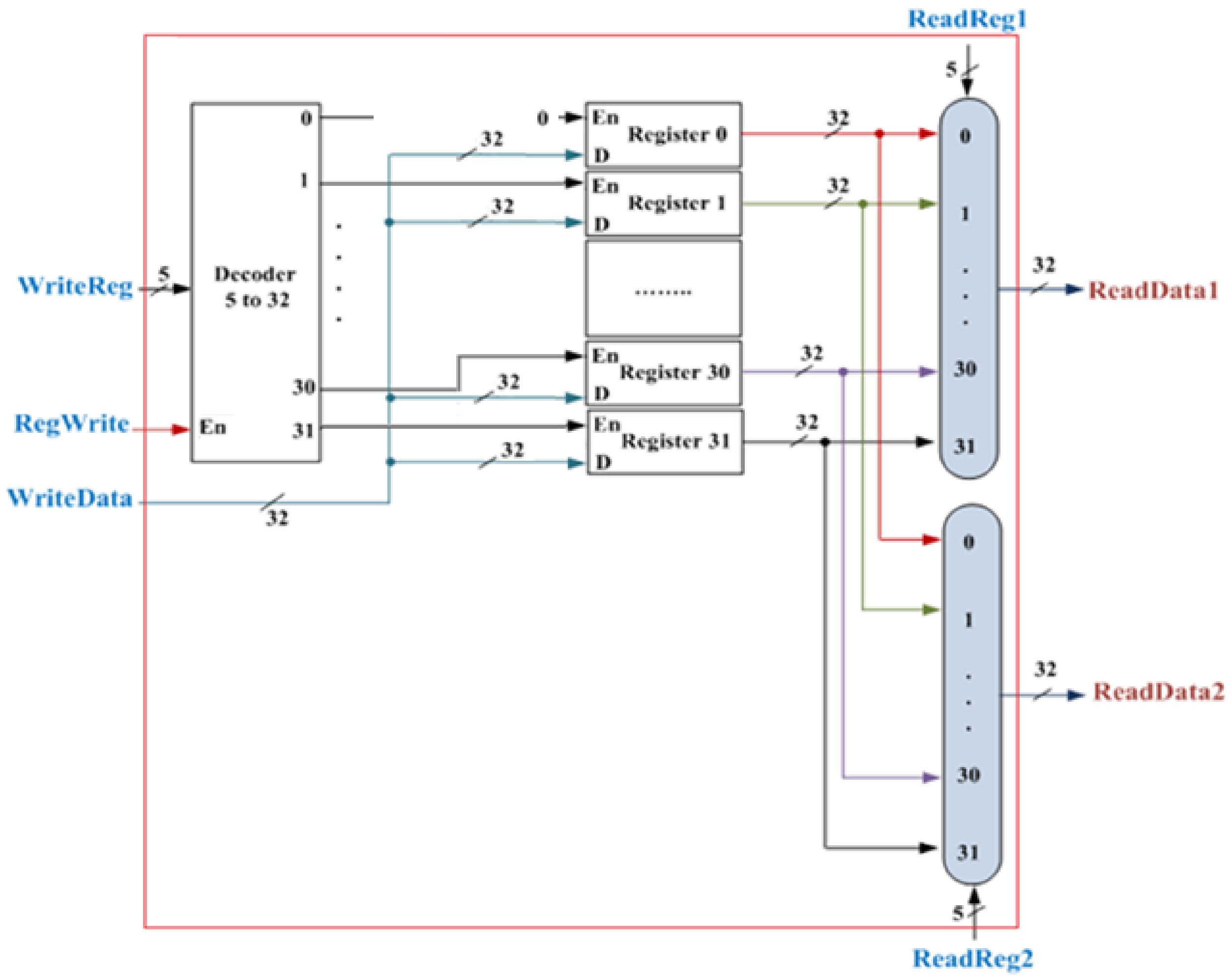


Figure 1. Layout of the register file.

- **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-to32 decoder as follows:

a) 2-to-4 Decoder

You should write a Verliog 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 Verliog 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 Verliog 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
```

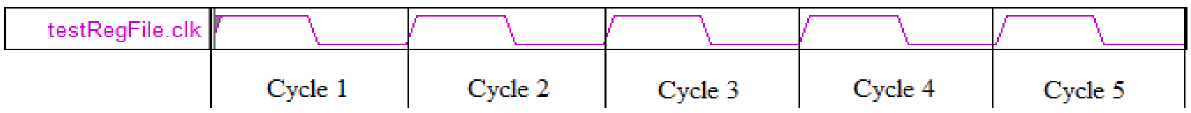
- <u>**Testing**</u>

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 <u>***estimate the maximum delay in your design.***</u>

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	$000000ff_h$	00011_b	0	00000_b	00011_b	1
2	1 to 0 to 1	00000150_h	00011_b	1	00011_b	00100_b	0
3	1 to 0 to 1	00000066_h	00100_b	1	00011_b	00100_b	0
4	1 to 0 to 1	00000008_h	00011_b	0	00011_b	01000_b	0
5	1 to 0 to 1	00000040_h	01000_b	0	00001_b	00101_b	0

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



University of Jordan
Computer Engineering Department
CPE439: Computer Organization 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.

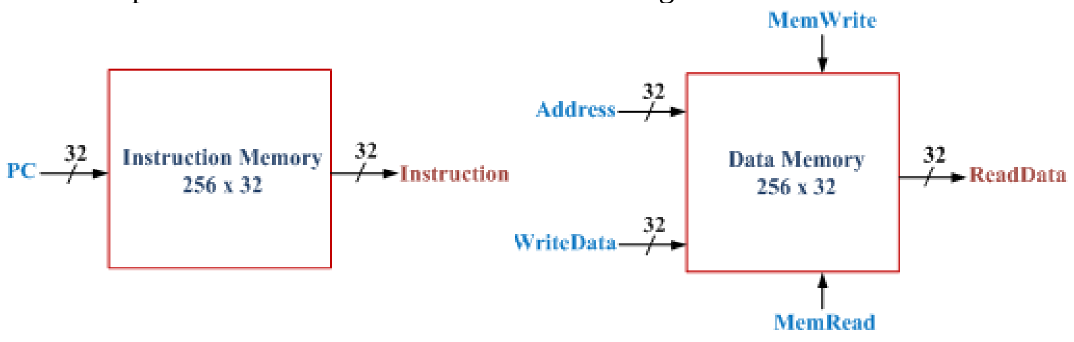


Figure 1. The Block Diagram for Instruction and Data Memories.

• 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
```

- **<u>Testing</u>**

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_h
00000004_h
00000008_h
$0000000C_h$
00000010_h
00000014_h

Table II. Test data for Data Memory

Cycle #	Clock	writedata	address	memread	memwrite
1	1 to 0 to 1	00000000_h	00000014_h	0	0
2	1 to 0 to 1	$000000e5_h$	00000014_h	1	0
3	1 to 0 to 1	$00000f14_h$	00000014_h	0	1
4	1 to 0 to 1	$0000000a_h$	00000018_h	0	1
5	1 to 0 to 1	$0000009e_h$	00000014_h	1	0
6	1 to 0 to 1	$0000007f_h$	00000018_h	1	0

University of Jordan
Computer Engineering Department
CPE439: Computer Organization 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 RISC-V 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 RISC-V instructions. The block diagram and specifications for this unit is shown in Figure 1.

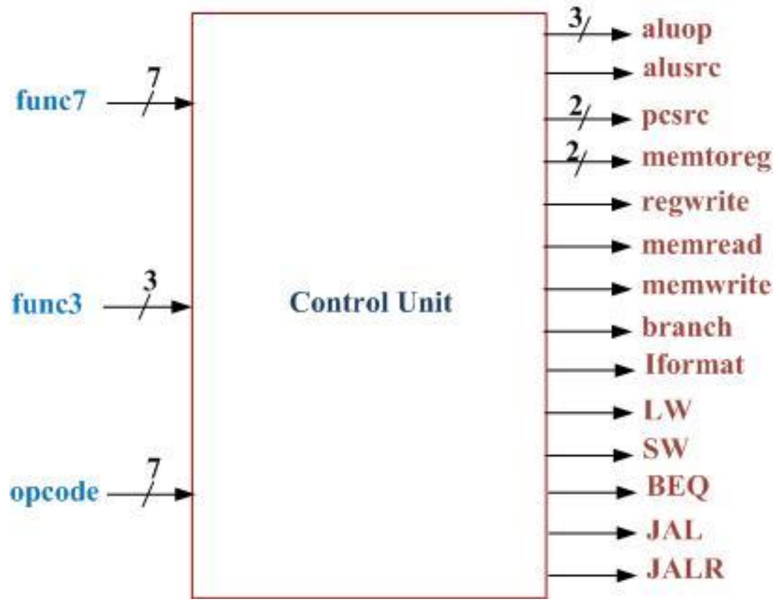


Figure 1. The Block Diagram for the Control Unit.

• 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 **behavioral** modeling. Don't attempt to use logic minimization (e.g. K-maps) as the hardware has 17 inputs. Instead, use the following approach:

1) Derive equations for signals that identify instructions from their **opcodes**. For example, a signal $\boldsymbol{LW}$ which identifies a load instruction can have the following equation:

$$\boldsymbol{LW} = \overline{op6}.\overline{op5}.\overline{op4}$$

This equation works because the load instruction is the only instruction type that has the most significant three bits of its opcode set to zero. Follow the same approach to write the equations for all instruction types. Notice that the first six instructions in the table share the same opcode; hence, only one equation is needed to identify them as R-format instructions. Similarly, the five I-format instructions share the same opcode and only one equation is needed to identify them.

Notice that six of these signals (i.e. Iformat, LW, SW, BEQ, JAL, JALR) will also be outputs of the control unit module because they will be needed in experiment 6.

2) Derive the equations for the output signals using the signals found in the first bullet. Most of the output signals (i.e. alusrc, pcsrc, memtoreg, regwrite, memread, memwrite, and branch) rely only on the opcode and are not affected by the function fields. For example, regwrite must be logic one for R-formate, I-formate, LW, JAL, and JALR instructions. Hence, the equation of regwrite can be derived as follows:

$$\boldsymbol{regwrite} = Rformat + Iformat + LW + JAL + JALR$$

3) For aluop[2:0] output signals, the equations depend on the opcode as well as the function fields and can be derived by investigating the values given in Table 1. For example, the equation of aluop[2] can be as follows:

$$\boldsymbol{aluop}[\mathbf{2}] = Rformat.\left(\overline{func3[2]}.func3[1] + func7[5]\right) + Iformat.\left(\overline{func3[2]}.func3[1]\right)$$

Your module should use the following template.

```
module ControlUnit(aluop, alusrc, pcsrc, memtoreg, regwrite,
                   memread, memwrite, branch, Iformat, LW, SW,
                   BEQ, JAL, JALR,
                   opcode, func3, func7);

  input [6:0] opcode, func7;
  input [2:0] func3;
  output [2:0] aluop;
  output [1:0] memtoreg, pcsrc;
  output alusrc, regwrite, memread, memwrite, branch, Iformat,
         LW, SW, BEQ, JAL, JALR;

     // implementation details are left to the student……

endmodule
```

instruction	opcode	func3	func7	aluop[2]	aluop[1]	aluop[0]	alusrc	pcsrc[1]	pcsrc[0]	memtoreg[1]	memtoreg[0]	regwrite	memread	memwrite	branch
OR	0110011	110	0000000	0	0	0	0	0	0	0	0	1	0	0	0
AND	0110011	111	0000000	0	0	1	0	0	0	0	0	1	0	0	0
XOR	0110011	100	0000000	0	1	0	0	0	0	0	0	1	0	0	0
ADD	0110011	000	0000000	0	1	1	0	0	0	0	0	1	0	0	0
SLT	0110011	010	0000000	1	1	0	0	0	0	0	0	1	0	0	0
SUB	0110011	000	0100000	1	1	1	0	0	0	0	0	1	0	0	0
ORI	0010011	110	-	0	0	0	1	0	0	0	0	1	0	0	0
ANDI	0010011	111	-	0	0	1	1	0	0	0	0	1	0	0	0
XORI	0010011	100	-	0	1	0	1	0	0	0	0	1	0	0	0
ADDI	0010011	000	-	0	1	1	1	0	0	0	0	1	0	0	0
SLTI	0010011	010	-	1	1	0	1	0	0	0	0	1	0	0	0
LW	0000011	010	-	0	1	1	1	0	0	0	1	1	1	0	0
SW	0100011	010	-	0	1	1	1	0	0	x	x	0	0	1	0
BEQ	1100011	000	-	x	x	x	x	0	0	x	x	0	0	0	1
JAL	1101111	-	-	x	x	x	x	0	1	1	0	1	0	0	x
JALR	1100111	000	-	0	1	1	1	1	0	1	0	1	0	0	x

Table 1. Truth Table for the Control Unit

- **Testing**

(Prelab.) 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	func3	func7
0110011_b	110_b	0000000_b
0110011_b	111_b	0000000_b
0110011_b	100_b	0000000_b
0110011_b	000_b	0000000_b
0110011_b	010_b	0000000_b
0110011_b	000_b	0100000_b
0010011_b	110_b	-
0010011_b	111_b	-
0010011_b	100_b	-
0010011_b	000_b	-
0010011_b	010_b	-
0000011_b	010_b	-
0100011_b	010_b	-
1100011_b	000_b	-
1101111_b	-	-
1100111_b	000_b	-

Table 2. Test data for the Control Unit

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

Description

In this experiment, students have to construct a Verilog module for a single cycle implementation of the RISC-V 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 2. 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: **(Adder 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

The Sign Extend unit should be able to handle the immediate extension regardless of the instruction's format. Figure 1 shows the different RISC-V instruction formats. Three instruction formats (i.e. I-type, S-type, and B-type) contain a 12-bit immediate and two instruction formats (i.e. U-type and J-type) contain a 20-bit immediate. For example, in case of the S-type, the Sign Extend unit needs to extract the 12-bit immediate from instruction bits [11:7] and [31:25] then extend it by replicating bit [31] of the instruction 20 times.

In order to extract immediate correctly, the Sign Extend unit needs to identify the instruction format. This can be done through the signals generated in the control unit which was designed in experiment 5. Notice that the I-type format is used by the immediate instructions (e.g. ADDI, ORI), the load instruction, and the jump-and-link-register instruction. The S-type is used by the store instruction, the B-type is used by the branch-if-equal instruction, and the J-type is used by the jump-and-link instruction. So and as an example, I-type input of the Sign Extend unit can be derived as follows:

$$\boldsymbol{I-type} = \ Iformat + LW + JALR$$

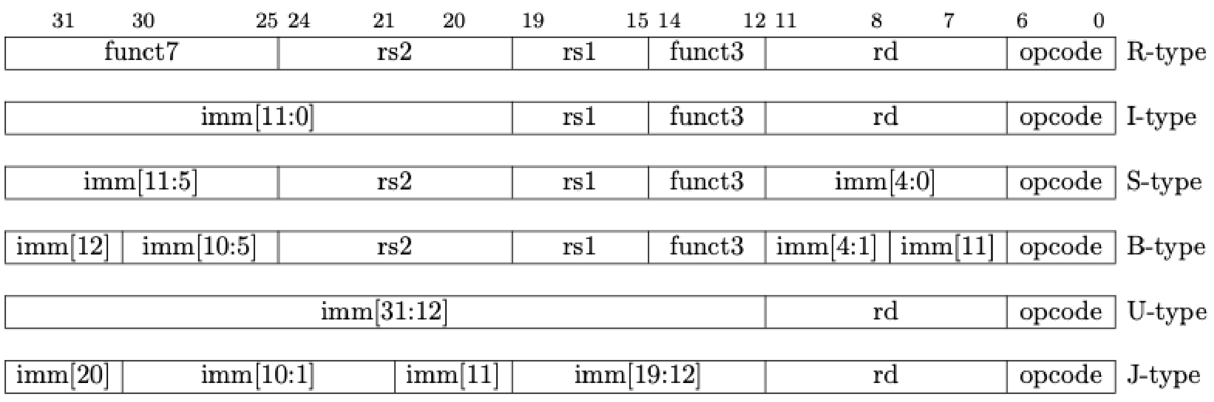

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 1. RISC-V Instruction Formats

Your module should use the following template:

```
module SignExtend (SEout, in, Iformat, LW, SW, BEQ, JAL,
                   JALR);
  input [31:0]in;
  input Iformat, LW, SW, BEQ, JAL, JALR;
  output [31:0]SEout;

  // 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) **32-Bit Shift Left by 1 Unit**

Your module should use the following template.

```
module ShiftLeft32_by1(out, in);
  input  [31:0]in;
  output [31:0]out;

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

5) ***(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
```

6) ***(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
```

7) **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
//ControlUnit(aluop, …, JALR, opcode, func3, func7);
//SignExtend (SEout, in, Iformat, LW, SW, BEQ, JAL, JALR);
//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_by1(out, in);
//Adder32bit (out, a, b); to calculate branch/jal target Address
//Comparator32bit (equal, a, b);
//AND (out, in1, in2);
//Mux_2_to_1_32bit(out, s, i1, i0); branch/jal address or PC + 4
//Mux_3_to_1_32bit(out, s, i2, 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

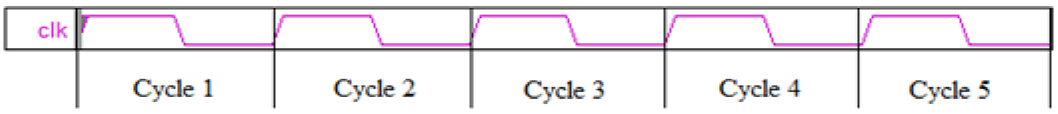
Address	Instruction	Machine Code
00	ORI x2, x0, 5	00506113h
01	LW x5, 4(x0)	
02	SUB x6, x2, x5	
03	ADD x6, x6, x6	
04	JAL x1, 6	
05	SLTI x7, x6, -4	
06	BEQ x7, x0, 4	
07	JALR x0, 0(x1)	
08	SW x6, 4(x0)	
09	LW x8, 4(x0)	
10	XORI x9, x8, -1	
11	AND x10, x9, x8	

- ***(Prelab.)*** Next, write a Verilog **test module** to test your processor module, your test module should run for 13 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

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



Your timing diagram should contain the following signals:

a) Clock, reset, and enable.
b) The output of the program counter (PC).
c) The output of the instruction memory (Instruction).
d) The output of the Sign Extend (SEout).
e) The writedata, readreg1, readreg2, and writereg for the register file.
f) The output for the registers x0, x1, x2, x5, x6, x7, x8, x9, and x10.
g) The input and the output of the ALU (a, b, m, result).
h) The output of the data memory.

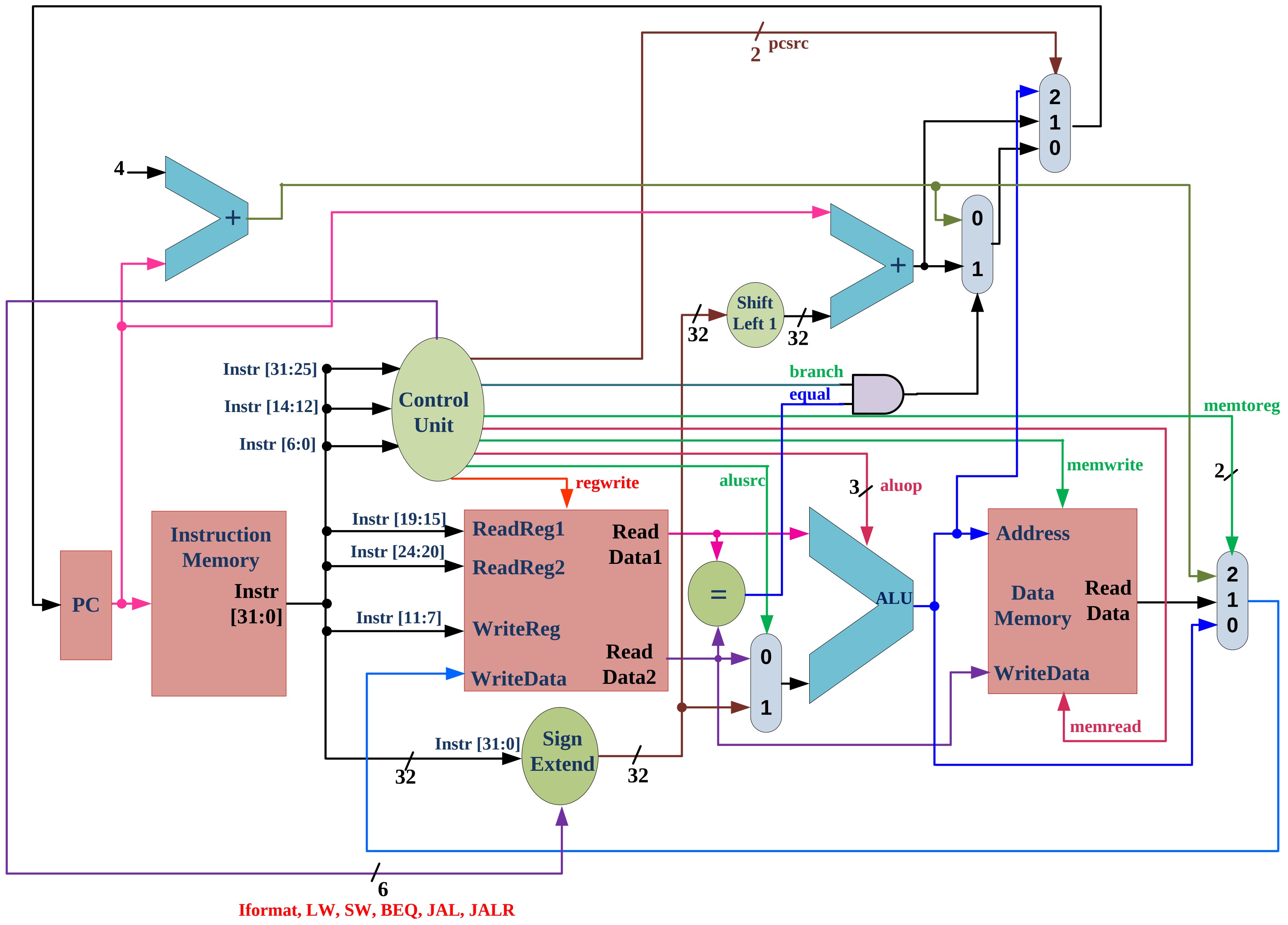


Figure 2. The Datapath of RISC-V Like Processor

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

Description

In this experiment, students have to construct a Verilog module for a 5-stage pipelined implementation of a RISC-V 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 are 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 96-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  [95:0] D;
  output [95:0] 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 153-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  [152:0] D;
  output [152:0] 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 memory and write back 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

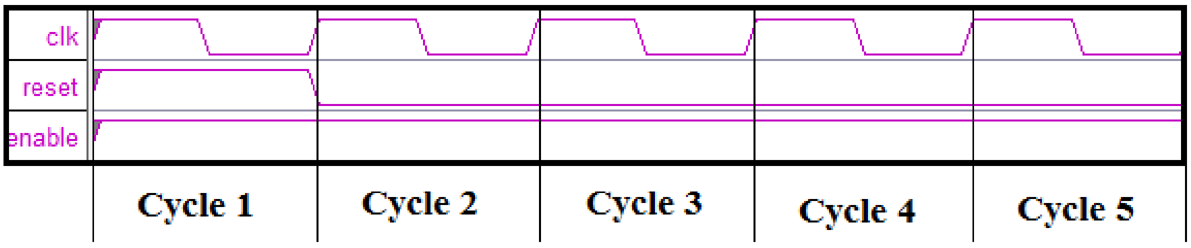
Address	Instruction		Machine Code
00	LW	x1, 4(x0)	00402083_h
01	LW	x2, 12(x0)	
02	LW	x3, 20(x0)	
03	LW	x4, 28(x0)	
04	AND	x5, x1, x2	
05	ORI	x6, x3, 1023	
06	SUB	x7, x4, x2	
07	XOR	x8, x5, x4	
08	ANDI	x9, x6, 2047	
09	SW	x6, 8(x0)	
10	LW	x10, 8(x0)	
11	OR	x11, x7, x8	
12	SLT	x12, x1, x4	

- *(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 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 x1, x5, x6, x7, x8, x9, x10, x11, x12.*
f) *The input and the output of the ALU (a, b, m, result).*
g) *The output of the data memory.*

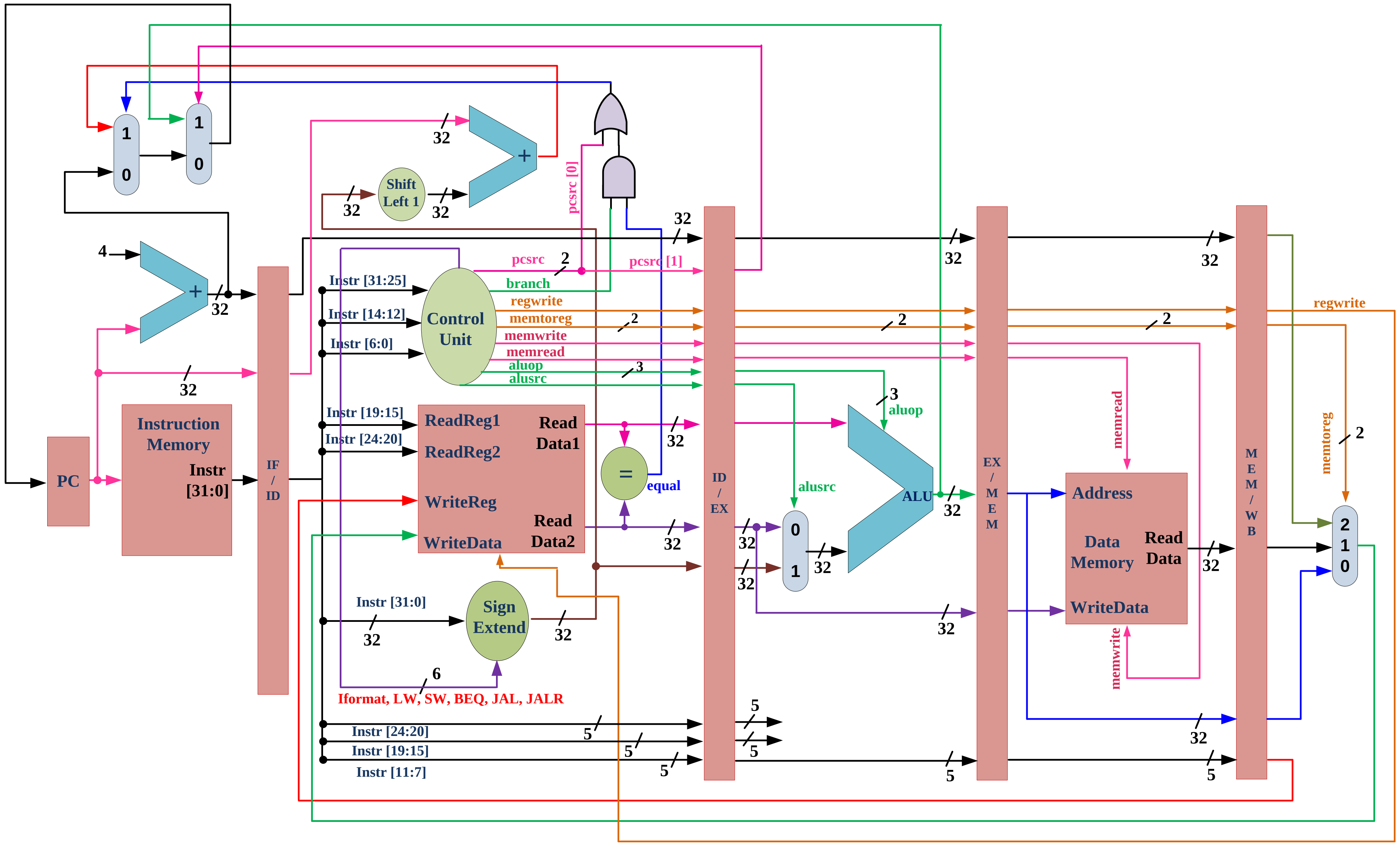


Figure 1: The Datapath of a 5-stage Pipeline RISC-V like Processor

University of Jordan
Computer Engineering Department
CPE439: Computer Organization Lab
Experiment 8: Resolving Data Hazards using Forwarding

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 can 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:

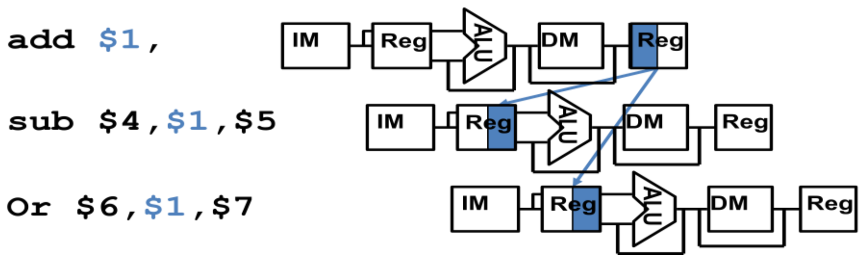


Figure 1. Illustration of data hazards.

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:

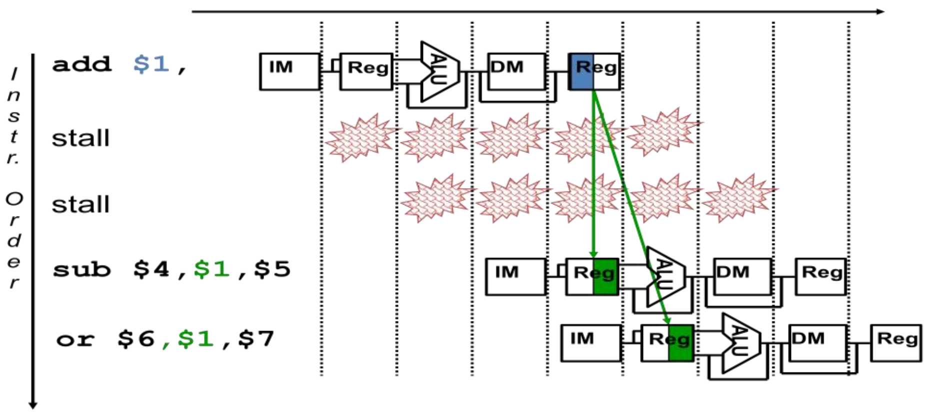


Figure 2. Solving data hazard by stalling the pipeline.

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.

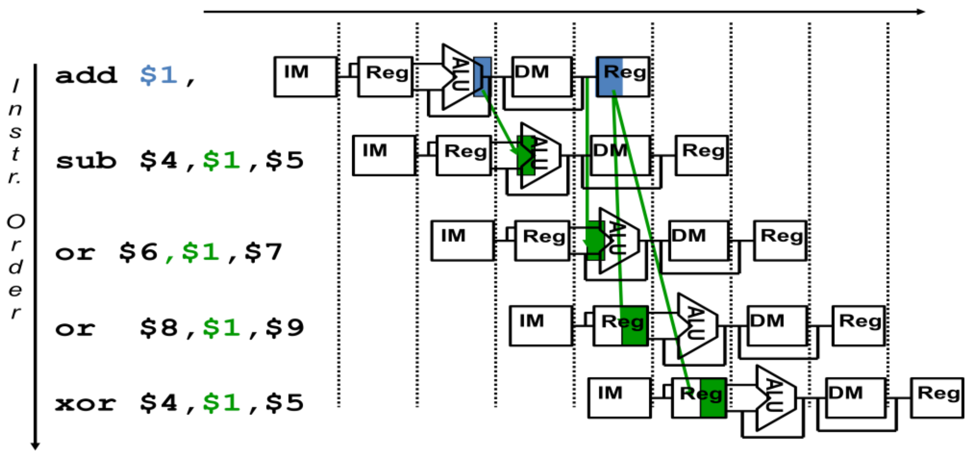


Figure 3. Solving data hazard by forwarding.

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 multiplexers connected to the ALU inputs as shown in Figure 4.

If the source register(s) (rs1 and/or rs2) 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 from the memory stage:***
   a. **if (EX/MEM.RegWrite && (EX/MEM.RegRd == ID/EX.RegRs1) && (EX/MEM.RegRd != 0))**

      ForwardA[0] = 1;

   b. **if (EX/MEM.RegWrite && (EX/MEM.RegRd == ID/EX.RegRs2) && (EX/MEM.RegRd != 0))**

      ForwardB[0] = 1;

2) ***Forwarding from the write-back stage:***
   a. **if (MEM/WB.RegWrite && (MEM/WB.RegRd == ID/EX.RegRs1) && ((EX/MEM.RegRd != ID/EX.RegRs1) || (EX/MEM.RegWrite==0)) && (MEM/WB.RegRd != 0))**

      ForwardA[1] = 1;

   b. **if (MEM/WB.RegWrite and (MEM/WB.RegRd == ID/EX.RegRs2) and ((EX/MEM.RegRd != ID/EX.RegRs2) || (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.

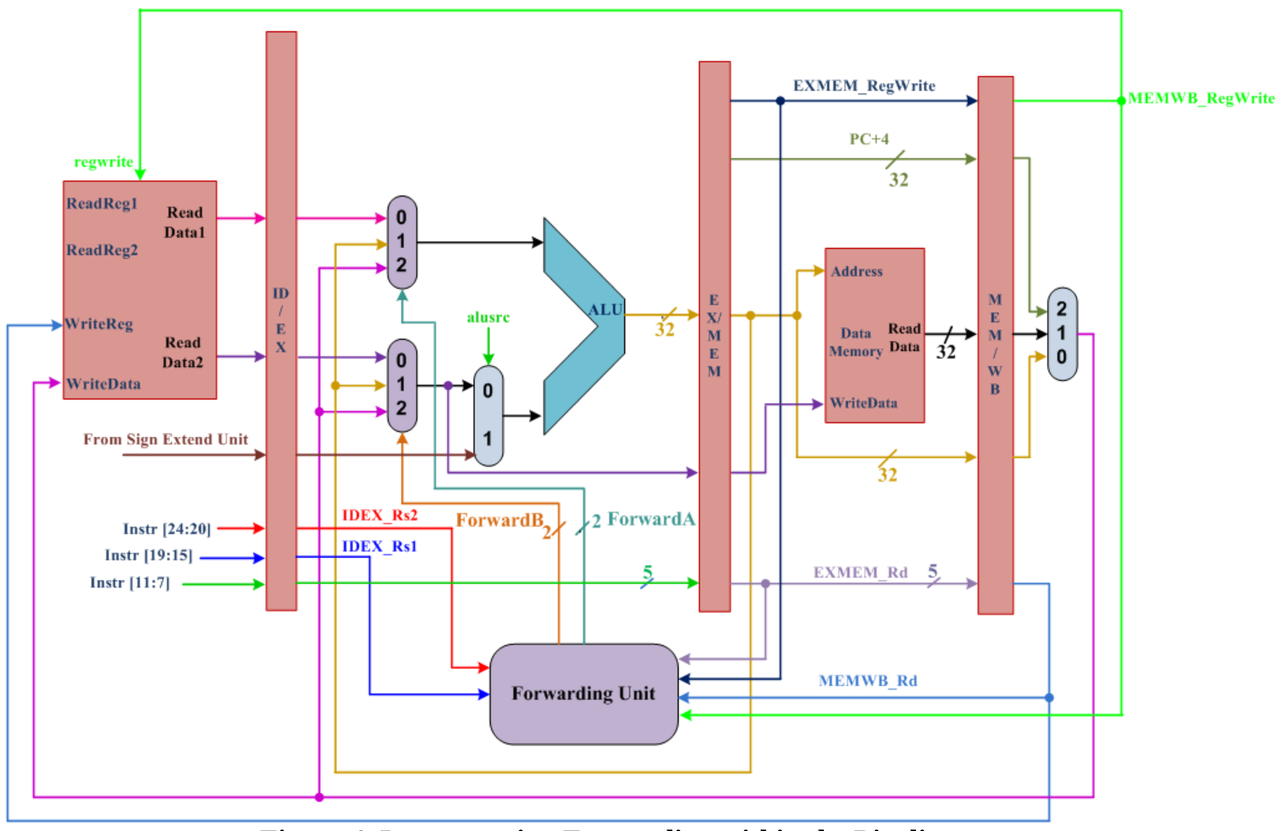


Figure 4. Incorporating Forwarding within the Pipeline.

Procedure

In order to incorporate the forwarding unit 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_Rs1, IDEX_Rs2,
                        EXMEM_RegWrite, MEMWB_RegWrite);

 input  [4:0] EXMEM_Rd, MEMWB_Rd, IDEX_Rs1, IDEX_Rs2;
 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_Rs1	IDEX_Rs2	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      x1, 4 (x0)`	00402083$_h$
01	`LW      x2, 12(x0)`	
02	`LW      x3, 20(x0)`	
03	`LW      x4, 28(x0)`	
04	`ADD     x5, x2, x1`	
05	`AND     x6, x5, x5`	
06	`SLTI    x6, x5, 3`	
07	`OR      x7, x2, x4`	
08	`XOR     x7, x2, x4`	
09	`ADD     x8, x7, x7`	

- ***(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) Instruction (The output of the instruction memory).
    c) The writedata, readreg1, readreg2, and writereg for the register file.
    d) The output for the registers x5, x6, x7, x8.
    e) The input and the output of the ALU (a, b, m, result).
    f) 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 Organization Lab
Experiment 9: Resolving Control Hazards

Description

In the previous experiment, students 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, jal, and jalr. After these instructions are fetched (i.e. stored in the IF/ID register), the processor starts fetching the following instruction (i.e. instruction at PC+4). In case of beq instruction, fetching the instruction at PC+4 might not be correct if the condition evaluates to true (i.e. equal = 1) which is done by the comparator in the decode stage. Instead, the processor should have fetched the instruction pointed-to by the branch target address (i.e. $PC\ of\ branch + SignExtend(immediate) \times 2$) which is computed by the adder in the decode stage.

In case of **unconditional** flow instructions (i.e. jal and jalr), fetching the instruction at PC+4 is always wrong since the processor is supposed to fetch the instruction pointed-to by the target address. Notice the target address of jal (i.e. $PC\ of\ jal + SignExtend(immediate) \times 2$) is computed using the adder in the decode stage (i.e. same as beq instruction). On the other hand, the target address of jalr (i.e. $(rs1) + SignExtend(immediate)$) is computed by the ALU in the execute stage.

In order to resolve these hazards, all instructions that are fetched wrongly have to be removed (flushed) from the pipeline. Since beq and jal instructions are resolved in the decode stage (i.e. branch condition is evaluated and target address is computed), we only need to flush the instruction in the fetch stage. On the other hand, jalr instruction is resolved in the execute stage and the instructions in the fetch and decode stage must be flushed.

The flushing is implemented by resetting the stage register(s) when the control hazard is detected. For beq instruction, the control hazard is detected when the beq is in the decode stage (i.e. branch control signal = 1) and the comparator output (i.e. equal) is 1. For jal instruction, the control hazard is detected when the jal is in the decode stage (i.e. pcsrc[0] = 1). Hence, for beq and jal instructions we only need to reset the IF/ID register in order to flush the instruction in the fetch stage. The required hardware to handle control hazards of beq and jal instructions is shown in Figure 1.

For jalr instruction, the control hazard is detected when the jalr is in the execute stage (i.e. pcsrc[1] in execute stage = 1) and we need to reset the IF/ID and ID/EX registers in order to flush the instructions in the fetch and decode stages, respectively. The required hardware to handle control hazards of jalr instruction is also shown in Figure 1.

Notice that our hardware already selects one of three options when updating the PC value: PC + 4, target address of beq/jal computed in the decode stage, and target address of jal computed in the execute stage.

Procedure

1) The processor module

You need to modify the pipelined processor module by **adding the OR gates required to resolve the control hazards** and **make the needed modifications**.

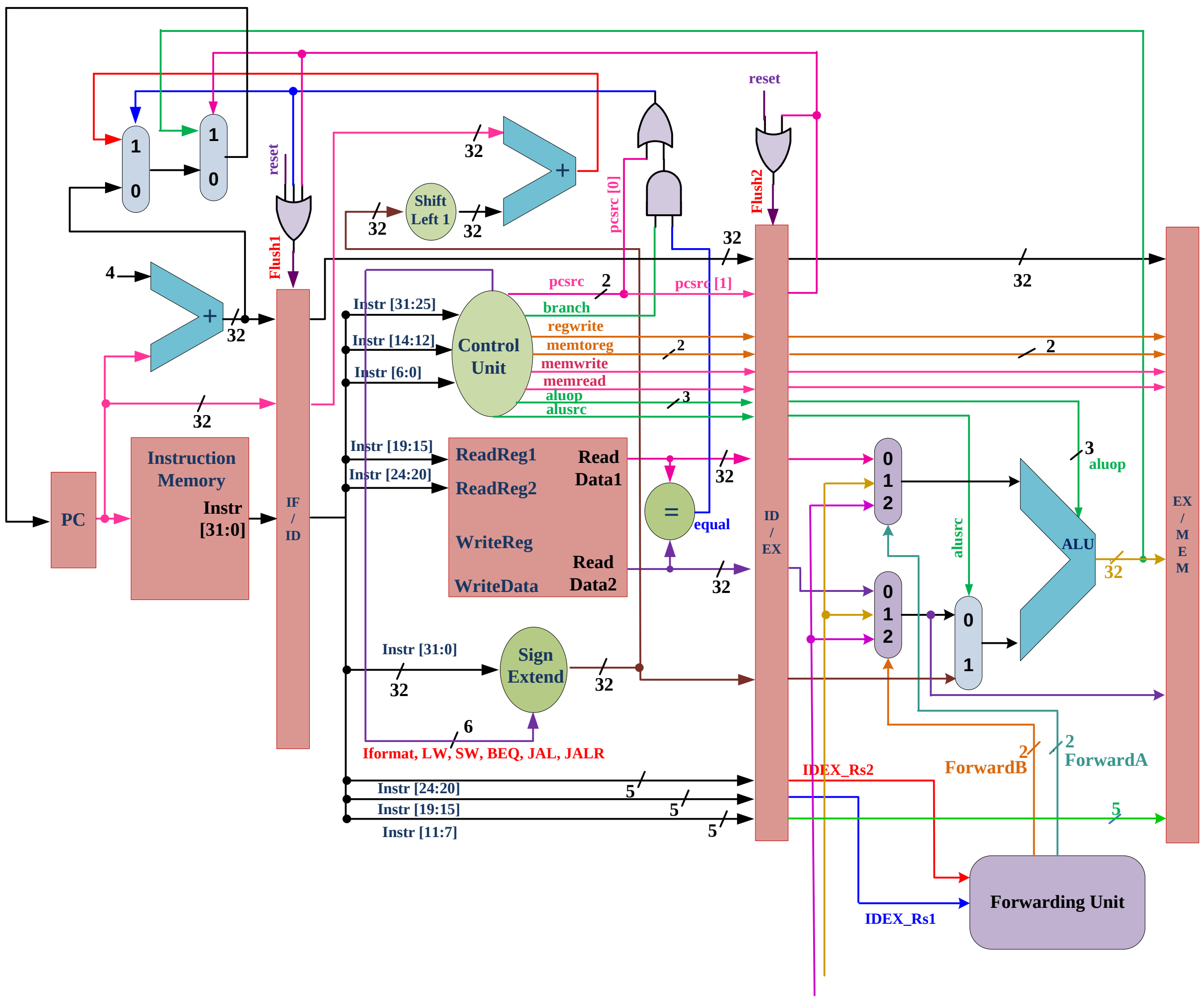


Figure 1: Resolving Control Hazards

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	x1, 4(x0)	00402083_h
01	LW	x2, 12(x0)	
02	LW	x3, 20(x0)	
03	LW	x4, 28(x0)	
04	AND	x5, x1, x3	
05	ORI	x6, x5, 1023	
06	SUB	x8, x4, x2	
07	JAL	x1, 8	
08	XOR	x7, x5, x6	
09	SW	x7, 8(x0)	
10	BEQ	x8, x4, 10	
11	ADDI	x8, x8, 3	
12	SW	x5, 8(x0)	
13	SW	x6, 24(x0)	
14	JALR	x0, 0(x1)	
15	SUB	x9, x8, x3	
16	SLT	x10, x9, x4	

- ***(Prelab.)*** Next, write a Verilog **test module** to test your processor module

  - ***Your timing diagram* should contain the following signals:**
    a) The output of the program counter (PC)
    b) The output of IFID register and IDEX register.
    c) The output for the registers R5, R6, R7, R8, R9, R10.
    d) The output of forwarding unit (ForwardA, ForwardB).
    e) Flush1 and Flush2.
    f) The writedata and memwrite for the data memory.

  - ***Calculate number of cycles needed to execute the above code.***

CPE 335
Computer Organization

Basic MIPS Pipelining – Part II

Dr. Iyad Jafar

Adapted from Dr. Gheith Abandah slides

http://www.abandah.com/gheith/Courses/CPE335_S08/index.html

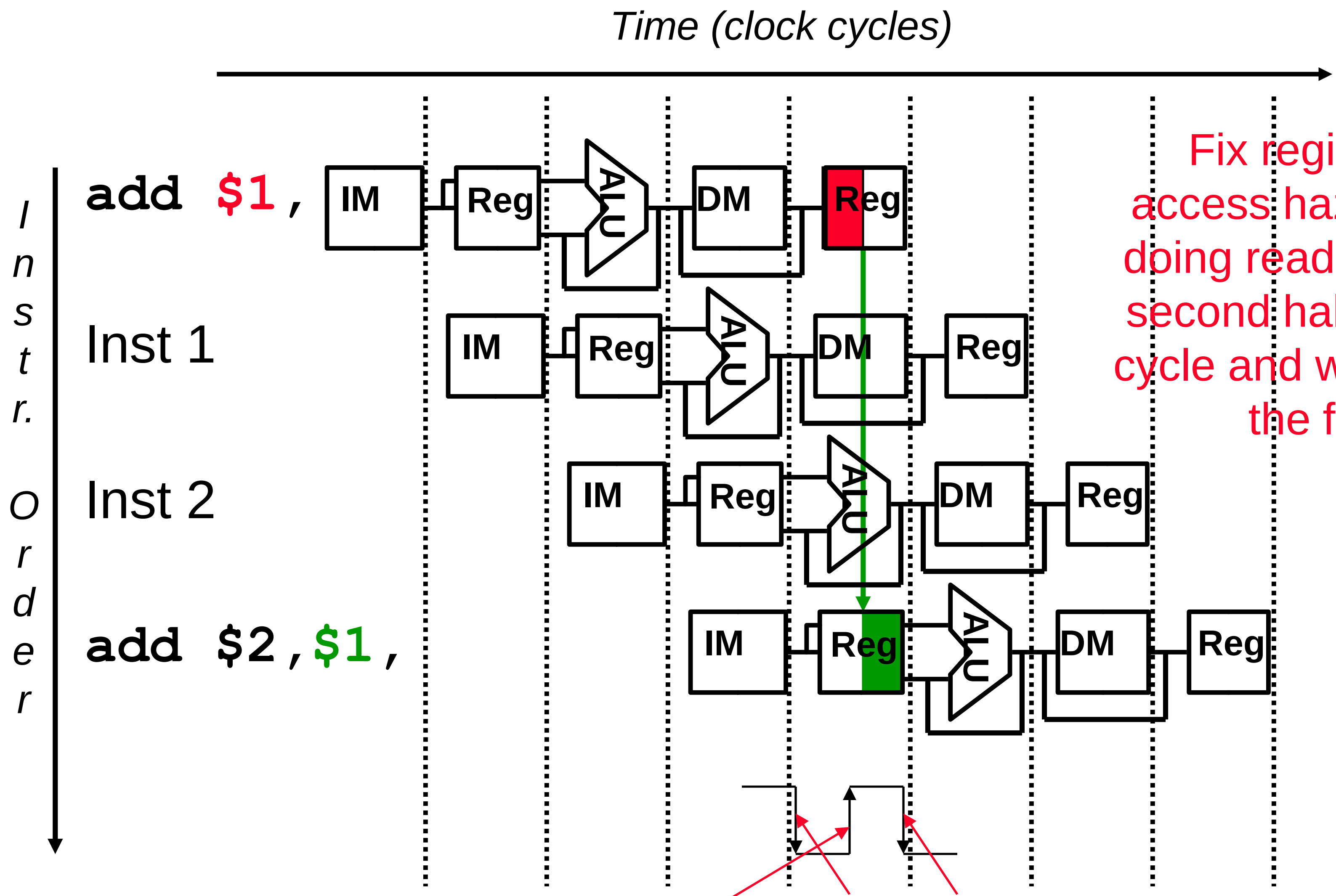
Pipelining the MIPS ISA

- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations can occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)
- What makes it hard
  - structural hazards: what if we had only one memory?
  - control hazards: what about branches?
  - data hazards: what if an instruction's input operands depend on the output of a previous instruction?

Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction’s source operand(s) are produced by a prior instruction still in the pipeline
    - Note that data hazards can come from R-type instructions or lw instructions
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch instructions
- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Structural Hazard – Register File



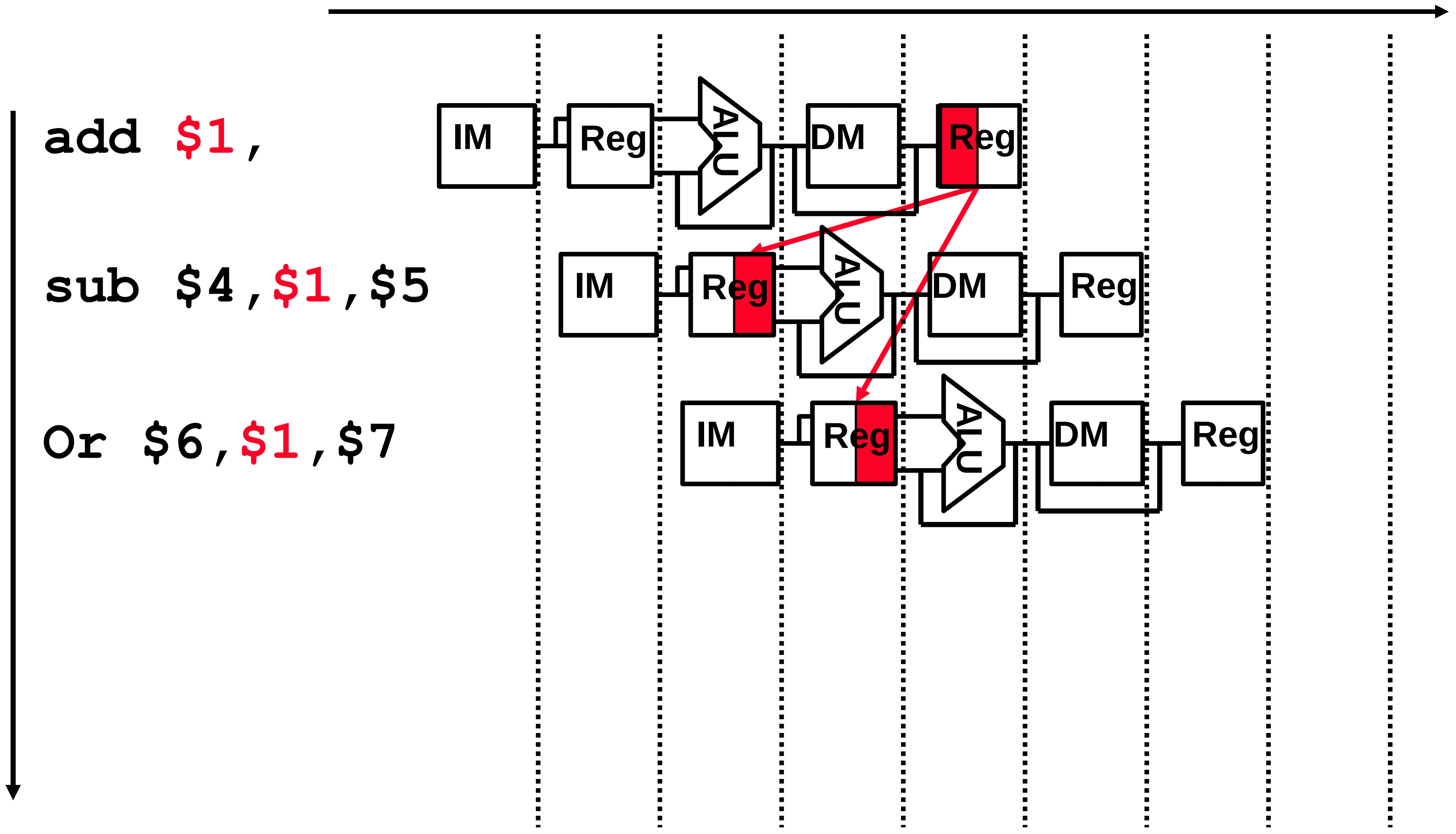
Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

clock edge that controls register writing

clock edge that controls loading of pipeline state registers

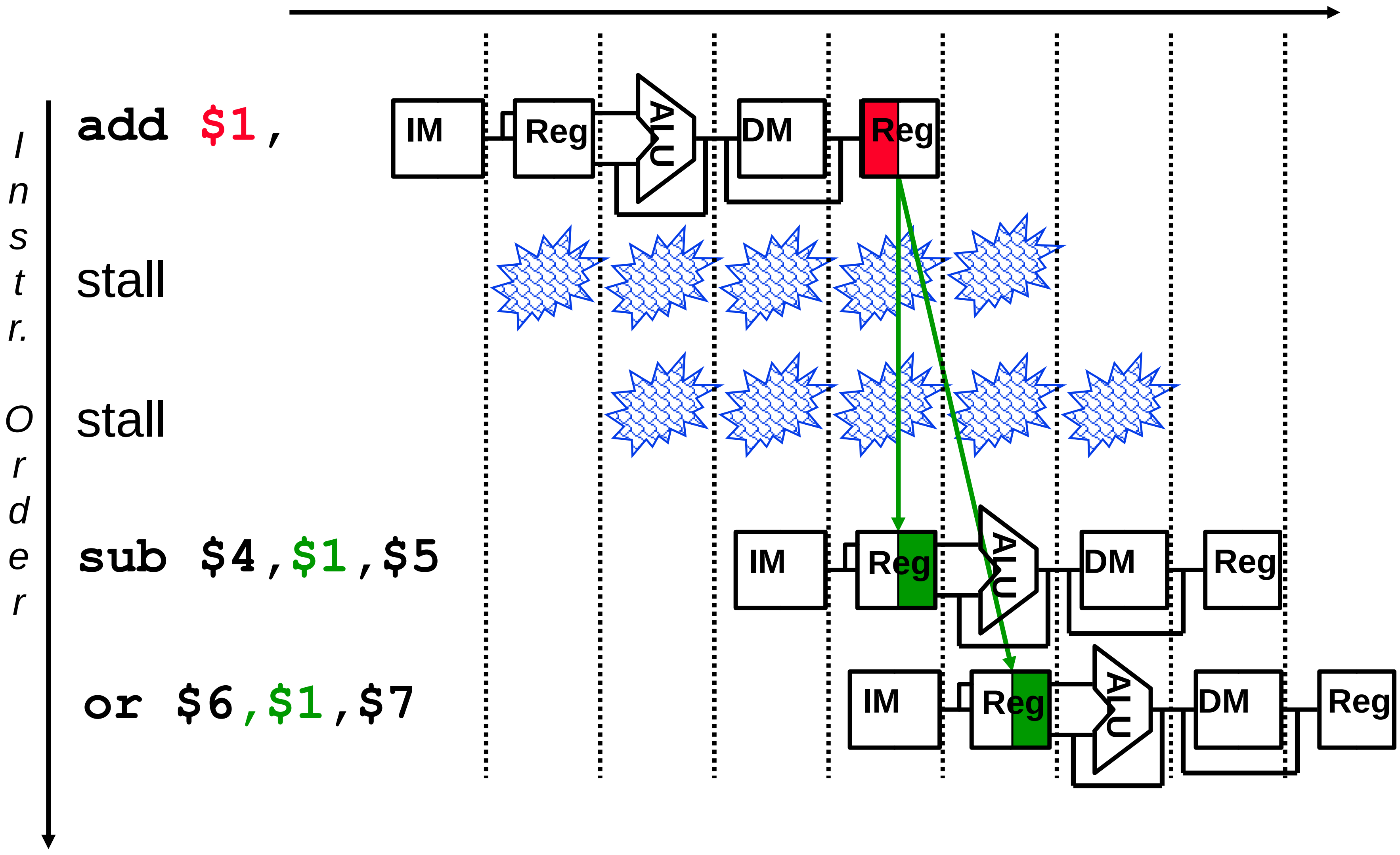
Data Hazards with Register Usage

- Dependencies backward in time cause hazards



- Register-use data hazard

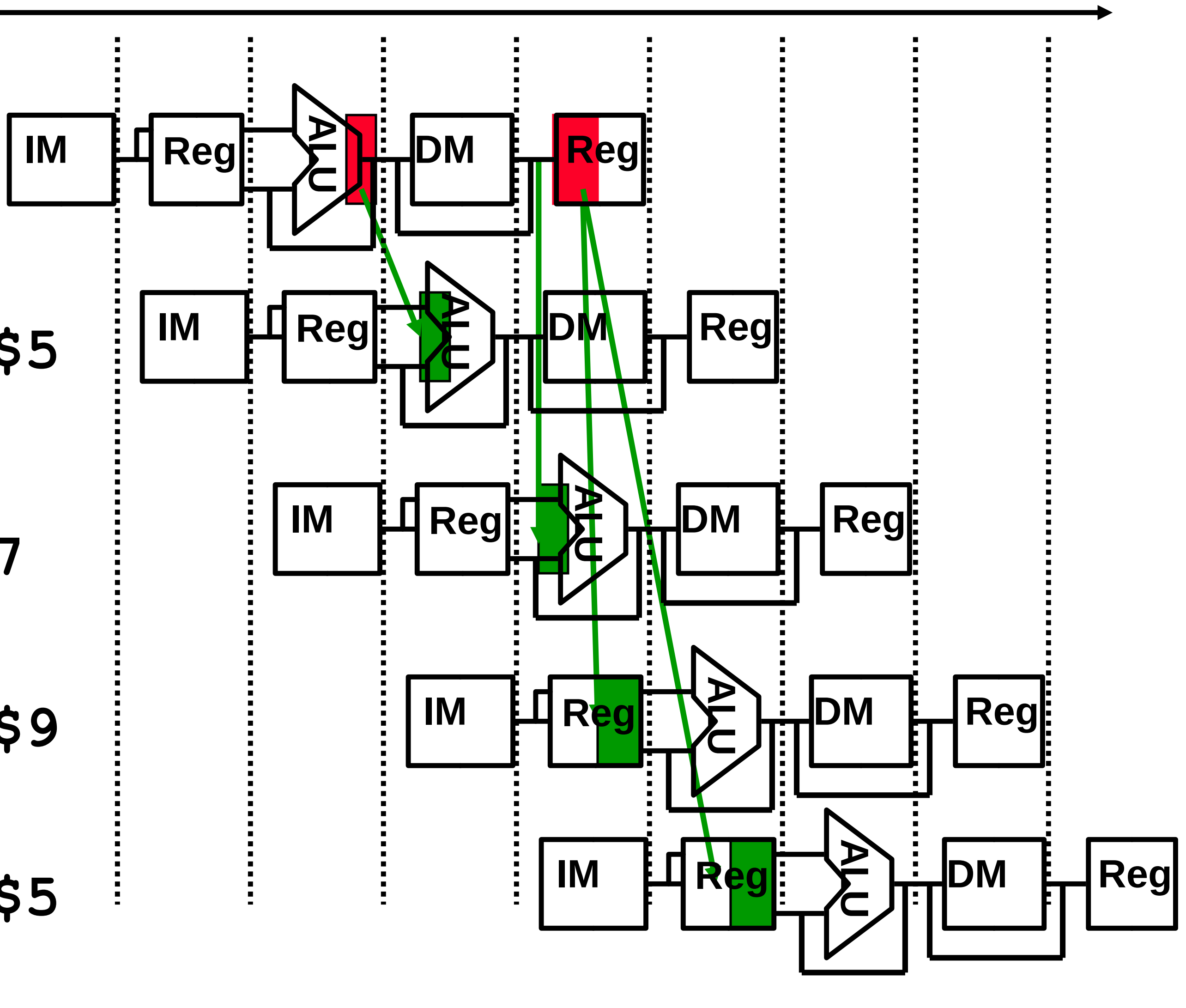
Fixing Register-use Data Hazards with Stalls



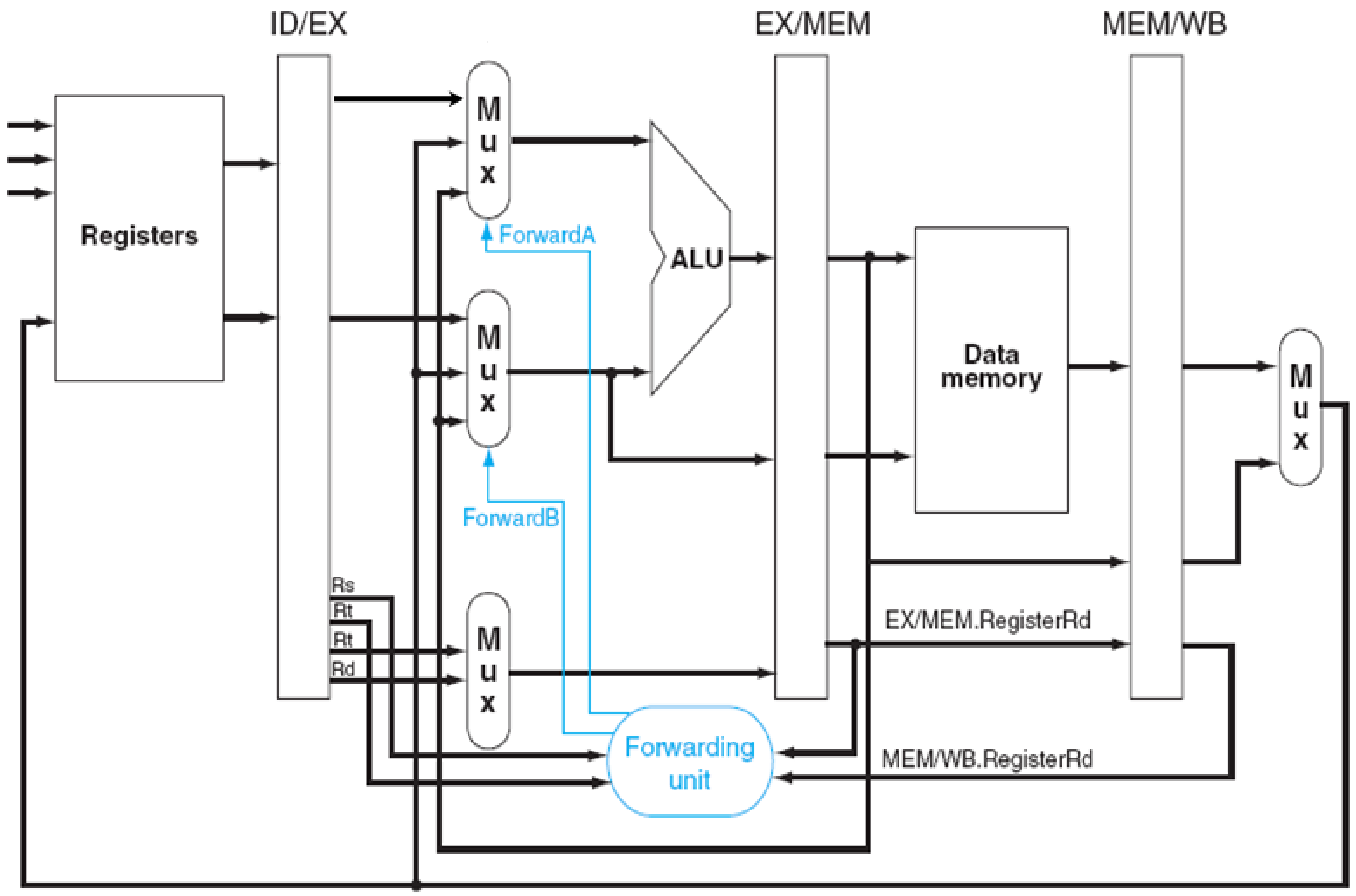
Fixing Data Hazards with Forwarding

Instr. Order

```
add $1,
sub $4,$1,$5
or $6,$1,$7
or  $8,$1,$9
xor $4,$1,$5
```



Forwarding Implementation



Pipelining Hazard – Example 1

- Consider the following code segment in C

  A = B + E

  C = B + F

  (1) Generate the MIPS code assuming that variables A,B, C, E, and F are in memory and addressable with offsets 0, 4, 8, 12, and 16 from $t0

  (2) Find all the data hazards and determine the number of cycles required to run the code .

  (3) Can you reorder the code to reduce the stalls ?

  * Assume the forwarding discussed so far is implemented

Pipelining Hazard – Example 1

# MIPS Code

```
lw     $t1, 4($t0)       # loads B
lw     $t2, 12($t0)      # loads E
add    $t3, $t1, $t2     # A = B + E
sw     $t3, 0($t0)       # stores A
lw     $t4, 16($t0)      # loads F
add    $t5, $t1, $t4     # C = B + F
sw     $t5, 8($t0)       # stores C
```

# we have two load-use hazards ➔ add two cycles

Pipelining Hazard – Example 1

- Ideally, we need 11 (5 + 6) cycles to execute the code
- with the presence of 2 stalls, we need 13 cycles
- Reordered code

```
lw     $t1, 4($t0)     # loads B
lw     $t2, 12($t0)    # loads E
lw     $t4, 16($t0)    # loads F
add    $t3, $t1, $t2   # A = B + E
sw     $t3, 0($t0)     # stores A
add    $t5, $t1, $t4   # C = B + F
sw     $t5, 8($t0)     # stores C
```

- Now we don’t have load-use hazards; we need 11 cycles

Pipelining Hazard – Example 2

- Assume that the pipelined MIPS processor without forwarding is used to run a program with the following instruction mix: 20% loads, 20% store, and 60% ALU. Then compute the average CPI given that
  - 10% of the ALU instructions result in load-use hazards.
  - 15% of the ALU instructions result in read-before-write hazards.
- Ideally, the average CPI is 1 for each instruction
- With no forwarding
  - Load-use hazards add two cycles
  - Register-use hazards add two cycles
- Average CPI = 0.2 x 1 + 0.2 x 1 + 0.75 x 0.60 x 1 +

  0.1 x 0.60 x 3 + 0.15 x 0.60 x 3 = 1.30