Computer Architecture (CS F342)

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Upendra Singh, BITS Pilani

Computer Architecture Lab

Instructor: Upendra Singh

p20170428@pilani.bits-Pilani.ac.in

TA1: ALARK FAJALIA

f20180296@pilani.bits-pilani.ac.in

TA2: AVADH RAJESH HARKISHANKA

f20180322@pilani.bits-pilani.ac.in

TA3: JVN Saketh Ram

h20200242@pilani.bits-pilani.ac.in

Section: P3 and P6

Day & Time: Tuesday, 10,11 hrs (P3)
Tuesday, 7,8 hrs (P6)

Reference Books: "**Verilog HDL: A guide to Digital Design and Synthesis**" by Samir Palnitkar.

Course Objectives

- Getting started with HDL program using Icarus Simulator
- Understand basic Verilog language primitives (e.g. module, data types, identifiers, vectors, registers, keywords etc.)
- To understand the various types of modelling

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Lab – 1

- Getting started with HDL program using Icarus Simulator.
- Understand basic Verilog language primitives.
- To understand the various types of modelling.

What is Verilog?

- Hardware Description Languages (HDL).
- VHDL was made an IEEE Standard in 1987, and Verilog in 1995.
- Verilog is very similar to C-language.
- Describe designs at a
  - High level of abstraction such as at the architectural or behavioral level
  - Lower level of abstraction(i. e., gate and switch levels).
- A primary use of HDLs is the simulation of designs before the designer must commit to fabrication.

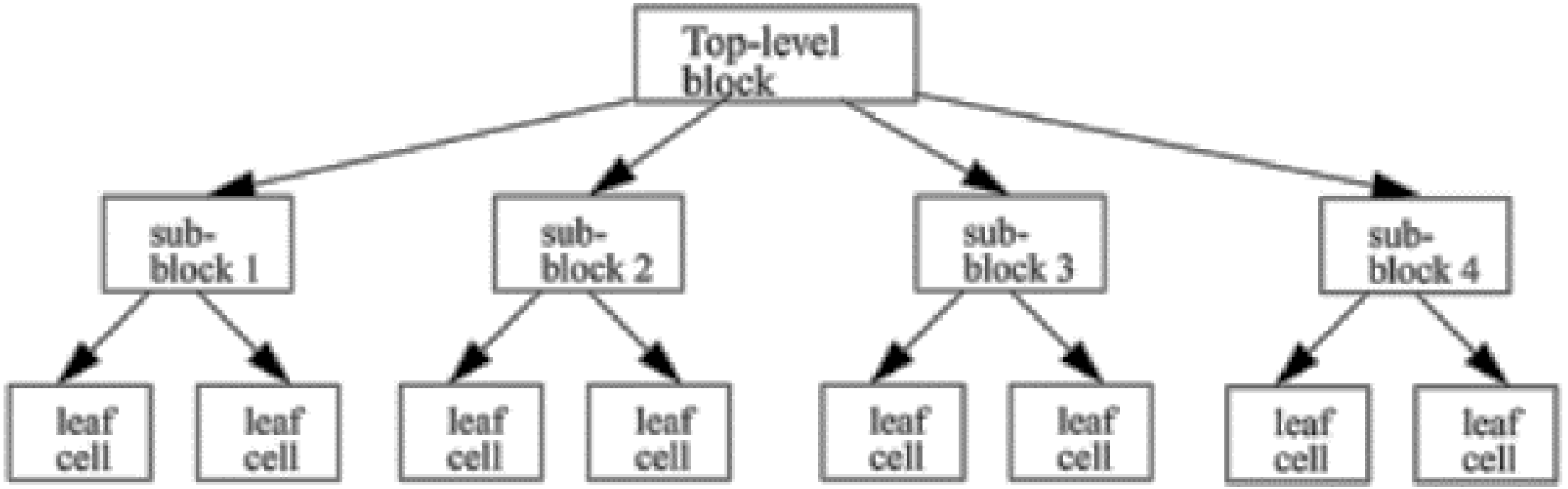
Verilog Structure

- Verilog differs from regular programming languages:
  - Simulation time concept,
  - Multiple threads, and
  - Some basic circuit concepts like network connections and primitive gates.

Design Methodologies

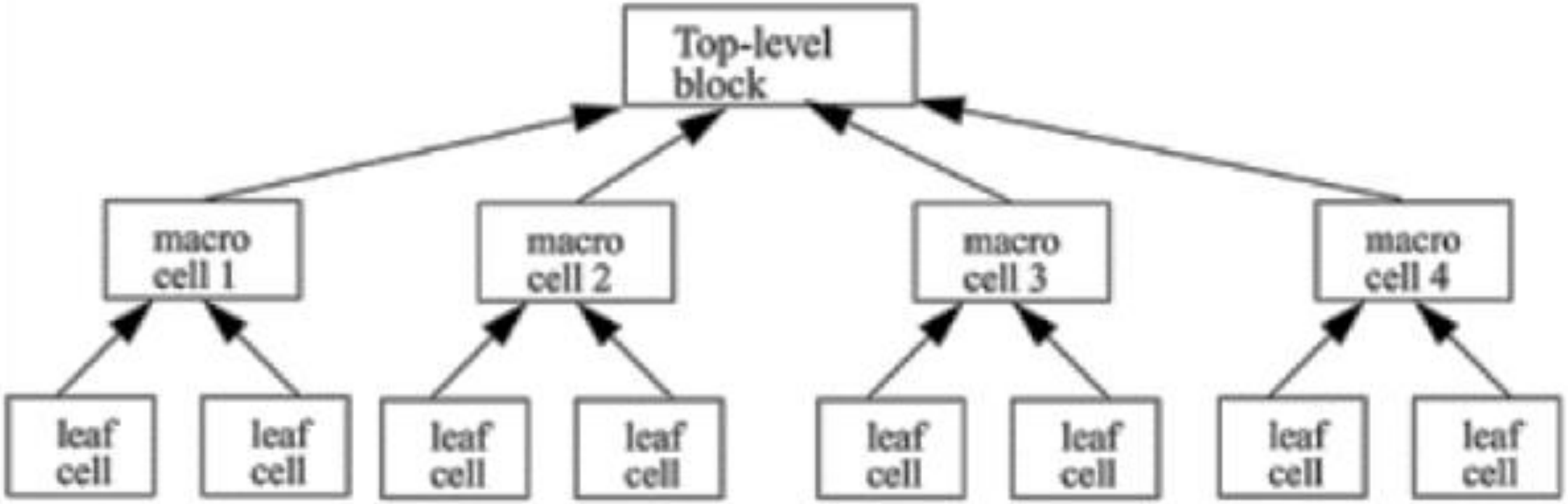
- There are two basic types of digital design methodologies.
  - A top-down design methodology.
  - A bottom-up design methodology.

A Top-down Design Methodology



In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design.

A Bottom-up Design Methodology

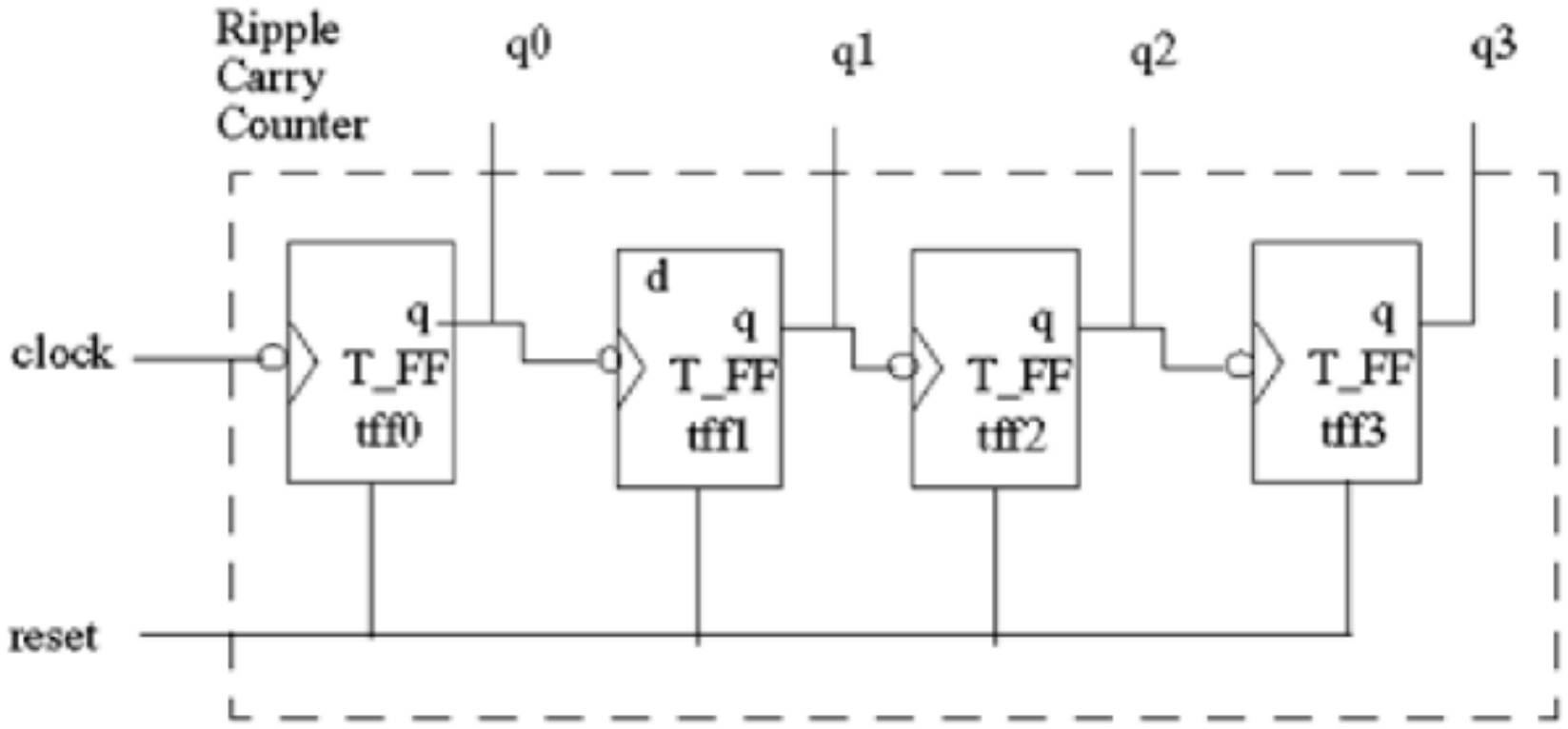


Design Methodologies

- Typically, a combination of top-down and bottom-up flows is used.
- Design architects define the specifications of the top-level block.
- Logic designers decide how the design should be structured by breaking up the functionality into blocks and sub-blocks.
- At the same time, circuit designers are designing optimized circuits for leaf-level cells.
- They build higher-level cells by using these leaf cells.

Design Methodologies : Example

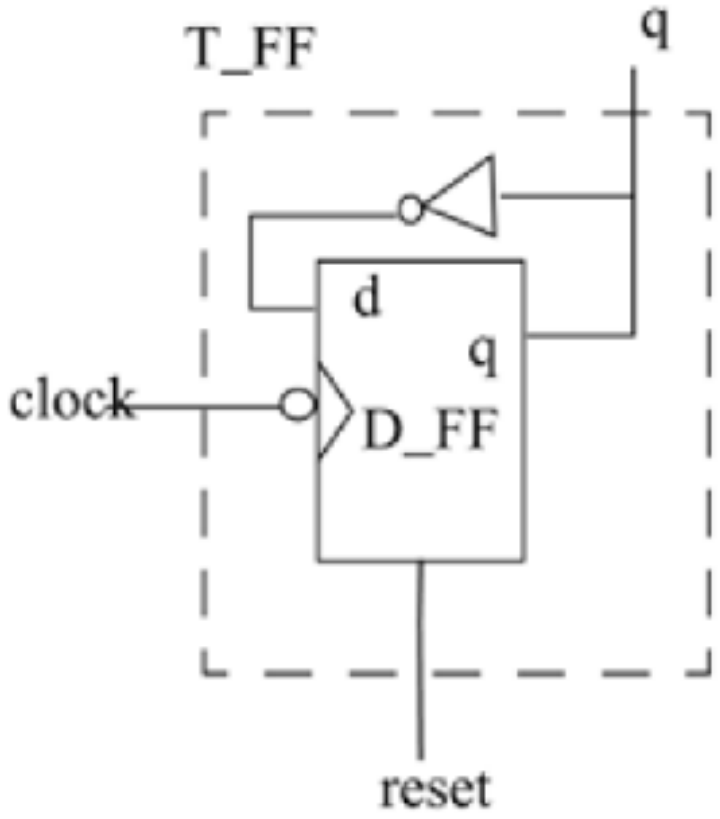
4-bit Ripple Carry Counter



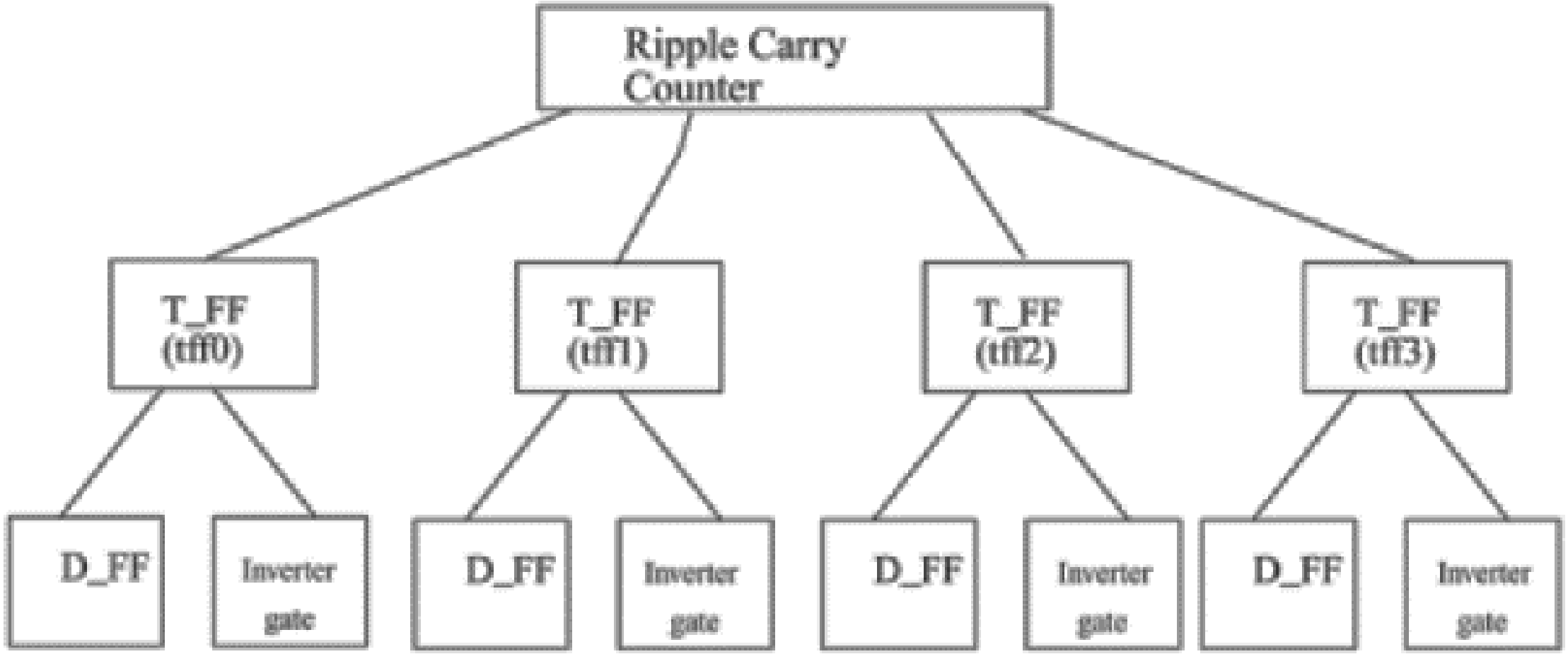
Design Methodologies : Example

Negative edge-triggered toggle flipflops.

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



Design Methodologies : Example



Module

- Circuit components are designed inside a module.
- Can contain both structural and behavioral statements.
- Structural statements represent circuit components like logic gates, counters, and microprocessors.
- Behavioral level statements are programming statements that have no direct mapping to circuit components.
- Loops, if- then statements, and stimulus vectors which are used to exercise a circuit.

Module: Syntax

```
module <module_name> (<module_terminal_list>);
...
<module internals>
...
...
endmodule
```

*Chapter 4 of the book “Verilog HDL” by Samir Palnitkar.

Module

- One module definition cannot contain another module definition within the module and endmodule statements.
- Instead, a module definition can incorporate copies of other modules by instantiating them.
- Module nesting is illegal in Verilog.

Module: Example

```
`timescale 1ns / 1ps
//create a NAND gate out of an AND and an Invertor
module some_logic_component (c, a, b);
        // declare port signals
  output c;
  input a, b;
      // declare internal wire
  wire d;
      //instantiate structural logic gates
  and a1(d, a, b); //d is output, a and b are inputs
  not n1(c, d);     //c is output, d is input
endmodule
```

Levels of Abstraction in Verilog Programming

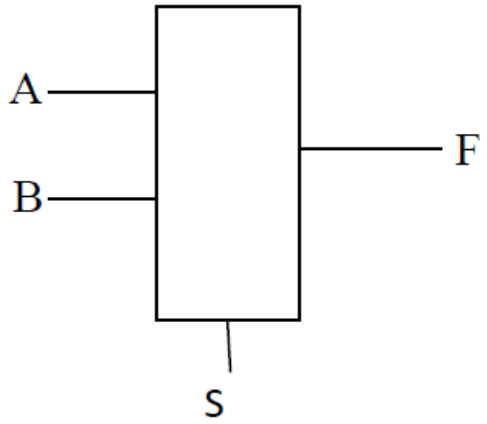
- Gate Level Modeling
- Data Flow Modeling
- Behavioral Modeling
- RTL Modeling

Gate Level Modeling

- Basic level of modeling in terms of logic gates and the connections between these gates.
- The circuit is described in terms of gates say AND, OR etc.
- For a user who is familiar with the basic knowledge of Digital logic Design.
- Correspondence between the Verilog Description and the Circuit Diagram.

*Chapter 2 of the book "Verilog HDL" by Samir Palnitkar.

Gate Level Modeling



2:1 Multiplexer

```
module mux2to1_gate (a,b,s,f);
      input a,b,s;
      output f;
      wire c,d,e;

      not n1(e,s);// e=~s
      and a1(c,a,s);
      and a2(d,b,e);
      or o1(f,c,d);
endmodule
```

F= OR (AND (S, A), AND ((NOT (S)), B))

Data Flow Modeling

- How the data flows between the hardware registers and how the data is processed.
- For small circuits, the gate level modeling works well as the number of gates is limited.
- In complex designs the designers may have to concentrate more on implementing the function than bother about the gates.

*Chapter 6 of the book "Verilog HDL" by Samir Palnitkar.

Data Flow Modeling

- 2:1 Multiplexer can be written as

```
F= (S&A)|(~S&B);
Or
F = S ? A : B;
```

```
module mux2to1_df (a,b,s,f);
        input a,b,s;
        output f;
        assign f = s ? a : b;
endmodule
```

Behavioral Modeling

- Highest level of abstraction provided by Verilog.
- At this level is like an algorithm.
- Module can be implemented in terms of the desired design algorithm without looking into the hardware details.

*Chapter 7 of the book "Verilog HDL" by Samir Palnitkar.

Behavioral Modeling

- 2:1 Multiplexer can be written as

```
If(s= = 1)
        F=A;
  Else
        F=B
```

```
module mux2to1_beh(a,b,s,f);
      input a,b,s;
      output f;
      reg f;
      always@(s or a or b)
            if(s==1) f = a;
            else f = b;
endmodule
```

RTL Modeling

- Register Transfer Level Modeling refers to the Verilog description that uses a combination of both Behavioral and Data Flow constructs that is synthesizable.

Instances

- A module provides a template.
- Verilog creates a unique object from the template.
- Each object has its own
  - Name,
  - Variables,
  - Parameters, and
  - I/O interface.
- The process of creating objects from a module template is called instantiation, and the objects are called instances.

Instances

```
module T_FF(q, clk, reset);
  output q;
  input clk, reset;
  wire d;
  D_FF dff0(q, d, clk, reset);
  // Instantiate D_FF. Call it dff0.
  not n1(d, q);
endmodule
```

Simulation: Testbench or Stimulus

```
module testbench;
      reg a,b,s;
      wire f;
      mux2to1_gate mux_gate (a,b,s,f);
      initial
            begin
                  $monitor(,$time," a=%b, b=%b, s=%b f=%b",a,b,s,f);
                  #0 a=1'b0;b=1'b1;
                  #2 s=1'b1;
                  #5 s=1'b0;
                  #10 a=1'b1;b=1'b0;
                  #15 s=1'b1;
                  #20 s=1'b0;
                  #100 $finish;
            end
endmodule
```

*Chapter 2 of the book "Verilog HDL" by Samir Palnitkar.

Thank You

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Lab – 2

- Combinational digital circuit modeling.
- Constructing a larger module using a smaller module.

Whitespace

- Blank spaces (\b) , tabs (\t) and newlines (\n) comprise the whitespace.
- Whitespace is ignored by Verilog except when it separates tokens.
- Whitespace is not ignored in strings.

Comments

```
a = b && c; // This is a one-line comment

/* This is a multiple line
comment */

/* This is /* an illegal */ comment */

/* This is //a legal comment */
```

Number Specification

- There are two types of number specification in Verilog:
  - sized and unsized.
- **Sized numbers**

<size> '<base format> <number>

```
4'b1111 // This is a 4-bit binary number
12'habc // This is a 12-bit hexadecimal number
16'd255 // This is a 16-bit decimal number
```

Number Specification

- **Unsized numbers**

```
23456 // This is a 32-bit decimal number by default
'hc3 // This is a 32-bit hexadecimal number
'o21 // This is a 32-bit octal number
```

Number Specification

- **X or Z values**
  - Verilog has two symbols for unknown and high impedance values.

```
12'h13x // This is a 12-bit hex number; 4 least
significant bits unknown
6'hx // This is a 6-bit hex number
32'bz // This is a 32-bit high impedance number
```

Number Specification

- **Negative numbers**
  - Negative numbers can be specified by putting a minus sign before the size for a constant number.

```
-6'd3 // 8-bit negative number stored as 2's complement of 3
-6'sd3 // Used for performing signed integer math
4'd-2 // Illegal specification
```

Number Specification

- **Underscore characters and question marks**
- An underscore character "_" is
  - allowed anywhere in a number except the first character.
  - improve readability of numbers
  - ignored by Verilog.
- A question mark "?" is for z in the context of numbers.
  - The ? is used to enhance readability in the casex and casez statements

```
12'b1111_0000_1010 // Use of underline characters
for readability
4'b10?? // Equivalent of a 4'b10zz
```

Strings

- A string is a sequence of characters that are enclosed by double quotes.
  - must be contained on a single line.
  - treated as a sequence of one-byte ASCII values

```
"Hello Verilog World" // is a string
"a / b" // is a string
```

Identifiers and Keywords

- Keywords are special identifiers reserved to define the language constructs.
  - Keywords are in lowercase.
- Identifiers are names given to objects.
  - alphanumeric characters, underscore, dollar sign
  - case sensitive.
  - cannot start with a digit or a $ sign

```
reg value; // reg is a keyword; value is an identifier
input clk; // input is a keyword, clk is an identifier
```

Escaped Identifiers

- All characters between backslash and whitespace are processed literally.
- Any printable ASCII character can be included in escaped identifiers.

```
\a+b-c

\**my_name**
```

Value Set

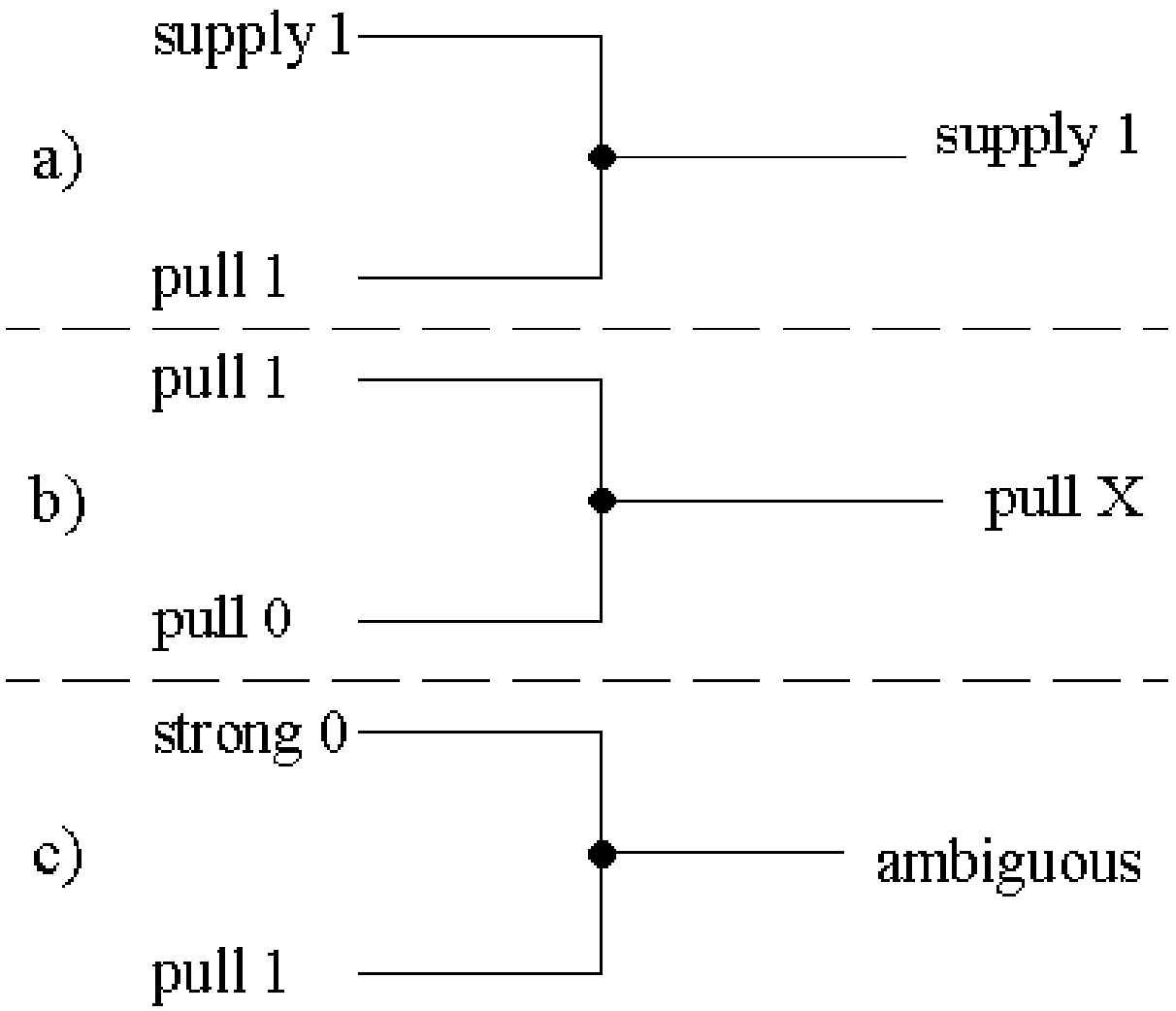
Verilog supports four values and eight strengths to model the functionality of real hardware.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Signal Strengths

- The ability to model varying signal levels as produced by digital hardware is fundamentally important for the simulation of switch level circuits. This is accomplished by assigning various signal strengths

Resolving Signal Contention



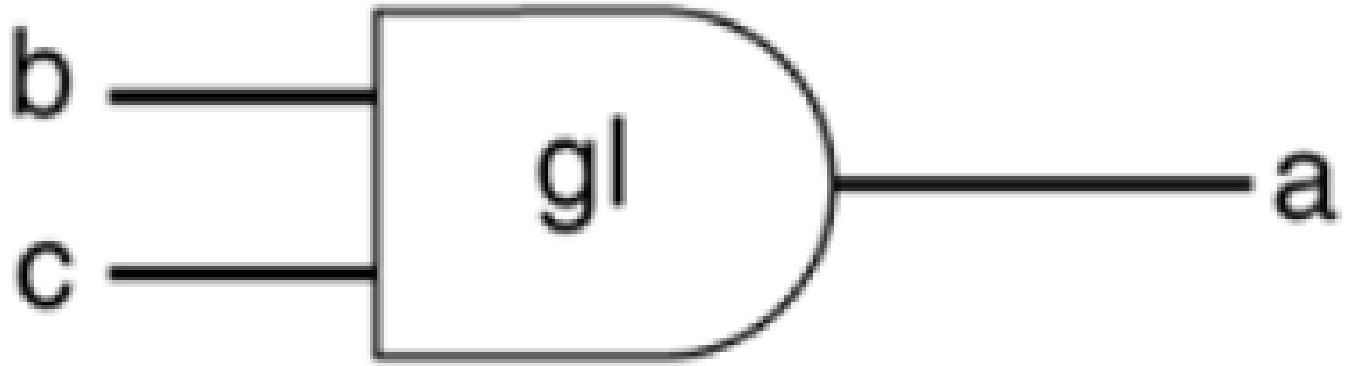
Nets

- Represent connections between hardware elements.
- Declared primarily with the keyword wire.
- Terms wire and net are often used interchangeably.
- Nets get the output value of their drivers.
  - If a net has no driver, it gets the value z.

```
wire a; // Declare net
wire b,c; // Declare two wires b,c
wire d = 1'b0; // Net d is fixed to logic value 0 at
                  declaration.
```

Nets

- In Figure net a is connected to the output of and gate g1. Net a will continuously assume the value computed at the output of gate g1, which is b & c.



Registers

- Represent data storage elements.
- Retain value until another value is placed onto them.
- Unlike a net, a register does not need a driver.
  - Values of registers can be changed by assigning a new value to the register.

```
reg reset; // declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
  reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
  #100 reset = 1'b0; // after 100 time units reset is deasserted.
end
```

Registers

- **Signed Register Declaration**

```
reg signed [63:0] m; // 64 bit signed value
integer i; // 32 bit signed value
```

Vectors

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of
32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; // Vector register,
virtual address 41 bitswide
```

Vectors

- **Vector Part Select**

```
busA[7] // bit # 7 of vector busA

bus[2:0] // Three least significant bits of
vector bus,

// using bus[0:2] is illegal because the
significant bit should always be on the left of
a range specification

virtual_addr[0:1] // Two most significant bits
of vector virtual_addr
```

Vectors

- **Variable Vector Part Select**
- [<starting_bit>+:width] - part-select increments from starting bit
- [<starting_bit>-:width] - part-select decrements from starting bit

Vectors

```
reg [255:0] data1; //Little endian notation
reg [0:255] data2; //Big endian notation
reg [7:0] byte;

//Using a variable part select, one can choose parts
byte = data1[31-:8]; //starting bit = 31, width =8 => data[31:24]
byte = data1[24+:8]; //starting bit = 24, width =8 => data[31:24]
byte = data2[31-:8]; //starting bit = 31, width =8 => data[24:31]
byte = data2[24+:8]; //starting bit = 24, width =8 => data[24:31]

//The starting bit can also be a variable. The width has
//to be constant. Therefore, one can use the variable part select
//in a loop to select all bytes of the vector.
for (j=0; j<=31; j=j+1)
    byte = data1[(j*8)+:8]; //Sequence is [7:0], [15:8]... [255:248]

//Can initialize a part of the vector
data1[(byteNum*8)+:8] = 8'b0; //If byteNum = 1, clear 8 bits [15:8]
```

Integer , Real, and Time Register Data Types

- **Integer**
  - General purpose register data type.
  - Default width for an integer is the host-machine word size.
    - At least 32 bits
    - Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities.

```
integer counter; // general purpose variable used as a counter.
initial
    counter = -1; // A negative one is stored in the counter
```

Integer , Real, and Time Register Data Types

- **Real**
  - Real number constants and real register data types.
  - can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6).
  - cannot have a range declaration, and their default value is 0.

Integer , Real, and Time Register Data Types

- **Real**

```
real delta; // Define a real variable called delta
initial
begin
    delta = 4e10; // delta is assigned in scientific notation
    delta = 2.13; // delta is assigned a value 2.13
end
integer i; // Define an integer i
initial
    i = delta; // i gets the value 2 (rounded value of 2.13)
```

Integer , Real, and Time Register Data Types

- **Time**
  - Special time register data type is used to store simulation time.
  - Width of time register is implementation-specific.
    - but is at least 64 bits.
    - The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time
initial
    save_sim_time = $time; // Save the current simulation time
```

Arrays: Declaration

```
integer count[0:7]; // An array of 8 count variables

reg bool[31:0]; // Array of 32 one-bit boolean register variables

time chk_point[1:100]; // Array of 100 time checkpoint variables

reg [4:0] port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits
wide

integer matrix[4:0][0:255]; // Two dimensional array of integers

reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array

wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire

wire w_array1[7:0][5:0]; // Declare an array of single bit wires
```

Arrays: Assignments

```
count[5] = 0; // Reset 5th element of array of count variables
chk_point[100] = 0; // Reset 100th time check point value
port_id[3] = 0; // Reset 3rd element (a 5-bit value) of port_id array.

matrix[1][0] = 33559; // Set value of element indexed by [1][0] to
33559
array_4d[0][0][0][0][15:0] = 0; //Clear bits 15:0 of the register
                                //accessed by indices [0][0][0][0]

port_id = 0; // Illegal syntax - Attempt to write the entire array
```

Verilog Timescale

- The `timescale compiler directive specifies the time unit and precision for the modules that follow it.

```
`timescale <time_unit>/<time_precision>
// for example
`timescale 1ns/1ps
`timescale 10us/100ns
`timescale 10ns/1ns
```

Verilog Timescale

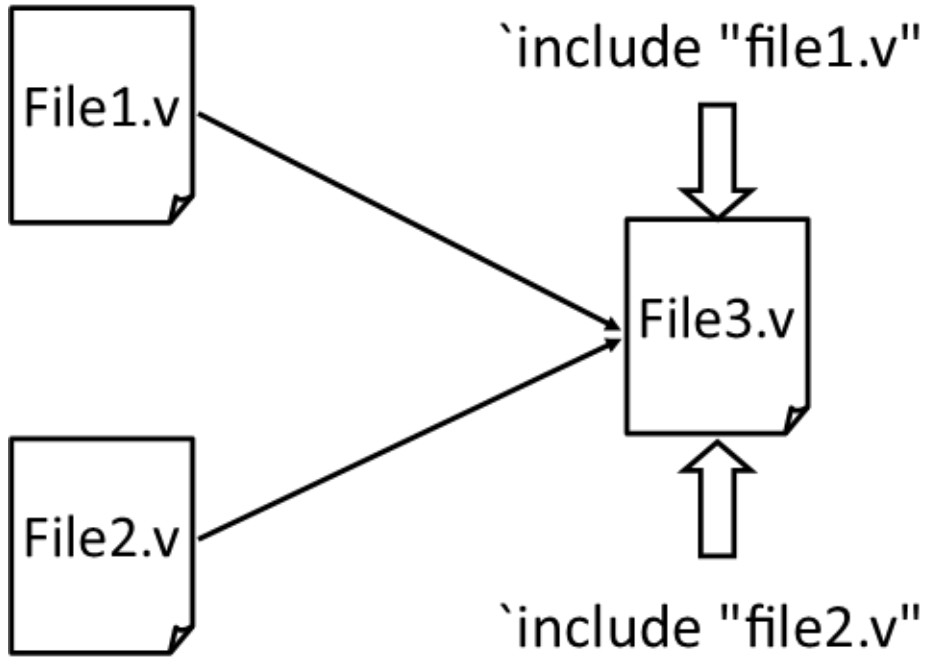
- The `time_unit` is the measurement of delays in simulation time,
- The `time_precision` specifies how delay values are rounded before being used in the simulation.
- `$time` and `$realtime` system functions return the current time, and the default reporting format can be changed with another system task `$timeformat`.

Verilog Timescale

Character	Unit
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	Femtoseconds

Multiple File Design

- `include "path/file.v"



Thank You

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Lab – 3

- Introduction to Sequential circuits.
- Blocking and Non-Blocking Assignments.
- Sequential and Parallel blocks.

Memories

- In digital simulation often needs to model register files, RAMs, and ROMs.
  - Memories are modeled in Verilog simply as a one-dimensional array of registers.
  - Each element of the array is known as an element or word and is addressed by a single array index.
  - Each word can be one or more bits.
    - It is important to differentiate between n 1-bit registers and one n-bit register.

Memories

- reg mem1bit[0:1023]; `// Memory mem1bit with 1K 1-bit words`

- reg [7:0] membyte[0:1023]; `// Memory membyte with 1K 8-bit words(bytes)`

- membyte[511] `// Fetches 1 byte word whose address is 511.`

System Tasks and Compiler Directives

- All system tasks appear in the form $<keyword>.
  - Displaying on the screen.
  - Monitoring values of nets.
  - Stopping, and finishing.
- All compiler directives are defined as `<keyword> construct.
  - Similar to the #define construct in C.

System Tasks - Displaying information

- $display is the main system task for displaying values of variables or strings or expressions.
- The format of $display is very similar to printf in C.
- A $display without any arguments produces a newline.

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex

System Tasks - Displaying information

%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

System Tasks - Displaying information

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World

//Display value of current simulation time 230
$display($time);
-- 230

//Display value of 41-bit virtual address 1fe0000001c at time 200
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c
```

System Tasks - Monitoring information

- Verilog provides a mechanism to monitor a signal when its value changes. This facility is provided by the $monitor task.
- Unlike $display, $monitor needs to be invoked only once.
- Only one monitoring list can be active at a time.
  - If there is more than one $monitor statement in your simulation, the last $monitor statement will be the active statement.
  - The earlier $monitor statements will be overridden.

System Tasks - Monitoring information

- The $monitoron tasks enables monitoring, and the $monitoroff task disables monitoring during a simulation.
- Monitoring is turned on by default at the beginning of the simulation.

```
//Monitor time and value of the signals clock and reset
//Clock toggles every 5 time units and reset goes down at 10 time units
initial
begin
    $monitor($time,
                " Value of signals clock = %b reset = %b", clock,reset);
end
```

System Tasks - Stopping and finishing Simulation

- The $stop task puts the simulation in an interactive mode.
  - The designer can then debug the design from the interactive mode.
- The $finish task terminates the simulation.

```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time 1000.
initial // to be explained later. time = 0
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

Compiler Directives - `define

- The `define directive is used to define text macros in Verilog.

```
//define a text macro that defines default word size
//Used as 'WORD_SIZE in the code
'define WORD_SIZE 32

//define an alias. A $stop will be substituted wherever 'S appears
'define S $stop;

//define a frequently used text string
'define WORD_REG reg [31:0]
// you can then define a 32-bit register as 'WORD_REG reg32;
```

Compiler Directives - `include

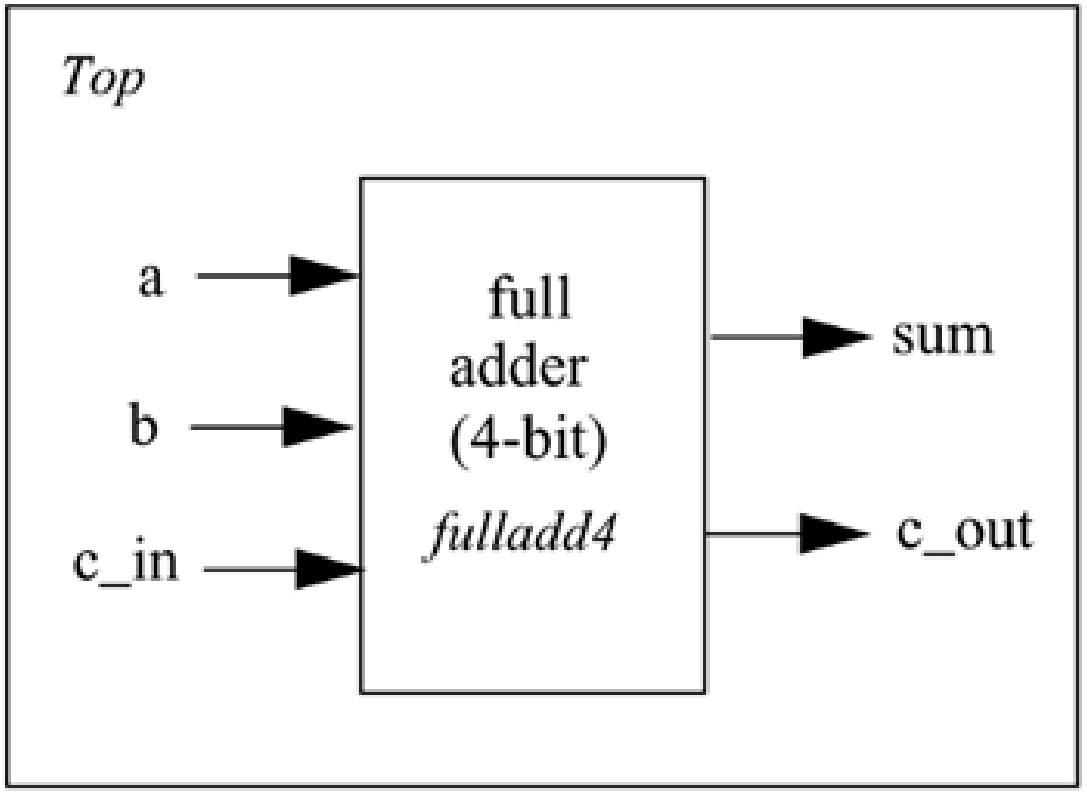
- The `include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation.

```
// Include the file header.v, which contains declarations in the
// main verilog file design.v.
'include header.v
...
...
<Verilog code in file design.v>
...
...
```

Ports

- **List of Ports**

```
module fulladd4(sum,
c_out, a, b, c_in);
  //Module with a list of
ports
module Top;
// No list of ports, top-
level module in simulation
```



Ports

- **Port Declaration**

Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port

Ports

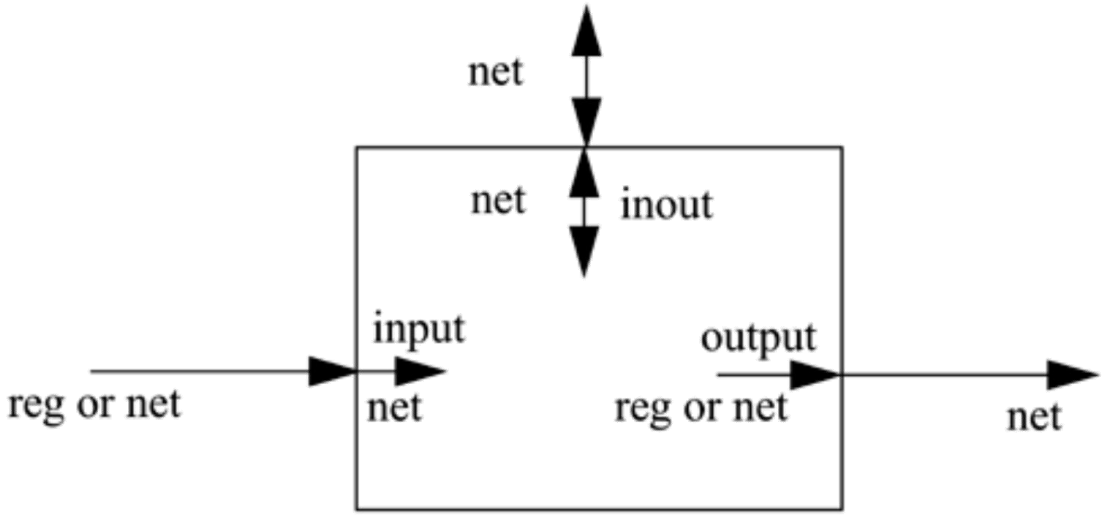
```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_out;

input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
...
endmodule
```

Port Connection Rules

- Inputs
  - Internally, must always be of the type net.
  - Externally, connected to a variable which is a reg or a net.
- Outputs
  - Internally, can be of the type reg or net.
  - Externally, must always be connected to a net.
- Inouts
  - Internally, inout ports must always be of the type net.
  - Externally, inout ports must always be connected to a net.



- Width matching
  - It is legal to connect internal and external items of different sizes when making inter-module port connections.
- Unconnected ports
  - Verilog allows ports to remain unconnected.

Example Illegal Port Connection

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
reg [3:0] SUM;

wire C_OUT;

    //Instantiate fulladd4, call it fa0
    fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
    //Illegal connection because output port sum in module fulladd4
    //is connected to a register variable SUM in module Top.
    .
    .
    <stimulus>
    .
    .
endmodule
```

Connecting Ports to External Signals

- **Connecting by ordered list**

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

    //Instantiate fulladd4, call it fa_ordered.
    //Signals are connected to ports in order (by position)
    fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
    ...
    <stimulus>
    ...
endmodule
```

```
module fulladd4(sum, c_out, a, b, c_in);
output[3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
    ...
    <module internals>
    ...
endmodule
```

Connecting Ports to External Signals

- **Connecting ports by name**

```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN),
.a(A),);
```

initial Statement

- Starts at time 0.
- Executes exactly once during a simulation.
- If Multiple initial block,
  - Each block starts to execute concurrently at time 0.

```
initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end
```

always Statement

- Starts at time 0.
- Executes the statements in the always block continuously in a looping fashion.

```
module clock_gen (output reg clock);
//Initialize clock at time zero
initial
	clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always
	#10 clock = ~clock;
initial
	#1000 $finish;
endmodule
```

Assignments

- c <= a&b;
- assign c = ~a;
- c = 1'b0;

1. <= non-blocking and is performed on every positive edge of clock.
   – these are evaluated in parallel so no guarantee of order
2. **assign =** continual assignment to wire outside an always statement. value of LHS is updated when RHS changes.
3. = blocking assignment, inside always statements enforces sequential order.

Blocking Assignments

- A blocking statement must be executed before the execution of the statements that follow it in a sequential block.

```
// Blocking assignments
initial begin
   a = #10 1'b1;// The simulator assigns 1 to a at time 10
   b = #20 1'b0;// The simulator assigns 0 to b at time 30
   c = #40 1'b1;// The simulator assigns 1 to c at time 70
end
```

Blocking Assignments

```
//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1; //Scalar assignments
        count = 0; //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a; //initialize vectors

        #15 reg_a[2] = 1'b1; //Bit select assignment with delay
        #10 reg_b[15:13] = {x, y, z} //Assign result of concatenation
to
                                     // part select of a vector
        count = count + 1; //Assignment to an integer (increment)
end
```

Non-blocking Assignments

- Nonblocking statements allow you to schedule assignments without blocking the procedural flow.
- Can use the nonblocking procedural to make several register assignments within the same time step without regard to order or dependence upon each other.

```
// Nonblocking assignments
initial begin
   d <= #10 1'b1;// The simulator assigns 1 to d at time 10
   e <= #20 1'b0;// The simulator assigns 0 to e at time 20
   f  <= #40 1'b1;// The simulator assigns 1 to f at time 40
end
```

Non-blocking Assignments

```
//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1; //Scalar assignments
        count = 0; //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a; //Initialize vectors

        reg_a[2] <= #15 1'b1; //Bit select assignment with delay
       reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
                                      //to part select of a vector
        count <= count + 1; //Assignment to an integer (increment)
end
```

Non-blocking Assignments

At each positive edge of clock

- A read operation is performed on each right-hand-side variable.
- The right-hand-side expressions are evaluated.
- results are stored internally in the simulator.
- The write operations to the left-hand-side variables are scheduled to be executed at the time specified.
- The order of write operations is not important because the internally stored right-hand-side expression values are used to assign to the left-hand-side values.

Timing Control

- **Delay-Based Timing Control**

```
//intra assignment delays
initial
begin
        x = 0; z = 0;
        y = #5 x + z; //Take value of x and z at the time=0, evaluate
                      //x + z and then wait 5 time units to assign value
                     //to y.

end
```

Timing Control

- **Event-Based Timing Control**

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                        //a positive transition ( 0 to 1,x or z,
                        // x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                        //a negative transition ( 1 to 0,x or z,
                        //x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned
                        //to q at the positive edge of clock
```

Timing Control

- **Delayed Assignment Vs Delayed Execution**

```
// Nonblocking assignments
initial begin
   d <= #10 1'b1;// The simulator assigns 1 to d at time 10
   $display($time," d = %b, e = %b, f = %b",d,e,f);
   e <= #20 1'b0;// The simulator assigns 0 to e at time 20
   $display($time," d = %b, e = %b, f = %b",d,e,f);
   f  <= #40 1'b1;// The simulator assigns 1 to f at time 40
   $display($time," d = %b, e = %b, f = %b",d,e,f);
   #50 ;
   $display($time," d = %b, e = %b, f = %b",d,e,f);
end
```

```
 0 d = x, e = x, f = x
 0 d = x, e = x, f = x
 0 d = x, e = x, f = x
50 d = 1, e = 0, f = 1
```

Timing Control

- **Delayed Assignment Vs Delayed Execution**

```
// Nonblocking assignments
initial begin
  #10 d <= 1'b1;// The simulator execute this line at time 10
  $display($time," d = %b, e = %b, f = %b",d,e,f);
  #20 e <= 1'b0;// The simulator execute this line at time 20
  $display($time," d = %b, e = %b, f = %b",d,e,f);
  #40 f  <= 1'b1;// The simulator execute this line at time 40
  $display($time," d = %b, e = %b, f = %b",d,e,f);
  #50 ;
  $display($time," d = %b, e = %b, f = %b",d,e,f);
end
```

```
 10 d = x, e = x, f = x
 30 d = 1, e = x, f = x
 70 d = 1, e = 0, f = x
120 d = 1, e = 0, f = 1
```

While Loop

```
initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                            //exit at count 128
        begin
                $display("Count = %d", count);
               count = count + 1;
        end
end
```

For Loop

```
integer count;

initial
    for ( count=0; count < 128; count = count + 1)
        $display("Count = %d", count);
```

Repeat Loop

```
initial
begin
    count = 0;
    repeat(128)
    begin
        $display("Count = %d", count);
        count = count + 1;
    end
end
```

Forever loop

```
//Use forever loop instead of always block
reg clock;

initial
begin
        clock = 1'b0;
        forever #10 clock = ~clock; //Clock with period of 20 units
end

//Example 2: Synchronize two register values at every positive edge of
//clock
reg clock;
reg x, y;

initial
        forever @(posedge clock) x = y;
```

Sequential Block

- All the statement in this group executes sequentially.
  - Not applicable for nonblocking assignments.

Example - 1:

```
reg a,b,c;
initial
begin
      a = 1'b1;
      b = 1'b0;
      c = 1'b1;
end
```

Example - 2:

```
reg a,b,c;
initial
begin
      #5 a = 1'b1;
      #10 b = 1'b0;
      #15 c = 1'b1;
end
```

Parallel Block

- Execute in parallel.
  - If the sequencing is required then it can be given by providing some delays before the statements.

```
reg a,b,c;
initial
fork
    #5 a = 1'b1;
    #10 b = 1'b0;
    #15 c = 1'b1;
join
```

```
reg a,b,c,d;
initial
begin
fork
    #5 a = 1'b1;
    #10 b = 1'b0;
    #15 c = 1'b1;
join
#30 d = 1'b0;
end
```

Thank You

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Lab – 4

- To design a Simple MIPS ALU in a step-by-step manner.
- To design the main Control PLA to generate control signals.

Concatenation Operator

- The concatenation operator ({, }) provides a mechanism to append multiple operands.
- The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.
- Concatenations are expressed as operands within braces, with commas separating the operands.
  - Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

Concatenation Operator

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant.
- A replication constant specifies how many times to replicate the number inside the brackets ({ }).

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

Generate Blocks

- Code to be generated dynamically at elaboration time before the simulation begins.
- Facilitates the creation of parametrized models.
- Convenient when the same operation or module instance is repeated for multiple bits of a vector

Generate Blocks

- Generated instantiations can be one or more of the following types:
  - Modules
  - User defined primitives
  - Verilog gate primitives
  - Continuous assignments
  - initial and always blocks

Generate Blocks

- Generated instances have unique identifier names and can be referenced hierarchically.
- Generate statements permit the following Verilog data types to be declared within the generate scope:
  - net, reg
  - integer, real, time, realtime
  - event

Generate Blocks

- Some module declarations and module items are not permitted in a generate statement.
  - parameters, local parameters
  - input, output, inout declarations
  - specify blocks

Generate Blocks

- There are three methods to create generate statements:
  - Generate loop
  - Generate conditional
  - Generate case

Generate Loop

- A generate loop permits one or more of the following to be instantiated multiple times using a for loop:
  - Variable declarations
  - Modules
  - User defined primitives, Gate primitives
  - Continuous assignments
  - initial and always blocks

Example Bit-wise XOR of Two N-bit Buses

```
// This module generates a bit-wise xor of two N-bit buses
module bitwise_xor (out, i0, i1);
    // Parameter Declaration. This can be redefined
    parameter N = 32; // 32-bit bus by default
    // Port declarations
    output [N-1:0] out;
    input [N-1:0] i0, i1;
    // Declare a temporary loop variable. This variable is used only
    // in the evaluation of generate blocks. This variable does not
    // exist during the simulation of a Verilog design
    genvar j;
    //Generate the bit-wise Xor with a single loop
```

Example Bit-wise XOR of Two N-bit Buses

```
    //Generate the bit-wise Xor with a single loop
    generate for (j=0; j<N; j=j+1) begin: xor_loop
    xor g1 (out[j], i0[j], i1[j]);
    end //end of the for loop inside the generate block
    endgenerate //end of the generate block
    // As an alternate style,
    // the xor gates could be replaced by always blocks.
    // reg [N-1:0] out;
    //generate for (j=0; j<N; j=j+1) begin: bit
    // always @(i0[j] or i1[j]) out[j] = i0[j] ^ i1[j];
    //end
    //endgenerate
endmodule
```

Observations from Generate Loop

- Simulator create a flat representation without the generate blocks.
- genvar is a keyword used to declare variables that are used only in the evaluation of generate block.
  - Genvars do not exist during simulation of the design.
- The value of a genvar can be defined only by a generate loop.

Observations from Generate Loop

- Generate loops can be nested.
  - two generate loops using the same genvar as an index variable cannot be nested.
- The name xor_loop assigned to the generate loop is used for hierarchical name referencing of the variables inside the generate loop.
- The relative hierarchical names of the xor gates will be xor_loop[0].g1, xor_loop[1].g1, ……., xor_loop[31].g1.

Generate Conditional

- A generate conditional is like an if-else-if generate construct that permits the following:
  - Modules
  - User defined primitives, Gate primitives
  - Continuous assignments
  - initial and always blocks

Parametrized Multiplier using Generate Conditional

```
module multiplier (product, a0, a1);
    // Parameter Declaration. This can be redefined
    parameter a0_width = 8; // 8-bit bus by default
    parameter a1_width = 8; // 8-bit bus by default
    // Local Parameter declaration.
    // This parameter cannot be modified with defparam or
    // with module instance # statement.
    localparam product_width = a0_width + a1_width;
    // Port declarations
    output [product_width -1:0] product;
    input [a0_width-1:0] a0;
    input [a1_width-1:0] a1;
```

Parametrized Multiplier using Generate Conditional

```
    // Instantiate the type of multiplier conditionally.
    // Depending on the value of the a0_width and a1_width
    // parameters at the time of instantiation, the appropriate
    // multiplier will be instantiated.
    generate
        if (a0_width <8) || (a1_width < 8)
            cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
        else
            tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
    endgenerate //end of the generate block
endmodule
```

Generate Case

- A generate case permits the following Verilog constructs to be conditionally instantiated into another module based on a select-one-of-many case construct that is deterministic at the time the design is elaborated.
  - Modules
  - User defined primitives, Gate primitives
  - Continuous assignments
  - initial and always blocks

Generate Case Example

```
module adder(co, sum, a0, a1, ci);
    // Parameter Declaration. This can be redefined
    parameter N = 4; // 4-bit bus by default
    // Port declarations
    output [N-1:0] sum;
    output co;
    input [N-1:0] a0, a1;
    input ci;
```

Generate Case Example

```
    // Instantiate the appropriate adder based on the width of the bus.
    // This is based on parameter N that can be redefined at
    // instantiation time.
    generate
    case (N)
        //Special cases for 1 and 2 bit adders
        1: adder_1bit adder1(c0, sum, a0, a1, ci); //1-bit implementation
        2: adder_2bit adder2(c0, sum, a0, a1, ci); //2-bit implementation
        // Default is N-bit carry look ahead adder
        default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
    endcase
    endgenerate //end of the generate block
endmodule
```

TASK 1

Design the ALU for MIPS ISA, such that it will implement the following instructions.

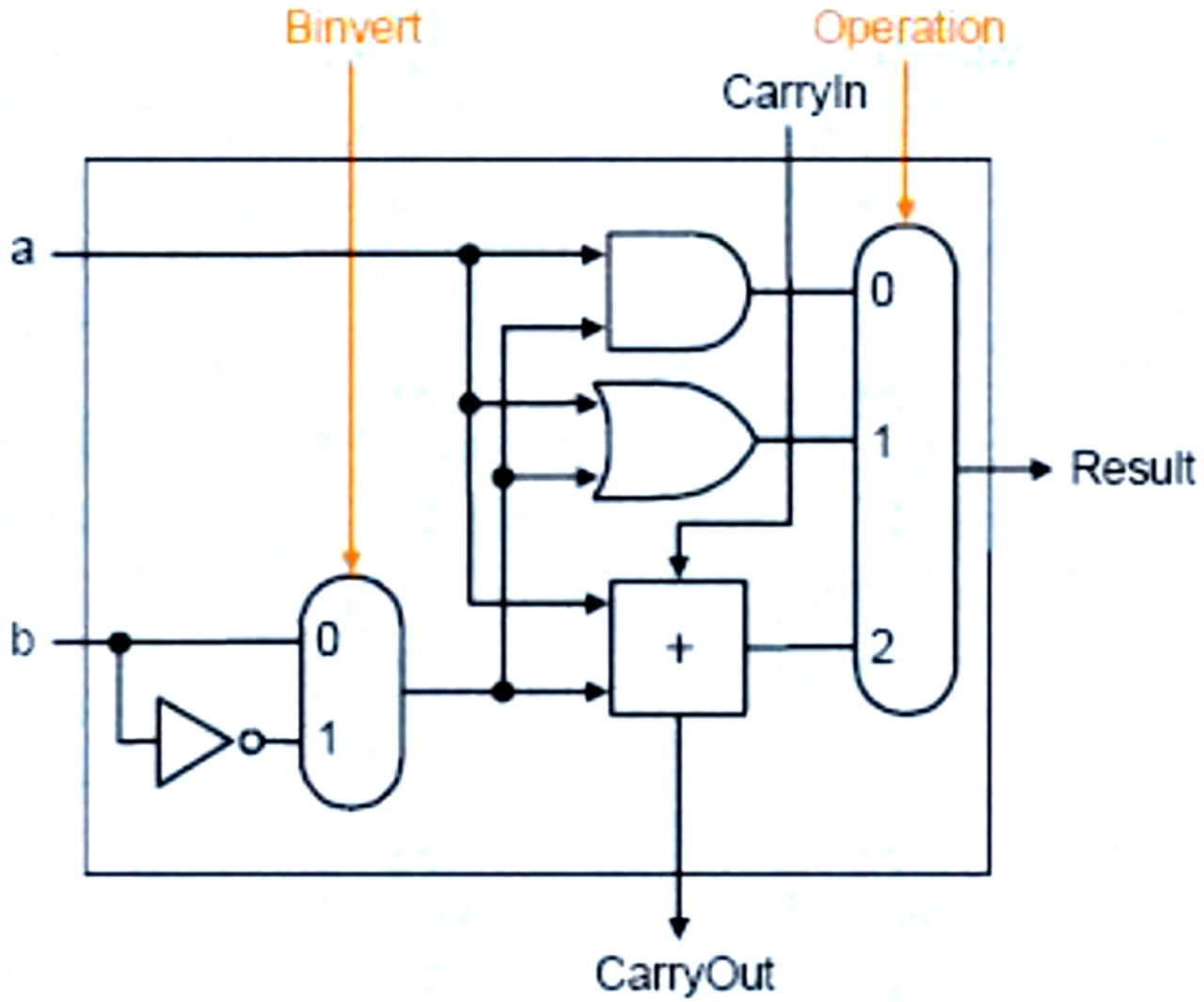
(i) and $s1, $s2, $s3

(ii) or $s1, $s2, $s3

(iii) add $s1, $s2, $s3

(iv) sub $s1, $s2, $s3

TASK 1



TASK 1

- **Sub Task 1.1:** In this sub-task let us design the 32 bit 2x1Multiplexer. We begin with a 2:1 Multiplexer.

```
module mux2to1(out,sel,in1,in2);
   input in1,in2,sel;
   output out;
   wire not_sel,a1,a2;
   not (not_sel,sel);
   and (a1,sel,in2);
   and (a2,not_sel,in1);
   or(out,a1,a2);
endmodule
```

TASK 1

Sub Task 1.1: Extend 1 bit 2x1 Mux to 8 bit 2x1 Mux

```
module bit8_2to1mux(out,sel,in1,in2);
    input [7:0] in1,in2;
    output [7:0] out;
    input sel;
    mux2to1 m0(out[0],sel,in1[0],in2[0]);
    mux2to1 m1(out[1],sel,in1[1],in2[1]);
    mux2to1 m2(out[2],sel,in1[2],in2[2]);
    mux2to1 m3(out[3],sel,in1[3],in2[3]);
    mux2to1 m4(out[4],sel,in1[4],in2[4]);
    mux2to1 m5(out[5],sel,in1[5],in2[5]);
    mux2to1 m6(out[6],sel,in1[6],in2[6]);
    mux2to1 m7(out[7],sel,in1[7],in2[7]);
endmodule
```

TASK 1

Sub Task 1.1: Extend 1 bit 2x1 Mux to 8 bit 2x1 Mux

```
module bit8_2to1mux(out,sel,in1,in2);
  input [7:0] in1,in2;
  output [7:0] out;
  input sel;
  genvar j;
//this is the variable that is be used in the generate
//block
  generate      for   (j=0;   j<8;j=j+1)   begin:   mux_loop
//mux_loop is the name of the loop
  mux2to1 m1(out[j],sel,in1[j],in2[j]);
 //mux2to1 is instantiated every time it is called
end
endgenerate
endmodule
```

TASK 1

Sub Task 1.1: Extend 8 bit 2x1 Mux to 32 bit 2x1 Mux

```
module  Mux32Bit_2To1(out, select, in1, in2);
  input [31:0] in1, in2;
  input select;
  output  [31:0] out;
  Mux8Bit_2To1_generate Mux1(out[7:0], select, in1[7:0], in2[7:0]);
  Mux8Bit_2To1_generate Mux2(out[15:8], select, in1[15:8], in2[15:8]);
  Mux8Bit_2To1_generate Mux3(out[23:16], select, in1[23:16], in2[23:16]);
  Mux8Bit_2To1_generate Mux4(out[31:24], select, in1[31:24], in2[31:24]);
endmodule
```

TASK 1

Sub Task 1.1: Extend 32 bit 2x1 Mux to 32 bit 4x1 Mux

```
module  Mux32Bit_4To1(out, select, in1, in2, in3, in4);
  input [31:0]  in1, in2, in3, in4;
  input [1:0] select;
  output  [31:0]  out;
  wire  [31:0]  w1, w2;
  Mux32Bit_2To1 Mux0(w1[31:0], select[0], in1[31:0], in2[31:0]);
  Mux32Bit_2To1 Mux1(w2[31:0], select[0], in3[31:0], in4[31:0]);
  Mux32Bit_2To1 Mux2(out[31:0], select[1], w1[31:0], w2[31:0]);
endmodule
```

TASK 1

- **Sub Task 1.2:** In this task we will design the 32-bit wide AND and OR gates.

```
module  AND32Bit(out, in1, in2);
   input [31:0]   in1, in2;
   output  [31:0]   out;
   assign  {out} = in1 & in2;
endmodule
```

TASK 1

- **Sub Task 1.2:** In this task we will design the 32-bit wide AND and OR gates.

```
module  OR32Bit(out, in1, in2);
  input [31:0]  in1, in2;
  output  [31:0]  out;
  assign  {out} = in1 | in2;
endmodule
```

TASK 1

- **Subtask 1.3:** designed a 1 bit Full Adder using dataflow modeling.

```
module FA_dataflow (Cout, Sum,In1,In2,Cin);
  input In1,In2;
  input Cin;
  output Cout;
  output Sum;
  assign {Cout,Sum}=In1+In2+Cin;
endmodule
```

- expand it to make a 8-bit adder.

TASK 1

- **Subtask 1.3:** designed a 8 bit Full Adder using dataflow modeling.

```
module FA_dataflow_8Bit(carry, sum, A, B, CarryIn);
  input [7:0] A, B;
  input CarryIn;
  output  [7:0]sum;
  output  carry;
  wire  c1, c2, c3, c4, c5, c6, c7;
  FA_dataflow_1Bit  mod1(c1, sum[0], A[0], B[0], CarryIn);
  FA_dataflow_1Bit  mod2(c2, sum[1], A[1], B[1], c1);
  FA_dataflow_1Bit  mod3(c3, sum[2], A[2], B[2], c2);
  FA_dataflow_1Bit  mod4(c4, sum[3], A[3], B[3], c3);
  FA_dataflow_1Bit  mod5(c5, sum[4], A[4], B[4], c4);
  FA_dataflow_1Bit  mod6(c6, sum[5], A[5], B[5], c5);
  FA_dataflow_1Bit  mod7(c7, sum[6], A[6], B[6], c6);
  FA_dataflow_1Bit  mod8(carry, sum[7], A[7], B[7], c7);
endmodule
```

- expand it to make a 32-bit adder.

TASK 1

- **Subtask 1.3:** designed a 32 bit Full Adder using dataflow modeling.

```
module FA_dataflow_32Bit(carry, sum, A, B, CarryIn);
  input [31:0] A, B;
  input CarryIn;
  output  [31:0]sum;
  output  carry;
  wire  c1, c2, c3;
  FA_dataflow_8Bit  mod1(c1, sum[7:0], A[7:0], B[7:0], CarryIn);
  FA_dataflow_8Bit  mod2(c2, sum[15:8], A[15:8], B[15:8], c1);
  FA_dataflow_8Bit  mod3(c3, sum[23:16], A[23:16], B[23:16], c2);
  FA_dataflow_8Bit  mod4(carry, sum[31:24], A[31:24], B[31:24], c3);
endmodule
```

TASK 1

- **Final Task:** Now integrate the above modules to make the final ALU.

```
module  ALU32Bit(CarryOut, Result, A, B, Op, BiInvert);
  output  [31:0]  Result;
  output  CarryOut;
  input [31:0]  A, B;
  input [1:0] Op;
  input BiInvert;
  wire  [31:0]  wAnd, wOr, wAdd, wMux, notB;
  assign  {notB} = ~B;
  AND32Bit  And(wAnd, A, B);
  OR32Bit Or(wOr, A, B);
  Mux32Bit_2To1 Mux1(wMux, BiInvert, B, notB);
  FA_dataflow_32Bit  Add(CarryOut, wAdd, A, wMux, BiInvert);
  Mux32Bit_4To1 Mux2(Result, Op, wAnd, wOr, wAdd, 32'h00000000);
endmodule
```

TASK 1

- **Final Task:** Now test final ALU module with test bench.

```
module  ALUTestBench;
  reg BiInvert;
  reg [1:0] Op;
  reg [31:0] A,B;
  wire  [31:0] Result;
  wire  CarryOut;
  ALU32Bit  ALU(CarryOut, Result, A, B, Op, BiInvert);
  initial begin
    $monitor($time, " :A = %b,\n\t B = %b,\n\t Operartion = %b,\n\t BiInvert =
    %b,\n\t Result = %b,\n\t Carry Out = %b.", A, B, Op, BiInvert, Result, CarryOut);
    #0  A = 32'ha5a5a5a5; B = 32'h5a5a5a5a; Op = 2'b00; BiInvert = 1'b0;
    //must perform AND resulting in zero
    #100  Op = 2'b01;        //OR
    #100  Op = 2'b10;        //ADD
    #100  BiInvert = 1'b1;   //SUB
    #200  $finish;
  end
endmodule
```

Thank You

innovate achieve lead

BITS Pilani

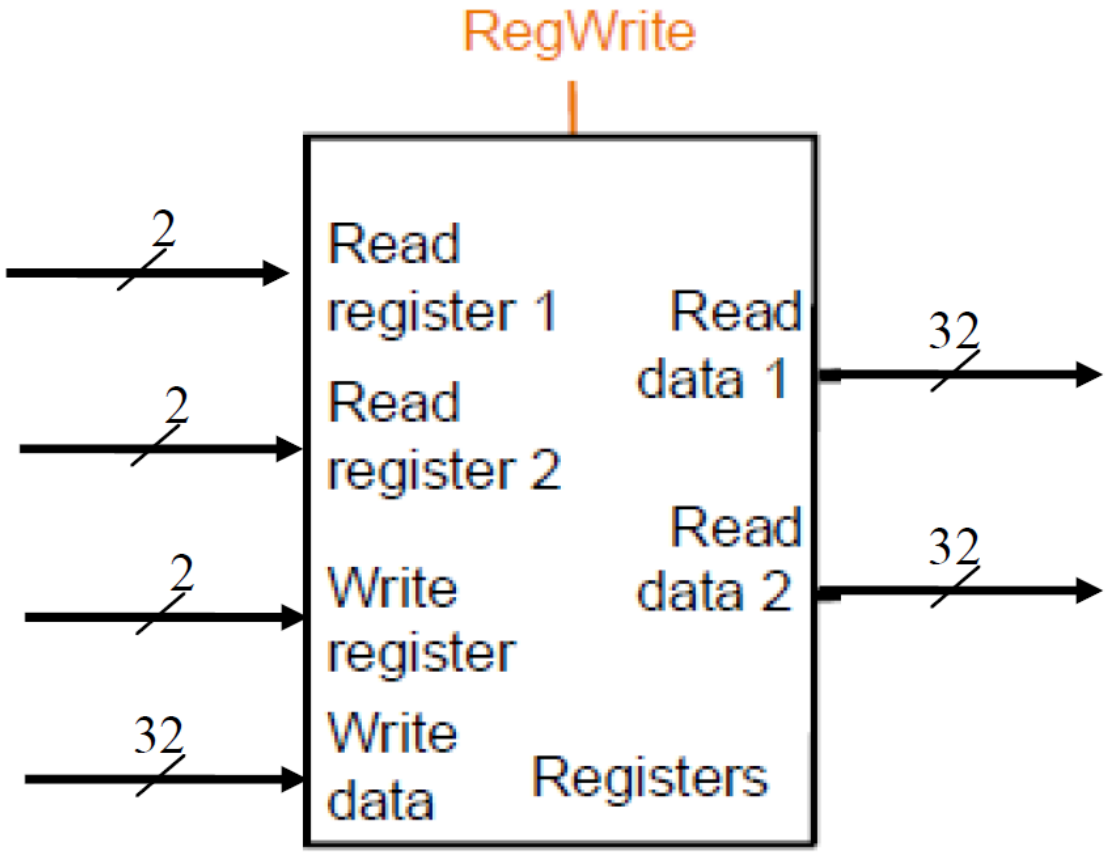
Pilani|Dubai|Goa|Hyderabad

Lab – 5

- Designing a sample 32-bit register file for a MIPS processor

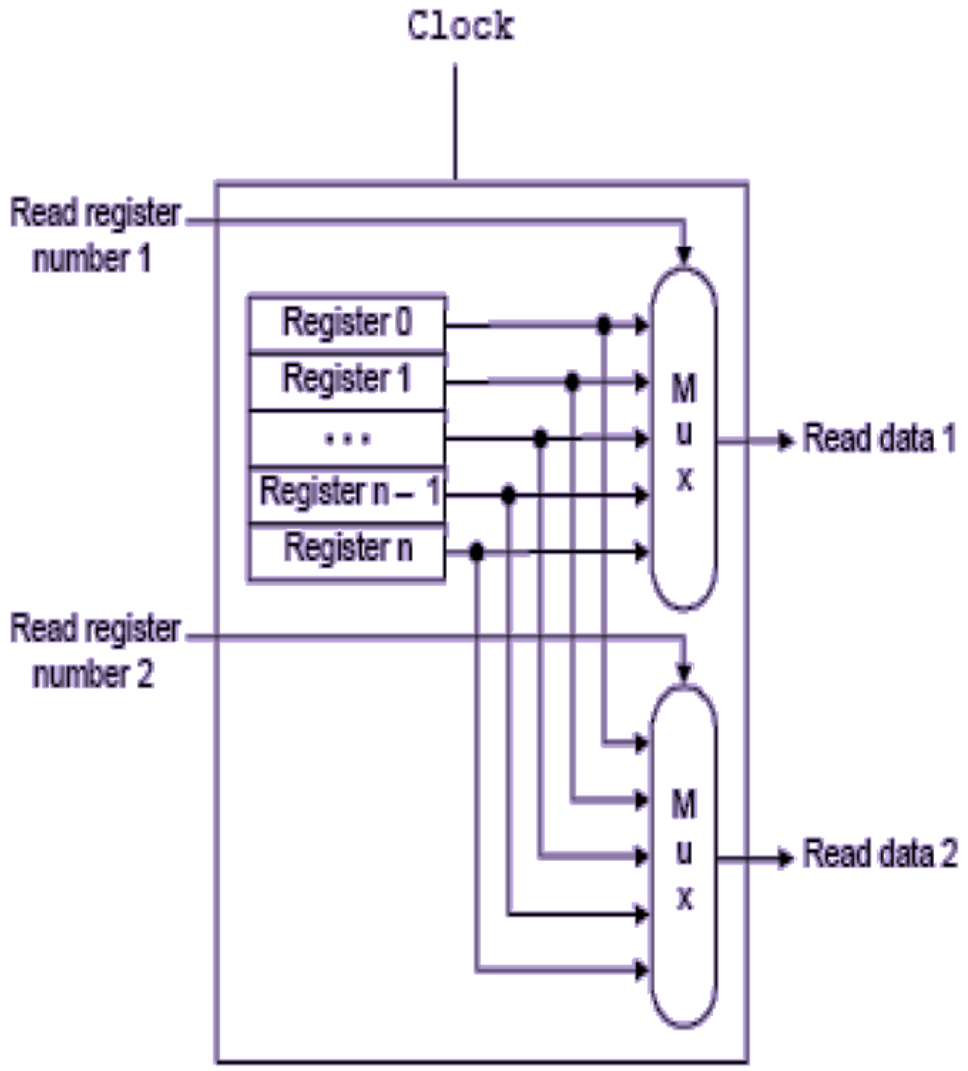
Designing 32-bit Register File

- Assume there are four 32-bit registers in a register file.
- Each register is made up of D flip-flop.
- Register file can read two registers at a time.



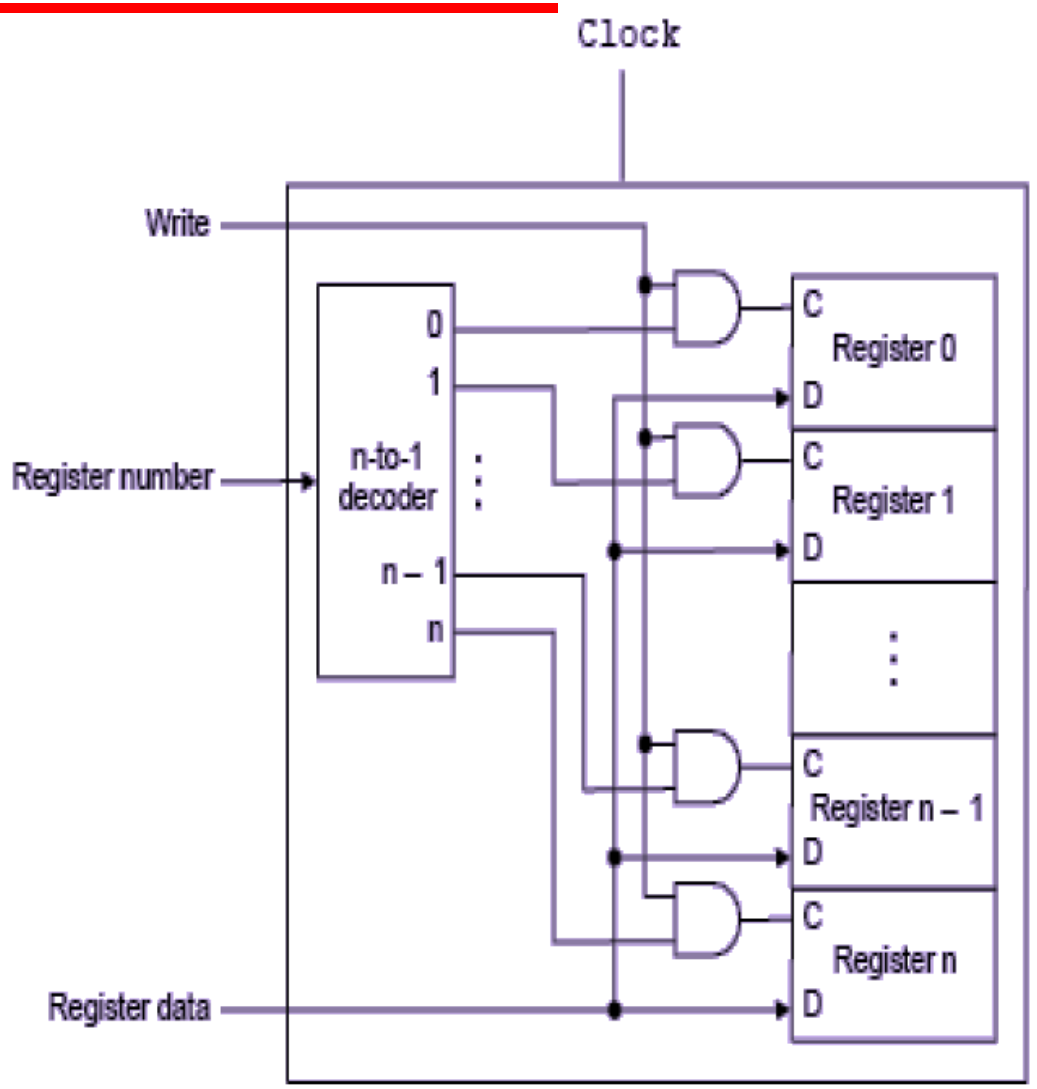
Read Register Module

- Input: -
  - Two register numbers
  - clock
- Output
  - Register data
- Two multiplexers are used select read registers.

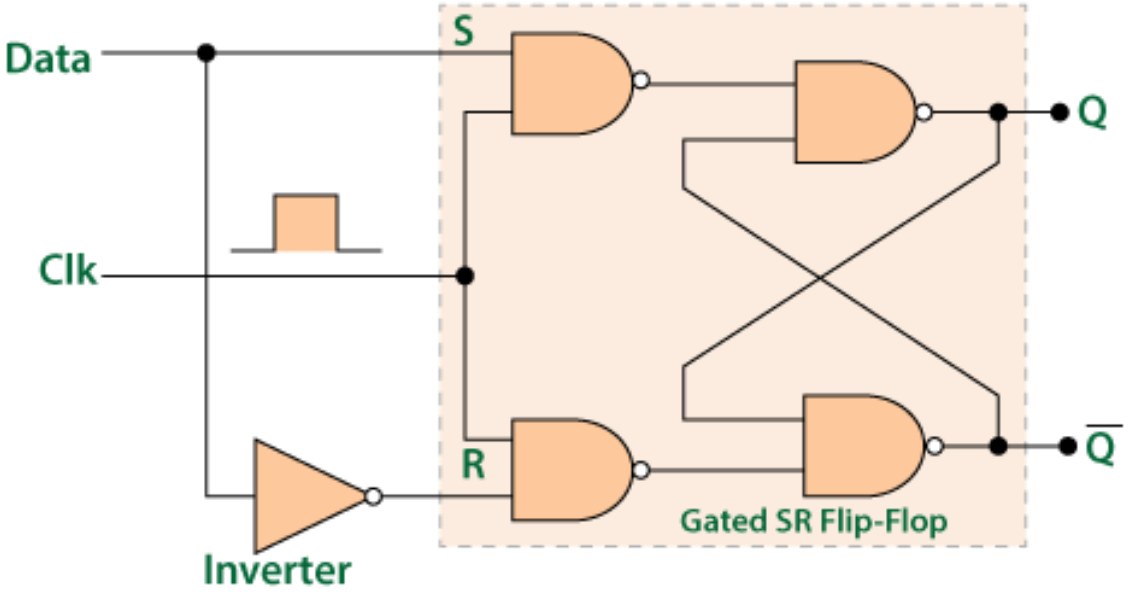


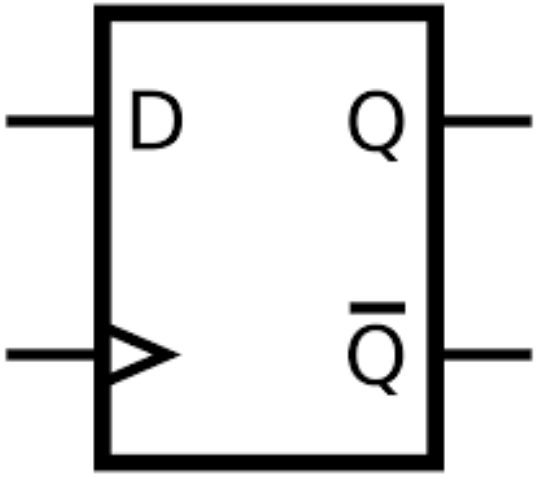
Write Register Module

- Input: -
  - Register data
  - Register number (Decoder I/P)
  - Write enable signal
  - Clock
- 2:4 decoder takes register number as input and enables required register



Sub task 1.1: Designing the registers





Sub task 1.1: Designing the registers

```
module d_ff(q, d, clock, reset);
  input d, clock, reset;
  output  q;
  reg q;
  always @ (posedge clock or negedge reset)
  if(~reset)
    q = 1'b0;
  else
    q = d;
endmodule
```

```
module reg_32bit(q, d, clock, reset);
  input [31:0]  d;
  input clock, reset;
  output  [31:0]  q;
  genvar j;
  generate
    for(j = 0; j < 32; j = j + 1) begin:  d_loop
      d_ff ff(q[j], d[j], clock, reset);
    end
  endgenerate
endmodule
```

Sub task 1.1: Designing the registers

- Four 32-bit registers file

```
reg_32bit r0(w0, WriteData, c0, Reset);
reg_32bit r1(w1, WriteData, c1, Reset);
reg_32bit r2(w2, WriteData, c2, Reset);
reg_32bit r3(w3, WriteData, c3, Reset);
```

Sub task 1.2: Designing the Decoder

- 2-bit address is used to enable the writing to a register by using a decoder. So, we use a 2:4 decoder.

```
module decoder2_4(Out, In);
   input [1:0] In;
   output  [3:0] Out;
   assign  Out[0] = (~In[1] & ~In[0]),
           Out[1] = (~In[1] & In[0]),
           Out[2] = (In[1] & ~In[0]),
           Out[3] = (In[1] & In[0]);
endmodule
```

Sub task 1.3: Designing the Multiplexer

- The register whose data is to be read on a particular read port is selected using this mux. Thus, it will be 4:1 mux as we have 4 registers.

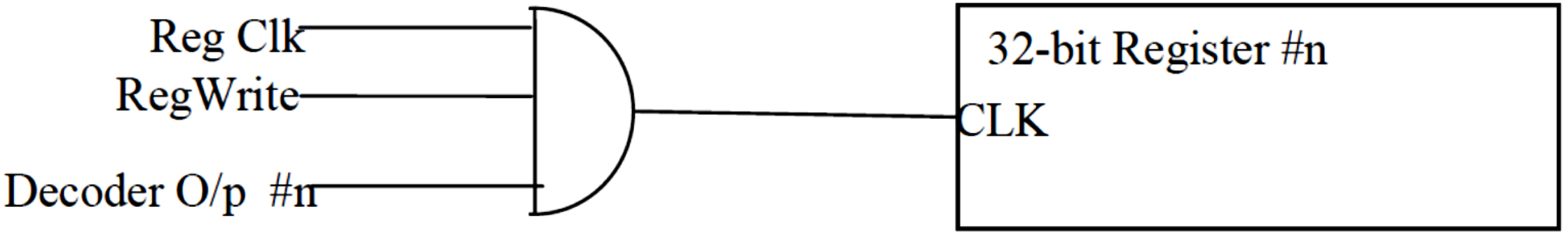
```
module mux4_1(Out, Data0, Data1, Data2, Data3, Select);
  input [31:0]  Data0, Data1, Data2, Data3;
  input [1:0] Select;
  output  [31:0]  Out;
  reg [31:0]  Out;
  always @ (Data0 or Data1 or Data2 or Data3 or Select)
    case  (Select)
      2'b00:  Out = Data0;
      2'b01:  Out = Data1;
      2'b10:  Out = Data2;
      2'b11:  Out = Data3;
    endcase
endmodule
```

Sub task 1.4: Integrating the subcomponents

1. **module** RegFile_4(ReadData1, ReadData2, Clock, Reset, RegWrite, ReadReg1, ReadReg2, WriteRegNo, WriteData);
2. decoder2_4 dec(Decode, WriteRegNo);
3. We must enable writing of only one register depending on the address specified by “Write Register”.

Sub task 1.4: Integrating the subcomponents

3. implement this by ANDing the Register File clock, decoder output for a register and RegWrite Signal.




```
and g0(c0, RegWrite, Decode[0], Clock);
and g1(c1, RegWrite, Decode[1], Clock);
and g2(c2, RegWrite, Decode[2], Clock);
and g3(c3, RegWrite, Decode[3], Clock);
```

Sub task 1.4: Integrating the subcomponents

4. Four 32 bits registers –

```
reg_32bit r0(w0, WriteData, c0, Reset);
reg_32bit r1(w1, WriteData, c1, Reset);
reg_32bit r2(w2, WriteData, c2, Reset);
reg_32bit r3(w3, WriteData, c3, Reset);
```

Register Clock

Sub task 1.4: Integrating the subcomponents

5. Two 4x1 Mux to select two read registers.

```
mux4_1   m0(ReadData1, w0, w1, w2, w3, ReadReg1);
mux4_1   m1(ReadData2, w0, w1, w2, w3, ReadReg2);
```

6. Test your design using test_module.

Thank You

innovate achieve lead

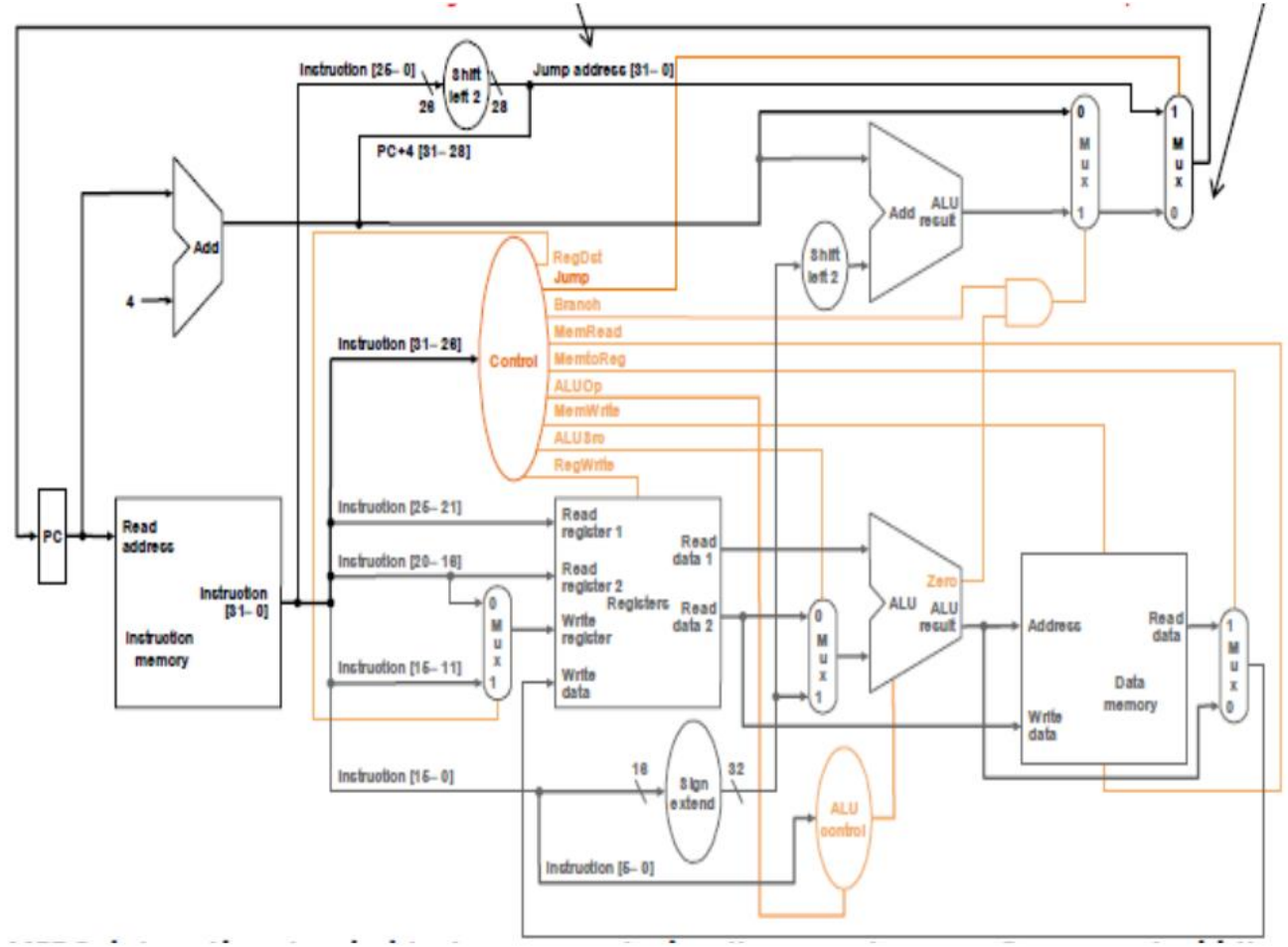
BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Lab – 6

- Single Cycle Datapath Design

Single Cycle Datapath



Single Cycle Datapath

- Datapath consists of the functional units of the processor.
  - Elements that hold data.
    - Program counter, register file, instruction memory, etc.
  - Elements that operate on data.
    - ALU, adders, etc.
  - Buses for transferring data between elements.
  - Control commands the Datapath regarding when and how to route and operate on data.

Single Cycle Datapath

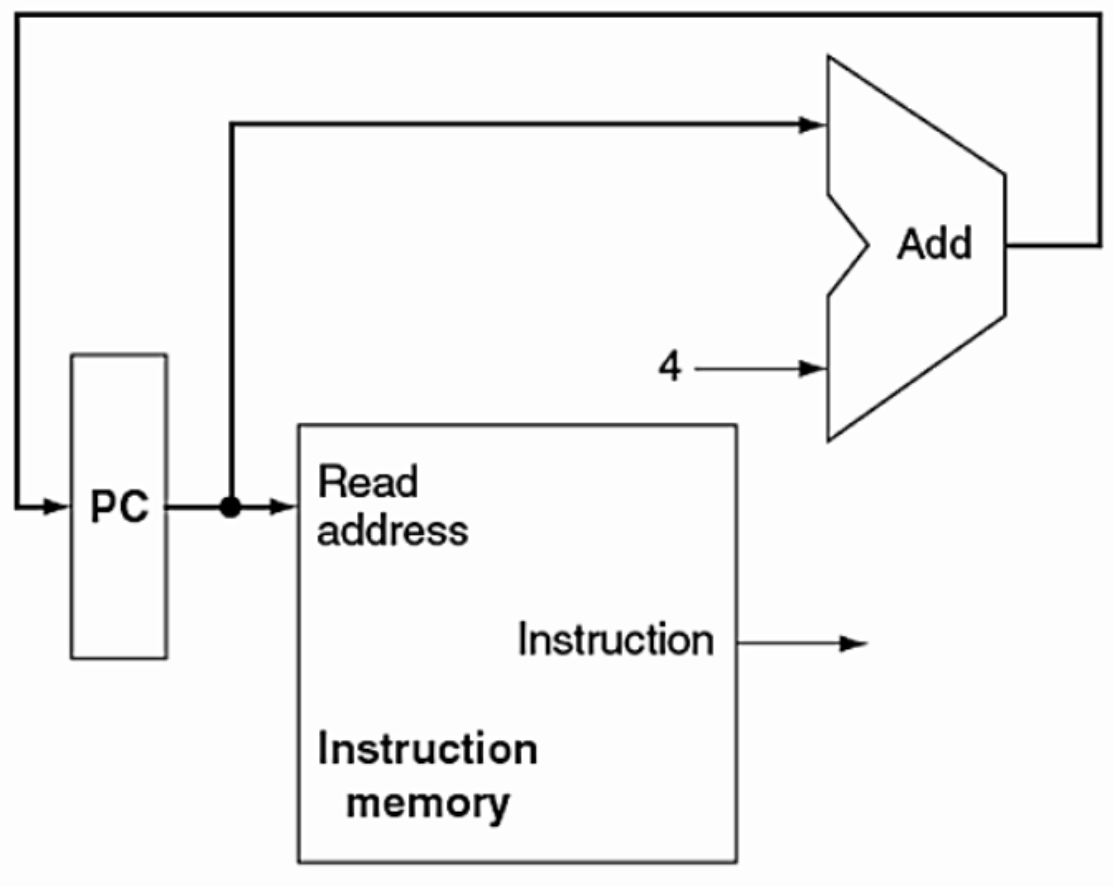
- To showcase the process of creating a datapath and designing a control, we will be using a subset of the MIPS instruction set. Our available instructions include:
  - add, sub, and, or, slt
  - lw, sw
  - beq, j

Steps Taken to Execute a Program

- Now, we can talk about the general steps taken to execute a program.
  - Instruction fetching: use the address in the PC to fetch the current instruction from instruction memory.
  - Instruction decoding: determine the fields within the instruction
  - Instruction execution: perform the operation indicated by the instruction.
  - Update the PC to hold the address of the next instruction.

Steps Taken to Execute a Program

- Fetch the instruction at the address in PC.
- Decode the instruction.
- Execute the instruction.
- Update the PC to hold the address of the next instruction.



Note: we perform PC+4 because MIPS instructions are word-aligned.

R-Format Instructions

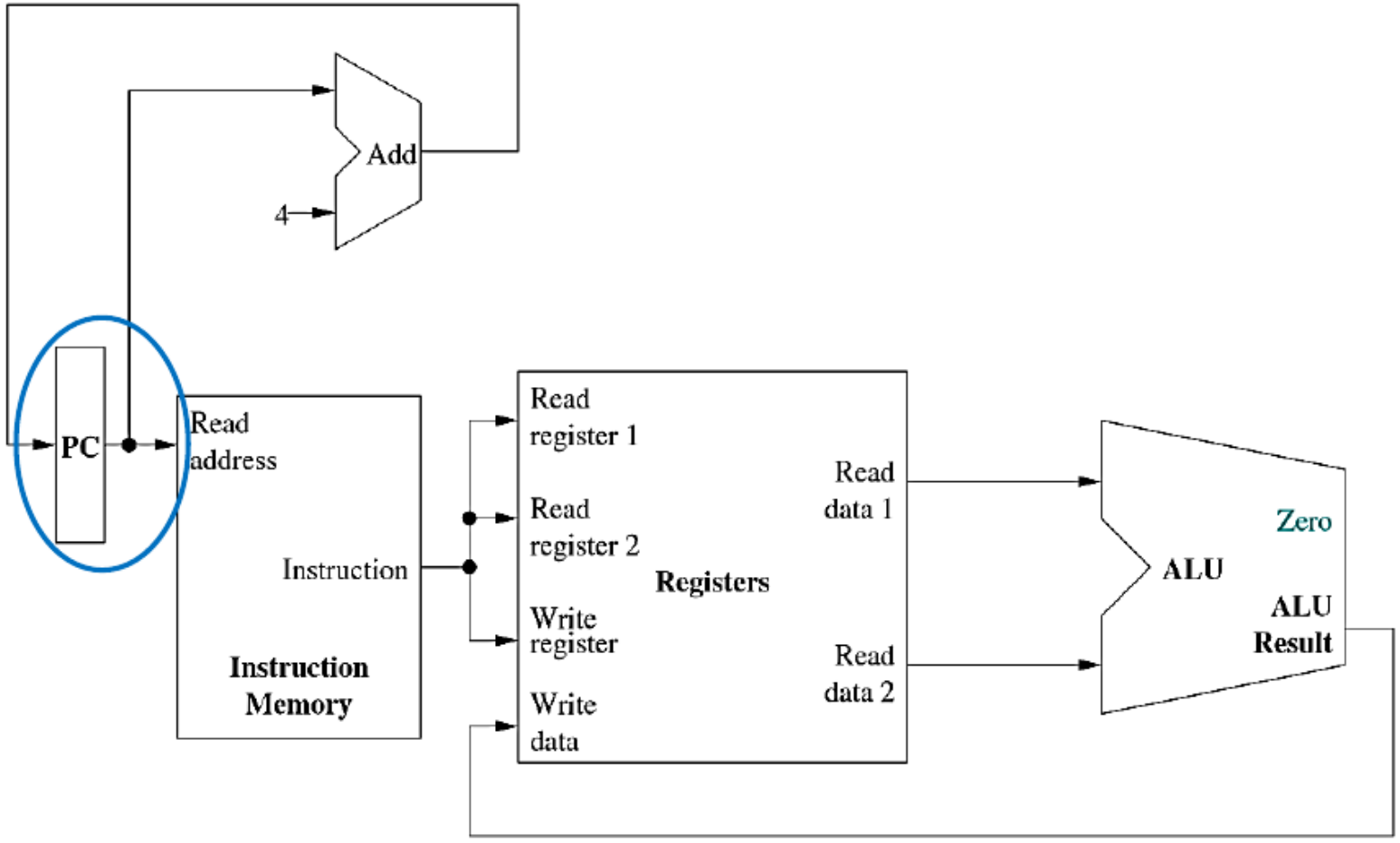
- All R-format instructions read two registers, rs and rt, and write to a register rd.

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.
shamt – shift amount.
funct – additional opcodes.

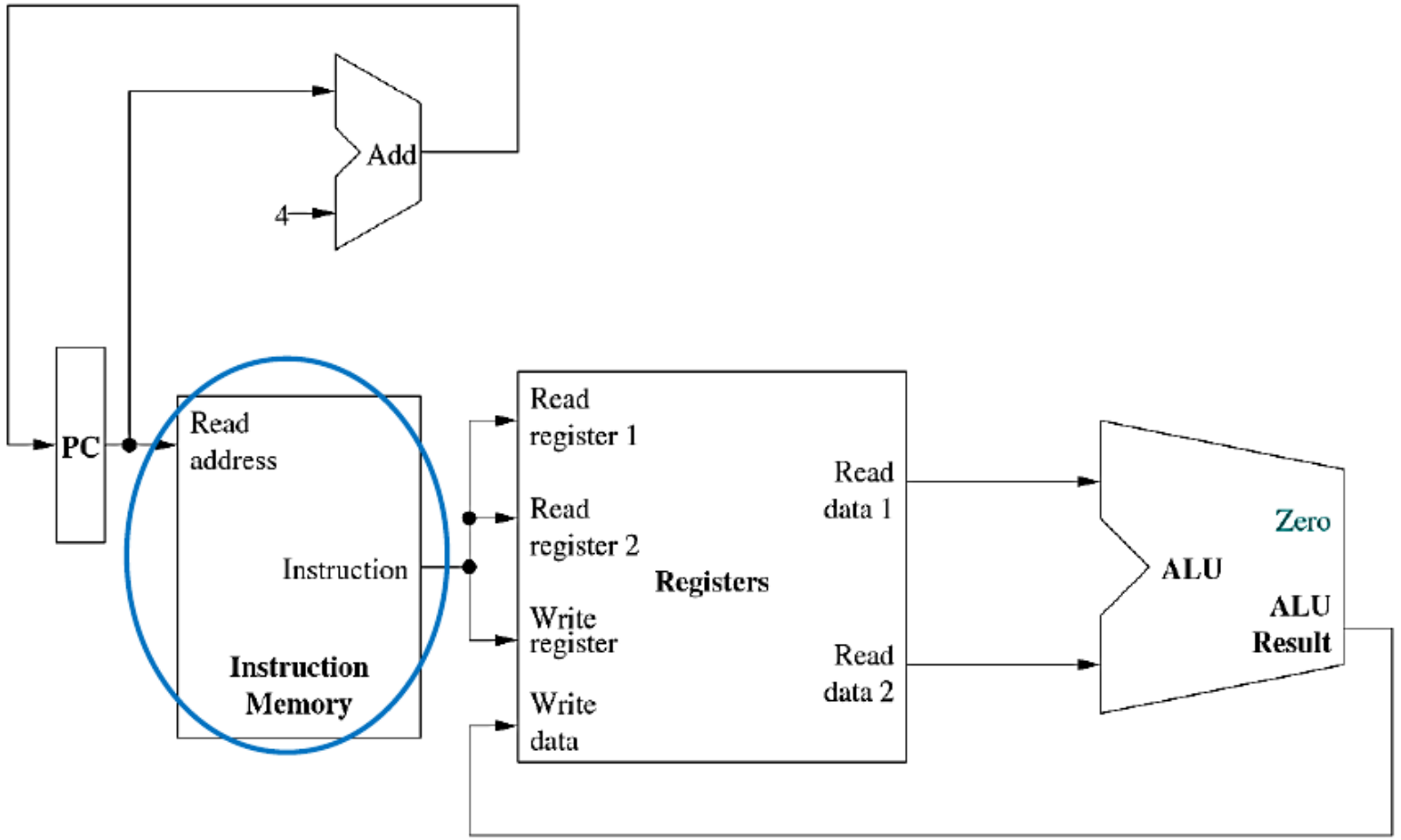
Datapath for R-format Instructions.

1. Grab instruction address from PC.



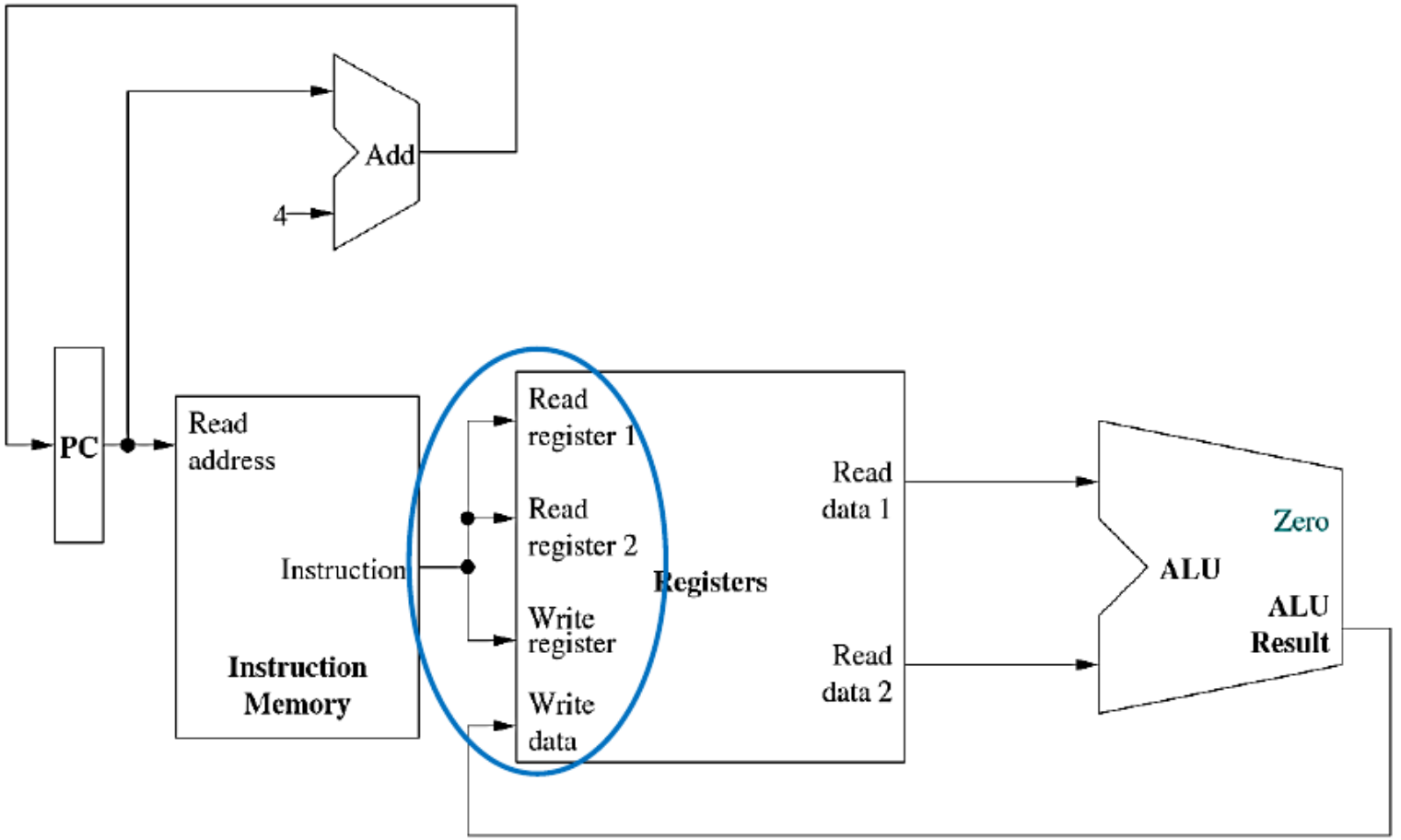
Datapath for R-format Instructions.

2. Fetch instruction from instruction memory.
3. Decode instruction.



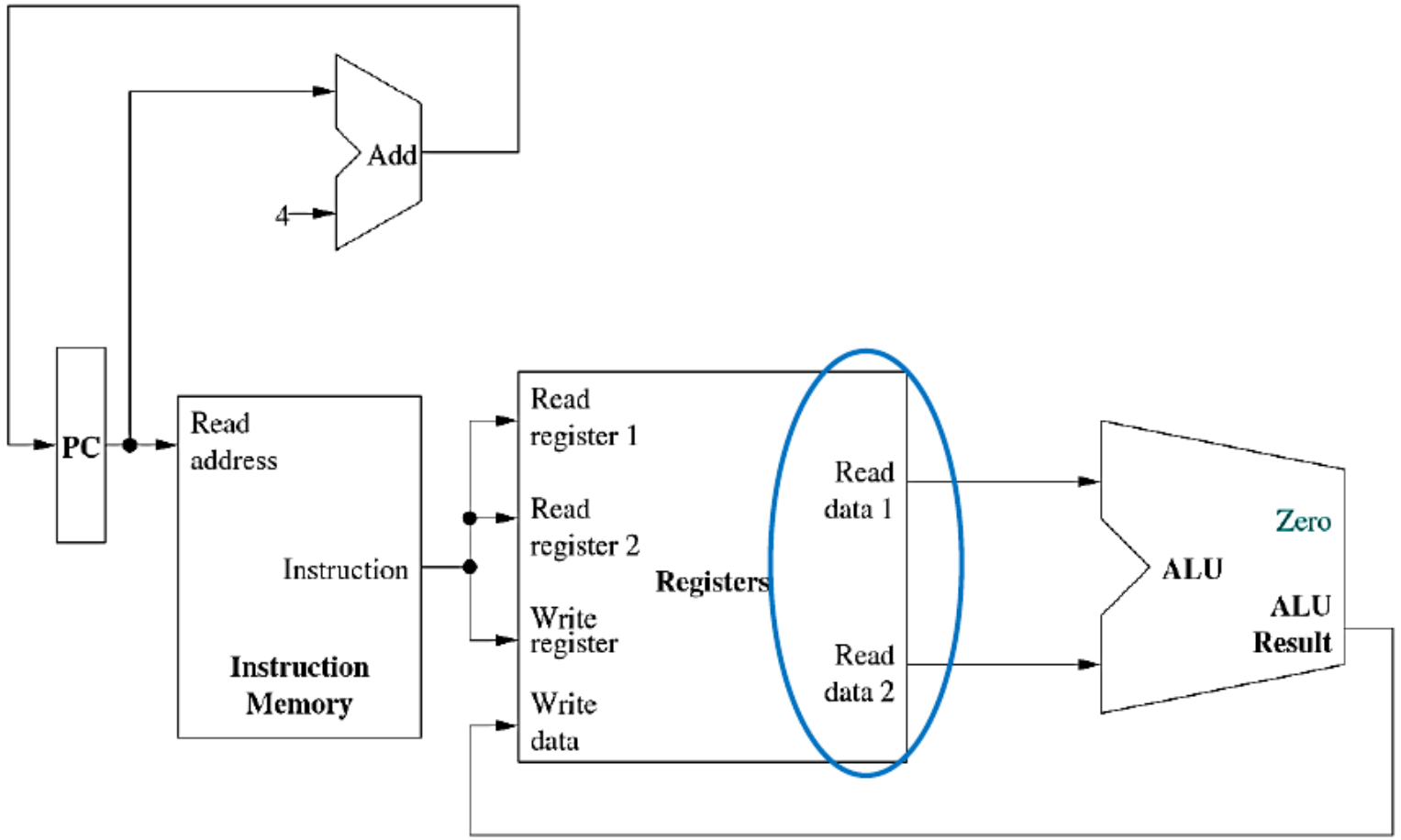
Datapath for R-format Instructions.

4. Pass rs, rt, and rd into read register and write register arguments.



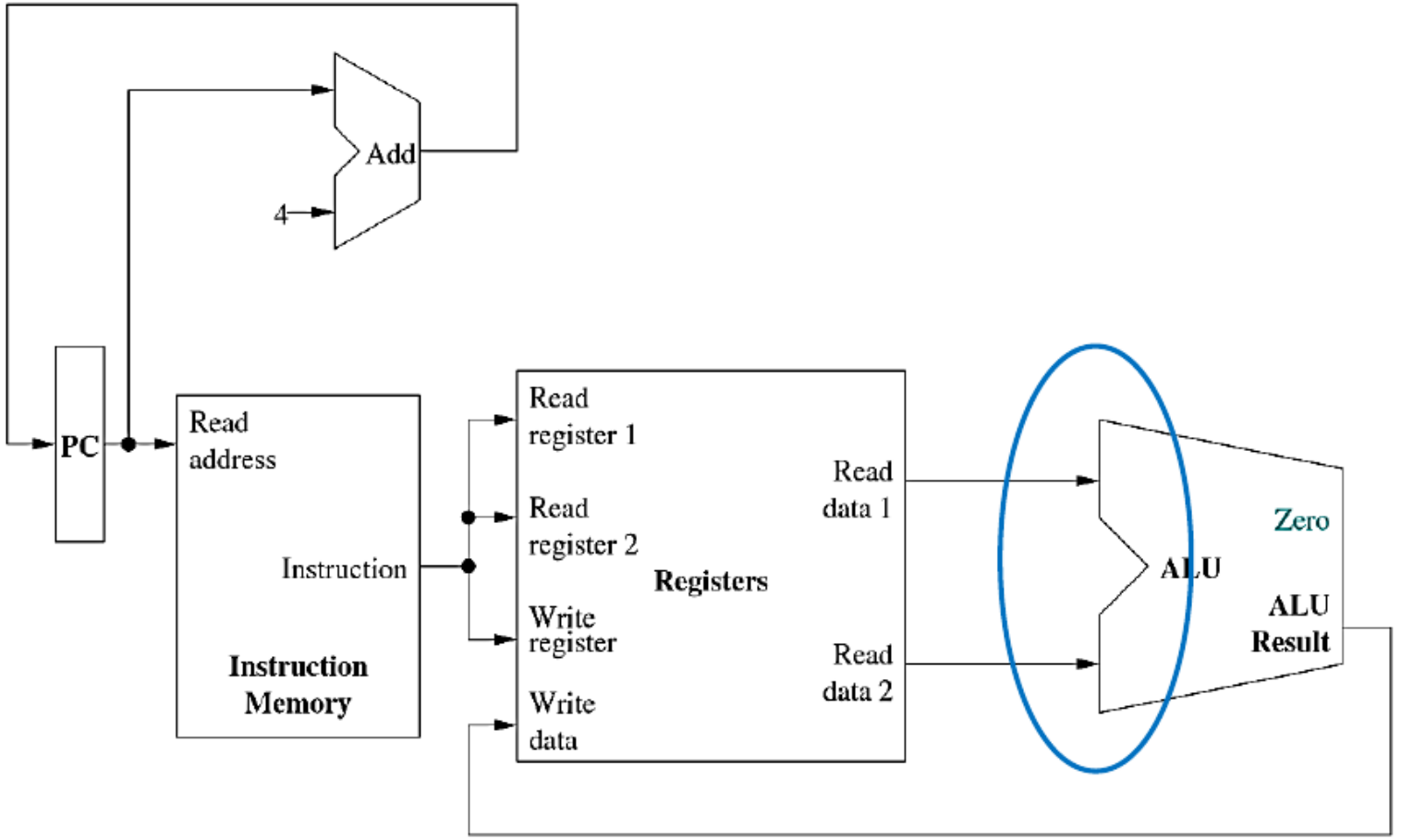
Datapath for R-format Instructions.

5. Retrieve data from read register 1 and read register 2 (rs and rt).



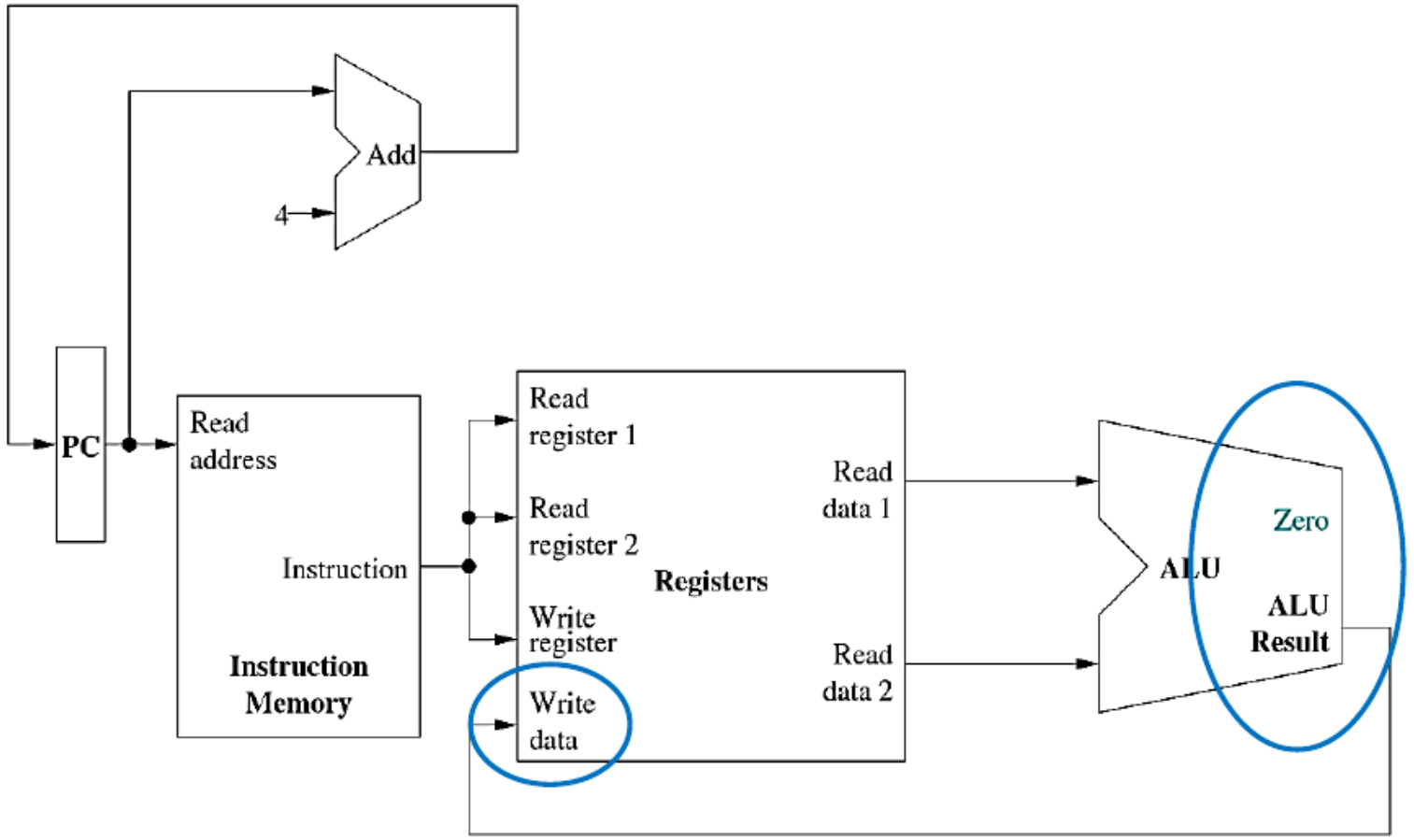
Datapath for R-format Instructions.

6. Pass contents of rs and rt into the ALU as operands of the operation to be performed.



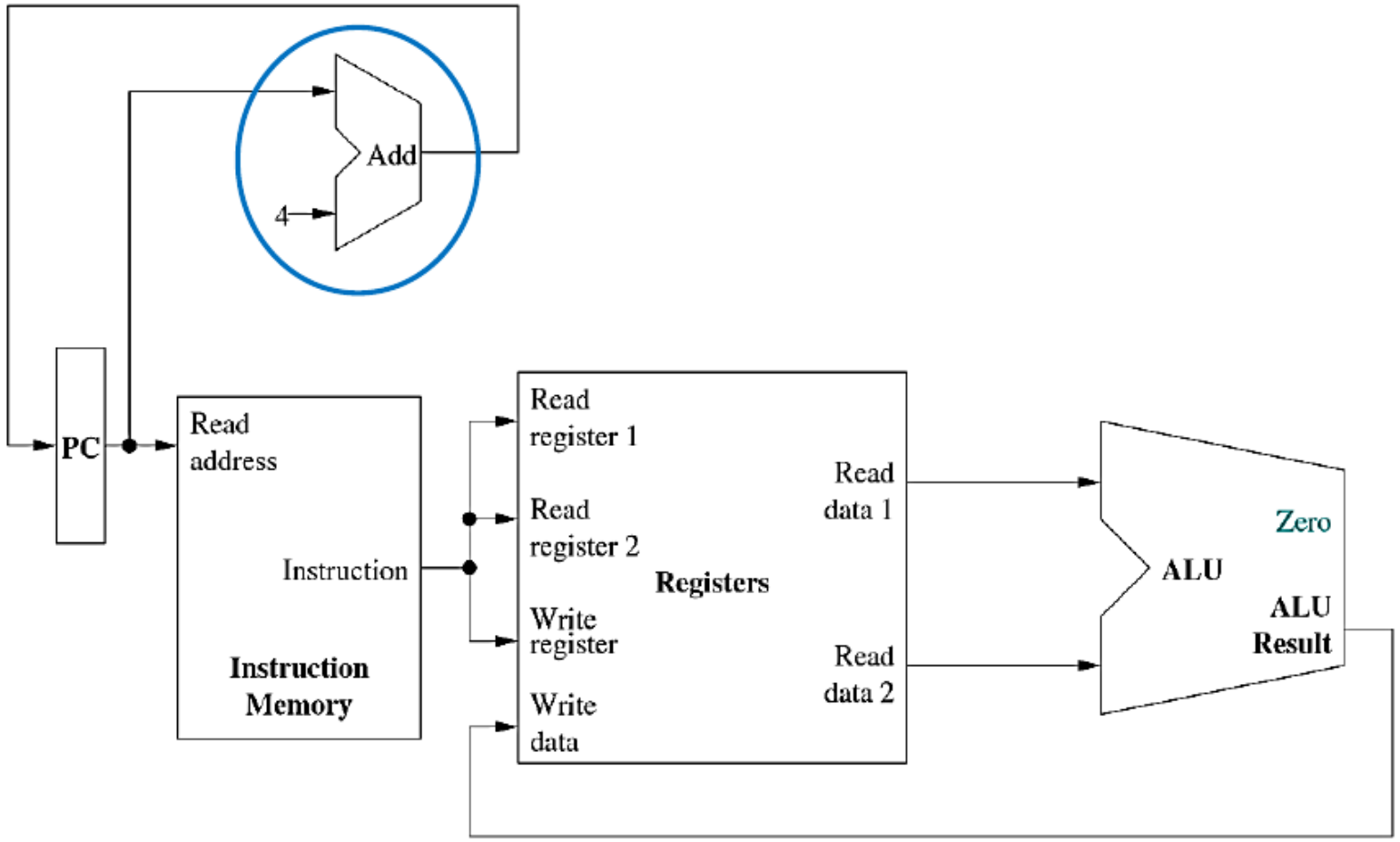
Datapath for R-format Instructions.

7. Retrieve result of operation performed by ALU and pass back as the write data argument of the register file (with the RegWrite bit set).



Datapath for R-format Instructions.

8. Add 4 bytes to the PC value to obtain the word-aligned address of the next instruction.



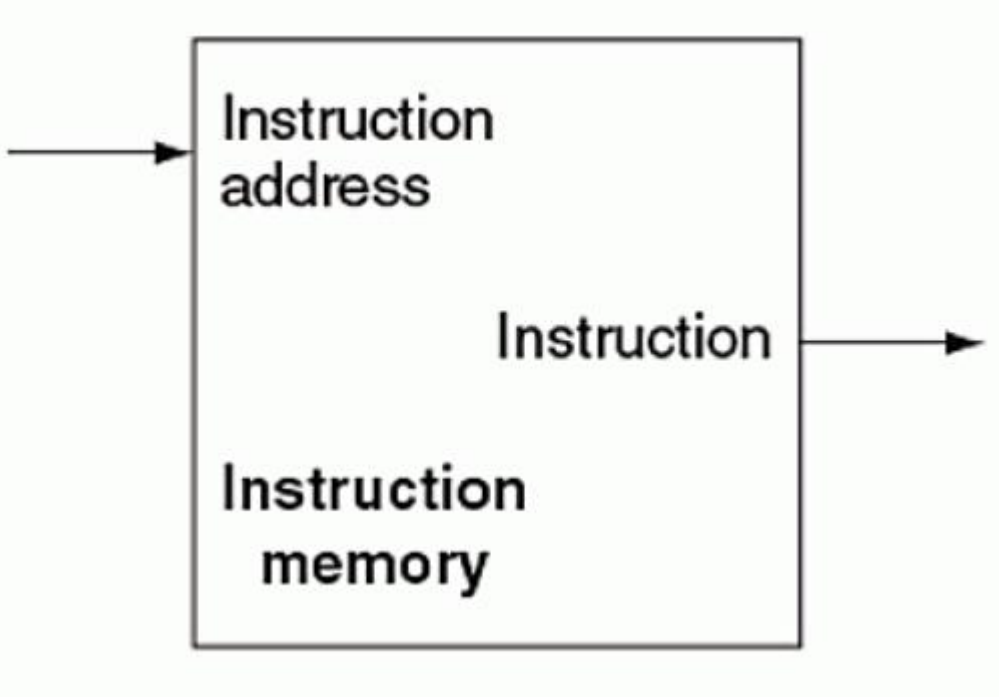
Component list in SCD

Following are the hardware components

1. **Instructio**n Memory
2. **Register File** (Lab 5)
3. **ALU** (Lab 4)
4. **Control Unit and ALU Control** (Lab 4)
5. **Data Memory**
6. **Sign Extender**
7. **Shifter** (Lab 3)
8. **Program Counter**
9. **Adders (required for computing next** PC) (Lab 2)
10. **Multiplexers** (Lab 1)

Instruction Memory

- First, we have instruction memory.
  - Instruction memory is a state element that provides read-access to the instructions of a program and, given an address as input, supplies the corresponding instruction at that address.



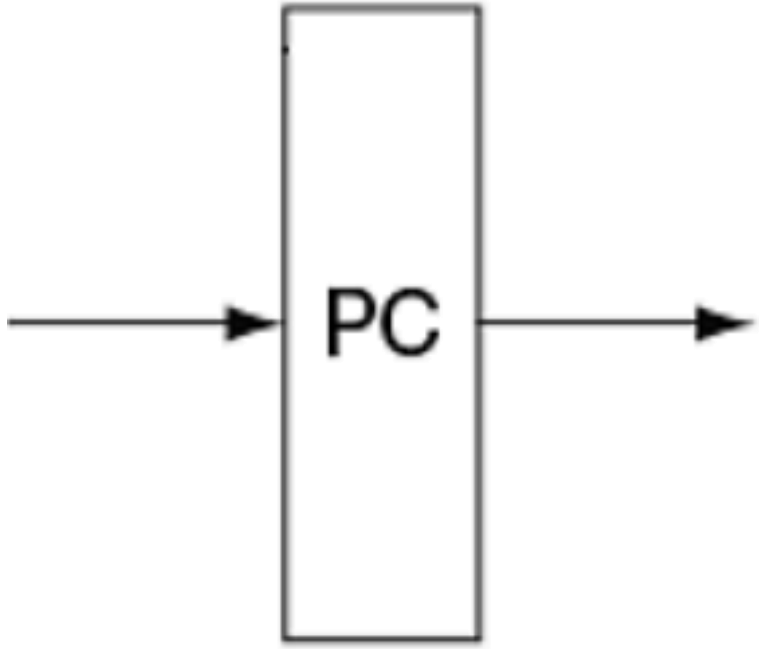
Instruction Memory

```
module Instruction_Memory(Inst, PC, clock);
    input[31:0] PC;
    input clock;
    output[31:0] Inst;
    reg [31:0] memory [0:31];
    reg [31:0] Inst;
  integer addr;
  initial begin
    memory[0] = 32'b00000000000000000000000000000000;  // nop
    memory[1] = 32'b00000000000000000000000000000000;  // nop
    memory[2] = 32'b00000000000000000000000000000000;  // nop
    memory[3] = 32'b10001100000100010000000000001000;  // lw  $s1($17), 8($0)
    memory[4] = 32'b10001100000100100000000000000100;  // lw  $s2($18), 4($0)
    memory[5] = 32'b00000010001100100100000000100000;  // add $t0($8), $s1($17), $s2($18)
```

```
    always @(posedge clock) begin
      addr = PC[31:0];
      Inst = memory[addr/4];
      end
```

Program Counter

- Next, we have the program counter or PC.
  - The PC is a state element that holds the address of the current instruction. Essentially, it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.
    - Normally PC increments sequentially except for branch instructions

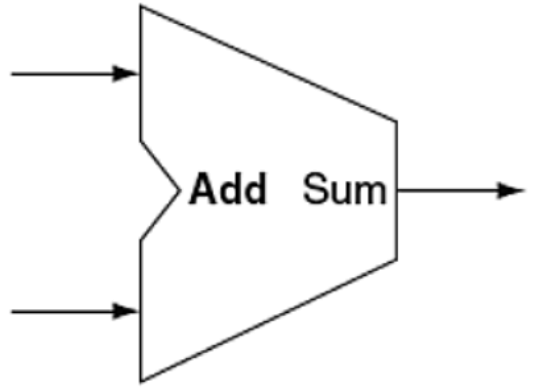


Program Counter

```
module PC(count, clock, reset);
   input       clock, reset;
   output      [31:0]  count;
   reg    [31:0]  count;
   always @(posedge clock)
      if(!reset)
         count = count + 1;
      else
         count = 0;
endmodule
```

FADDER32 (2 modules)

- The adder is responsible for incrementing the PC to hold the address of the next instruction.
- It takes two input values, adds them together and outputs the result.

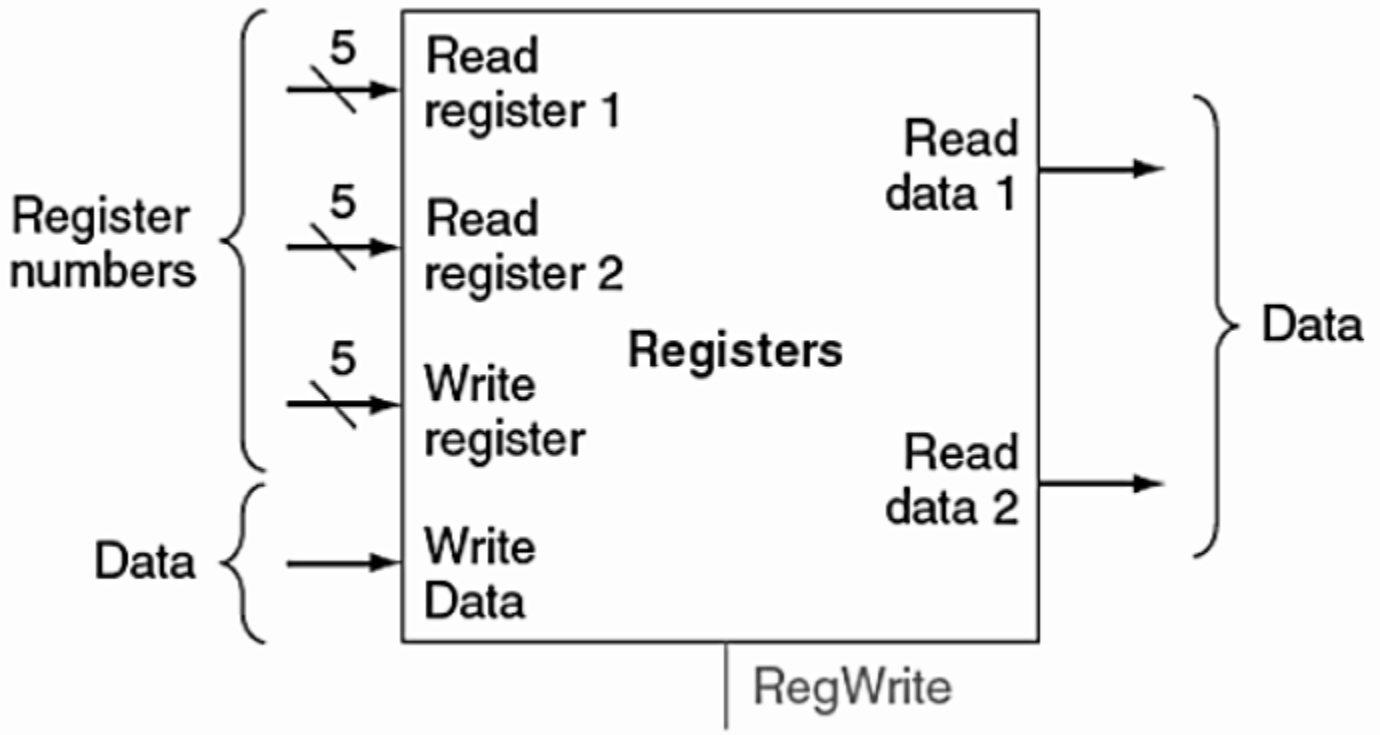



```
module FADDER32(carry, sum, A, B, CarryIn);
  input [31:0] A, B;
  input CarryIn;
  output  [31:0]sum;
  output  carry;
  wire  c1, c2, c3;
  FADDER8  mod1(c1, sum[7:0], A[7:0], B[7:0], CarryIn);
  FADDER8  mod2(c2, sum[15:8], A[15:8], B[15:8], c1);
  FADDER8  mod3(c3, sum[23:16], A[23:16], B[23:16], c2);
  FADDER8  mod4(carry, sum[31:24], A[31:24], B[31:24], c3);
endmodule
```

Register File

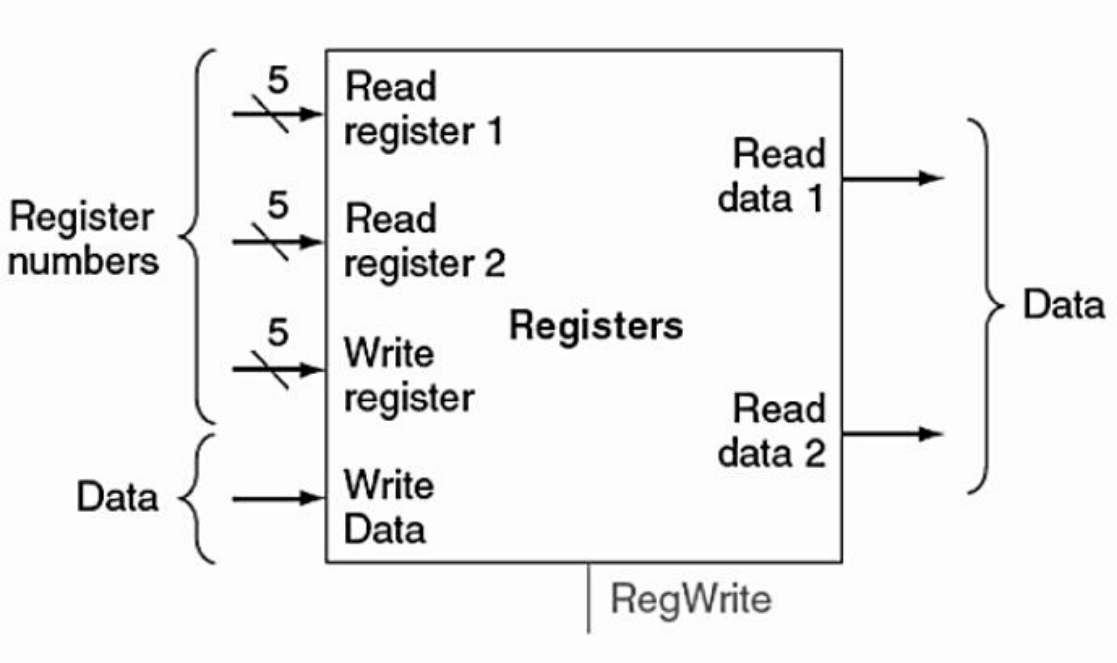
- To support R-format instructions, we'll need to add a state element called a register file. A register file is a collection readable/writeable registers.

- `Read register 1` – first source register. 5 bits wide.
- `Read register 2` – second source register. 5 bits wide.
- `Write register` – destination register. 5 bits wide.
- `Write data` – data to be written to a register. 32 bits wide.



Register File

- At the bottom, we have the RegWrite input. A writing operation only occurs when this bit is set.
- The two output ports are:
  - Read data 1 – contents of source register 1.
  - Read data 2 – contents of source register 2.



Register File

```
module d_ff(q, d, clock, reset);
  input d, clock, reset;
  output  q;
  reg q;
  always @ (posedge clock or negedge reset)
  if(~reset)
    q = 1'b0;
  else
    q = d;
endmodule
```

```
module reg_32bit(q, d, clock, reset);
  input [31:0]  d;
  input clock, reset;
  output  [31:0]  q;
  genvar j;
  generate
    for(j = 0; j < 32; j = j + 1) begin:  d_loop
      d_ff ff(q[j], d[j], clock, reset);
    end
  endgenerate
endmodule
```

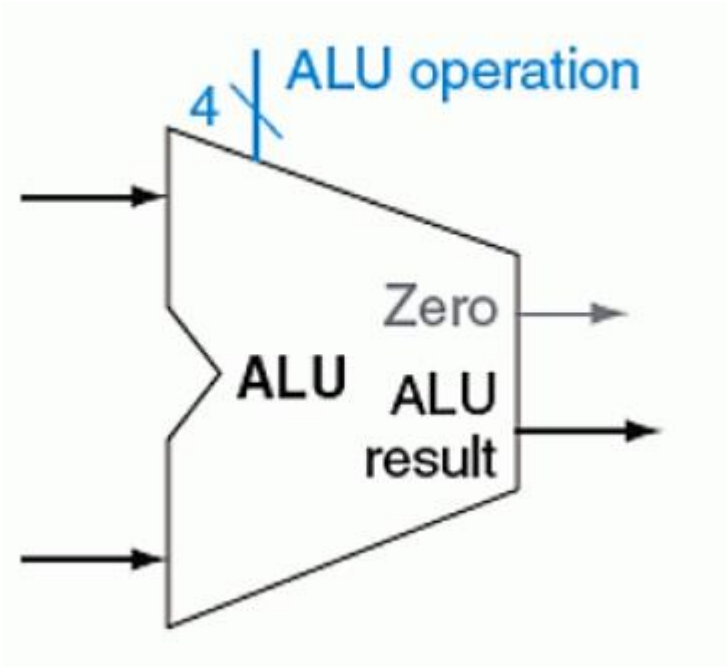
Register File

- Thirty-two registers of 32-bit wide (32 instances)

```
reg_32bit r0(w0, WriteData, c0, Reset);
reg_32bit r1(w1, WriteData, c1, Reset);
reg_32bit r2(w2, WriteData, c2, Reset);
reg_32bit r3(w3, WriteData, c3, Reset);
```

ALU Unit

- To execute R-format instructions, we need to include the ALU element.
  - The ALU performs the operation indicated by the instruction.
  - It takes two operands, as well as a 4-bit wide operation selector value. The result of the operation is the output value.

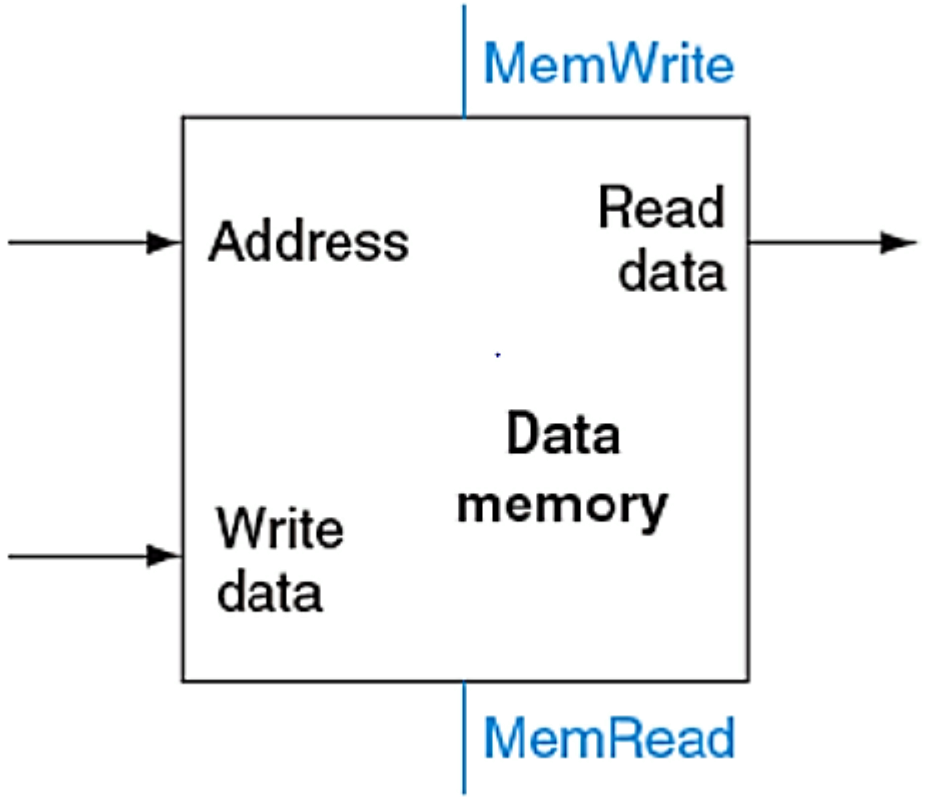


ALU Unit

```
module  ALU32Bit(Zero, CarryOut, Result, A, B, Op);
  input [2:0] Op;
  input [31:0] A, B;
  output [31:0] Result;
  reg [31:0] Result;
  output CarryOut, Zero;
  reg CarryOut;
  assign Zero = (({Result} == 0)) ? 1 : 0;
  always @ (Op, A, B) begin
    case(Op)
      0:  Result <= A & B;
      1:  Result <= A | B;
      2:  {CarryOut, Result[31:0]} <= A + B;
      6:  {CarryOut, Result[31:0]} <= A - B;
      7:  Result <= A < B ? 1 : 0;
      default: Result <= 0;
    endcase
  end
endmodule
```

Data Memory

- There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable.
- The output is the data read from the memory location accessed, if applicable.



Data Memory

- **Input**:- Read address, write address, write data
- **Output**: Read data

```
input MemRead, MemWrite, Clock;
input [31:0] ReadAddress, WriteAddress;
input [31:0] WriteData;
output reg [31:0] ReadData;
reg [31:0] memory [0:31];
integer raddr, waddr;
```

```
initial begin
  memory[0] = 32'b00000000000000000000000000000000;  // nop
memory[1] = 32'b00000000000000000000000001010100;  // Value of 84
memory[2] = 32'b00000000000000000000000000001011;  // Value of 11
```

Data Memory

```
always @(posedge Clock) begin
    raddr = ReadAddress;
    waddr = WriteAddress;
    if(MemRead)
        ReadData = memory[raddr/4];
    else if(MemWrite)
        memory[waddr/4] = WriteData;
end
```

Multiplexers

- Multiplexers 32bit-2x1 (4 modules)
  - Generating the new address after the branch MUX
  - Generating the new address after the Jump MUX
  - Writing back to register file
  - Selecting the second ALU Source
- Multiplexers 5bit-2x1 (1 module)
  - Selecting the writing register

Multiplexers 5bit-2x1

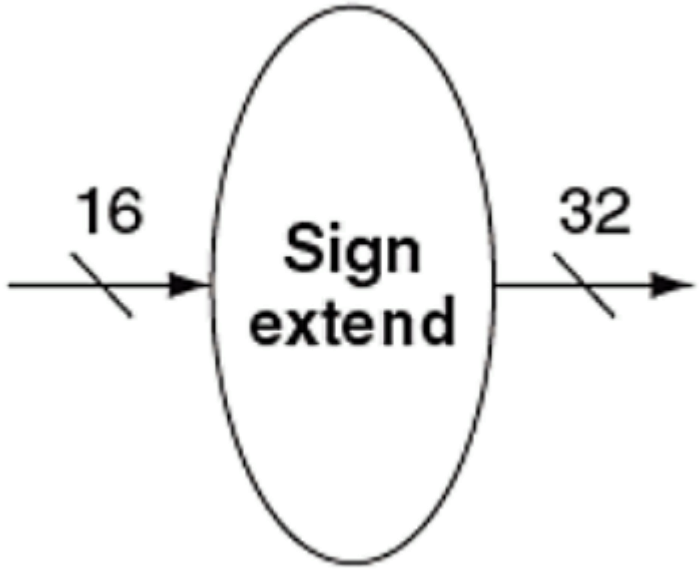
```
module MUX5Bit_2To1(out, select, q1, q2);
    input  [4:0] q1, q2;
    input select;
    output [4:0] out;
    genvar  j;
  generate  for(j = 0; j < 5; j = j + 1)
    begin:  mux_loop
      Mux2To1 Mux(out[j], select, q1[j], q2[j]);
    end
  endgenerate
endmodule
```

Multiplexers 32bit-2x1

```
module  Mux32Bit_2To1(out, select, in1, in2);
  input [31:0] in1, in2;
  input select;
  output  [31:0] out;
  Mux8Bit_2To1_generate Mux1(out[7:0], select, in1[7:0], in2[7:0]);
  Mux8Bit_2To1_generate Mux2(out[15:8], select, in1[15:8], in2[15:8]);
  Mux8Bit_2To1_generate Mux3(out[23:16], select, in1[23:16], in2[23:16]);
  Mux8Bit_2To1_generate Mux4(out[31:24], select, in1[31:24], in2[31:24]);
endmodule
```

Sign Extender

- To perform sign-extending, we can add a sign extension element.
  - The sign extension element takes as input a 16-bit wide value to be extended to 32-bits.
  - To sign extend, we simply replicate the most-significant bit of the original field until we have reached the desired field width.



Sign Extender

- Sign Extend the Branch Offset

```
module Sign_Extender(out, in);
    input [15:0] in;
    output [31:0] out;
    assign out = { {16{in[15]}}, in};
endmodule
```

Shifter (2 Modules)

- Shift left by 2 bits the Sign Extended Branch Offset
- Shift left by 2 bits the Jump Offset.

```
module Shift_Left(out, in);
    input [31:0] in;
    output [31:0] out;
    assign out = {in[29:0],1'b0,1'b0};
endmodule
```

Concatenate PC and Jump Offset

- Concatenate PC and Left Shifted Jump Offset

```
module concatJuPC(out, J, PC);
  input [31:0] J, PC;
  output [31:0] out;
  assign {out} = {{PC[31:28]}, {J[27:0]}};
endmodule
```

ALU Control Unit

```
module  ALUControlUnit(Op, Func, ALUOp);
   input [5:0] Func;
   input [1:0] ALUOp;
   output  [2:0] Op;
   assign  Op[0] = ALUOp[1] & (Func[3] | Func[0]);
   assign  Op[1] = (~ALUOp[1]) | (~Func[2]);
   assign  Op[2] = ALUOp[0] | (ALUOp[1] & Func[1]);
endmodule
```

Main Control Unit

- Generate the Control Signals

```
output  RegDst, Jump, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite,
        Branch, ALUOp0, ALUOp1;
input [5:0] Op;
wire  RFormat, LW, SW, BEQ, J;
assign  RFormat = (~Op[5])&(~Op[4])&(~Op[3])&(~Op[2])&(~Op[1])&(~Op[0]);
assign  LW = (Op[5])&(~Op[4])&(~Op[3])&(~Op[2])&(Op[1])&(Op[0]);
assign  SW = (Op[5])&(~Op[4])&(Op[3])&(~Op[2])&(Op[1])&(Op[0]);
assign  BEQ = (~Op[5])&(~Op[4])&(~Op[3])&(Op[2])&(~Op[1])&(~Op[0]);
assign  J = (~Op[5])&(~Op[4])&(~Op[3])&(~Op[2])&(Op[1])&(~Op[0]);
```

Final Task

- Integrate all Modules.
- Test on given testbench.

Thank You

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

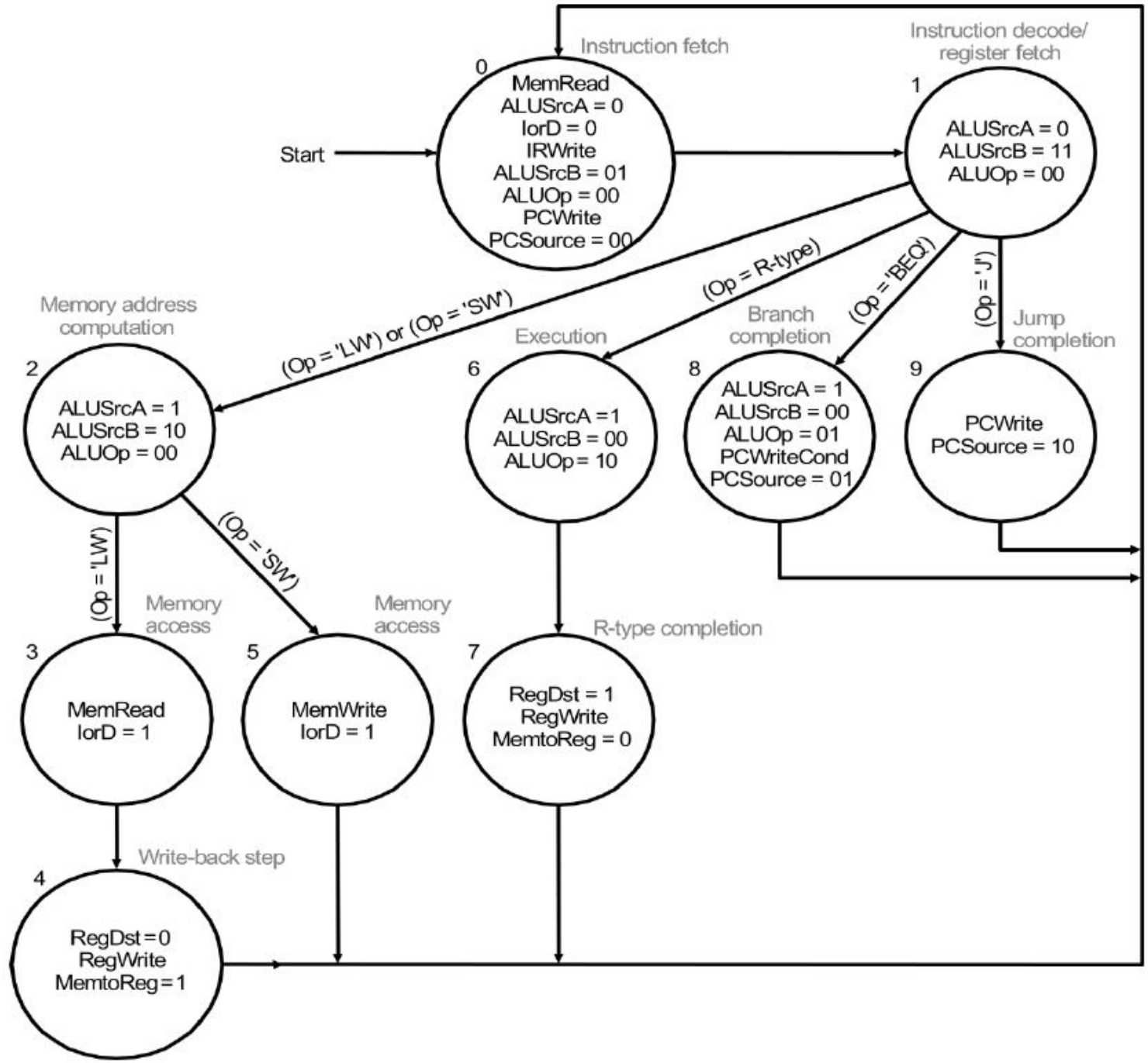
• Lab – 7

- **Main controller for MIPS multicycle implementation.**

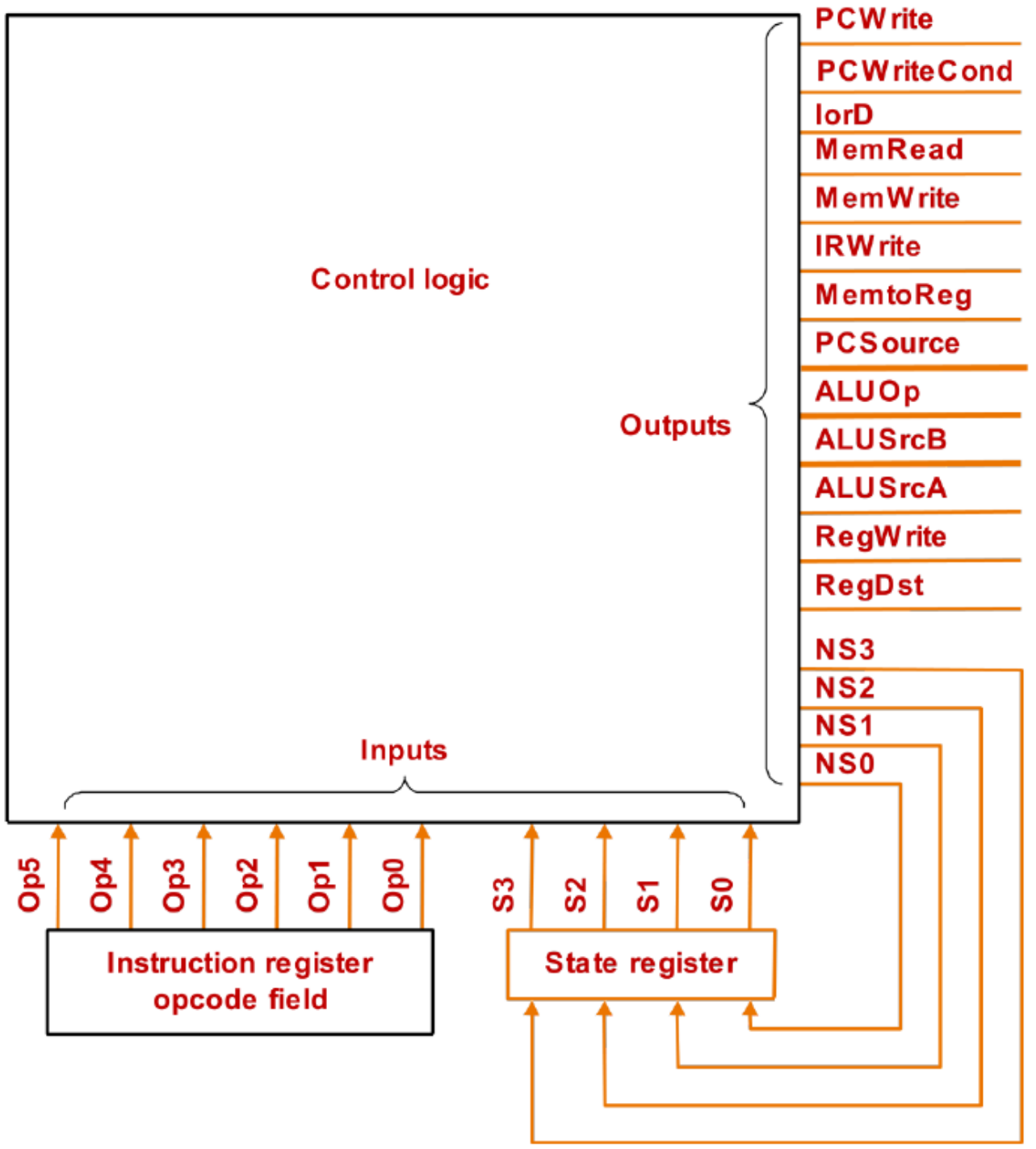
Steps in Multicycle Datapath

- We already observed that our instructions can be roughly broken up into the following steps:
  1. Instruction fetch
  2. Instruction decode and register fetch
  3. Execution, memory address computation, branch completion, or jump completion.
  4. Memory access or R-type instruction completion.
  5. Memory read completion (WriteBack).
- Instructions take 3-5 of the steps to complete. The first two are performed identically in all instructions.

Controller FSM



PLA implementation



Controller Output Signals

1-Bit Signal Name	Effect When Deasserted	Effect When Asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	Write register is written with the value of the Write data input.
ALUSrcA	The first ALU operand is PC.	The first ALU operand is A register.
MemRead	None	Content of memory at the location specified by the Address input is put on the Memory data output.
MemWrite	None	Memory contents of the location specified by the Address input is replaced by the value on the Write data input.

Controller Output Signals

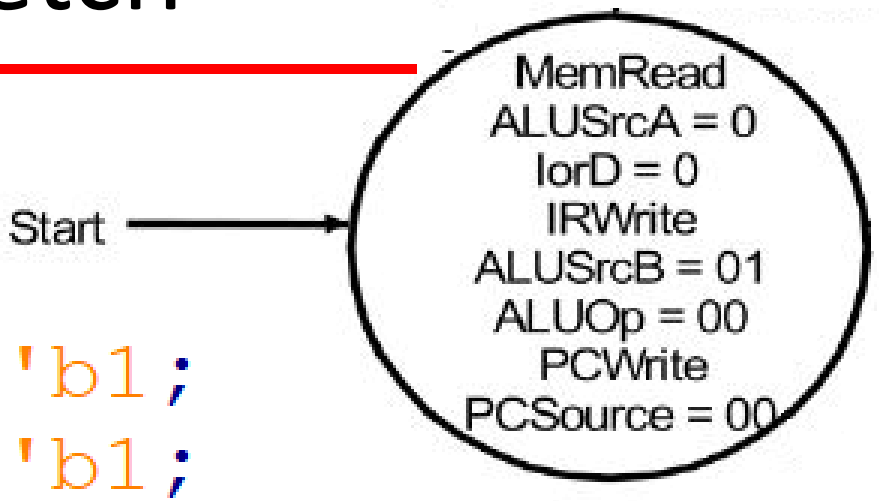
1-Bit Signal Name	Effect When Deasserted	Effect When Asserted
MemToReg	The value fed to the register file input is ALUout.	The value fed to the register file input comes from Memory data register.
IorD	The PC supplies the Address to the Memory element.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the Instruction Register (IR).
PCWrite	None	The PC is written; the source is controlled by PC-Source.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

Controller Output Signals

2-bit Signal	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the operation.
ALUSrcB	00	The second input to ALU comes from the B register.
	01	The second input to ALU is 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the Instruction Register (IR).
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left by 2 bits.
PCSource	00	Output of the ALU (PC+4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25-0] shifted left 2 bits and concatenated with PC + 4[31-28]) is sent to the PC for writing.

1-Instruction Fetch

Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	00
ALUSrcB	01
ALUSrcA	0
RegWrite	0



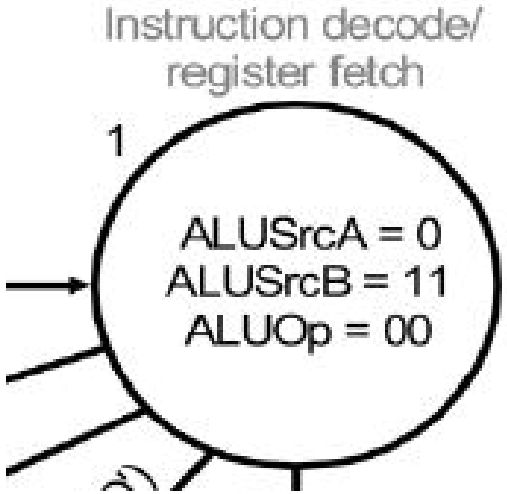

```
FETCH:
   begin
      MemRead = 1'b1;
      IRWrite = 1'b1;
      ALUSrcB = 2'b01;
      PCWrite = 1'b1;
   end

    case (state)
    FETCH:  nextstate = DECODE;
```

2-Instruction Decode and Register Fetch

Signal	Value
ALUOp	00
ALUSrcB	11
ALUSrcA	0

```
DECODE:
ALUSrcB = 2'b11;
```

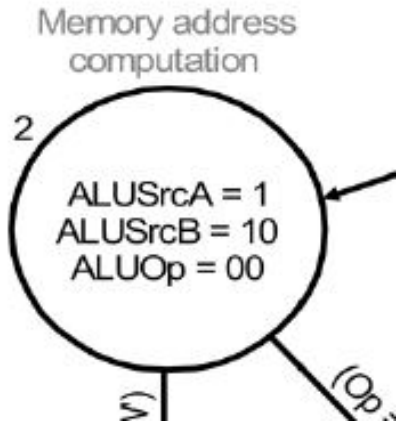



```
DECODE:   case(Op)
            //OpCode
          6'b100011:   nextstate = MEMADRCOMP;//lw
          6'b101011:   nextstate = MEMADRCOMP;//sw
          6'b000000:   nextstate = EXECUTION;//r
          6'b000100:   nextstate = BEQ;//beq
          default: nextstate = FETCH;
        endcase
```

3-Execution: Memory Reference

Signal	Value
ALUOp	00
ALUSrcB	10
ALUSrcA	1

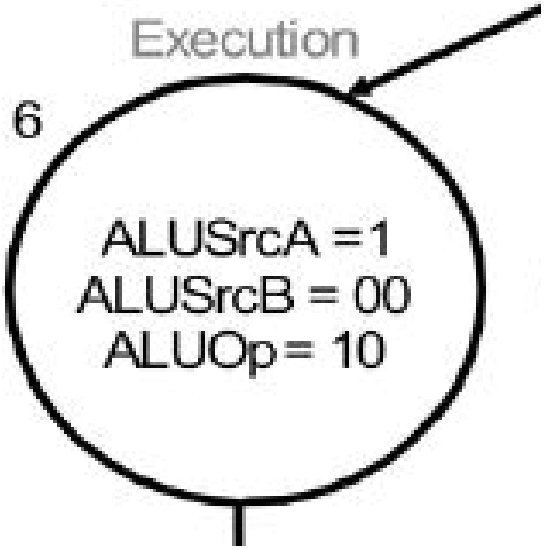
```
MEMADRCOMP:
   begin
      ALUSrcA = 1'b1;
      ALUSrcB = 2'b10;
   end
```




```
MEMADRCOMP:   case(Op)
                 6'b100011:      nextstate = MEMACCESSL;//lw
                 6'b101011:      nextstate = MEMACCESSS;//sw
                 default: nextstate = FETCH;
```

3-Execution: Arithmetic/Logical Op

Signal	Value
ALUOp	10
ALUSrcB	00
ALUSrcA	1

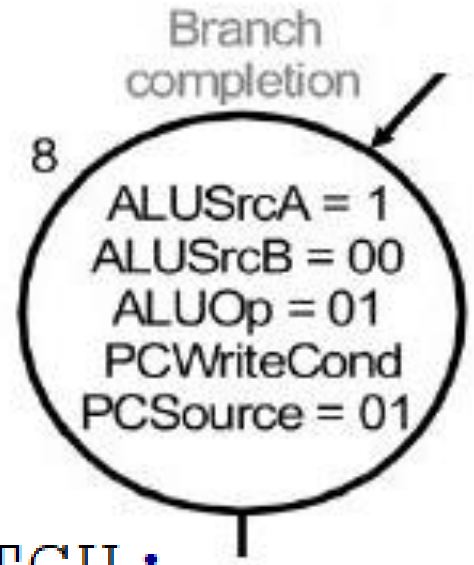



```
EXECUTION: nextstate = RTYPEEND;

EXECUTION:
   begin
      ALUSrcA = 1'b1;
      ALUOp   = 2'b10;
   end
```

3-Execution: Branch

Signal	Value
ALUOp	01
ALUSrcB	00
ALUSrcA	1
PCSource	01
PCWriteCond	1

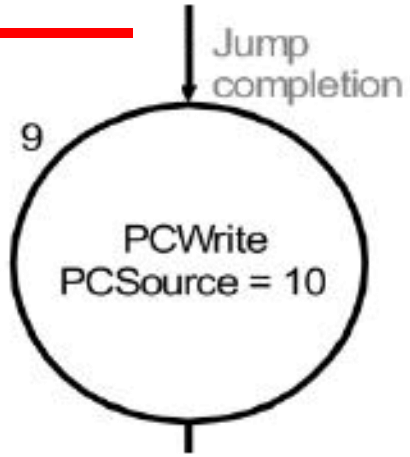



```
BEQ:    nextstate = FETCH;
```

```
BEQ:
   begin
      ALUSrcA = 1'b1;
      ALUOp   = 2'b01;
      PCWriteCond = 1'b1;
      PCSource = 2'b01;
   end
```

3-Execution: Jump

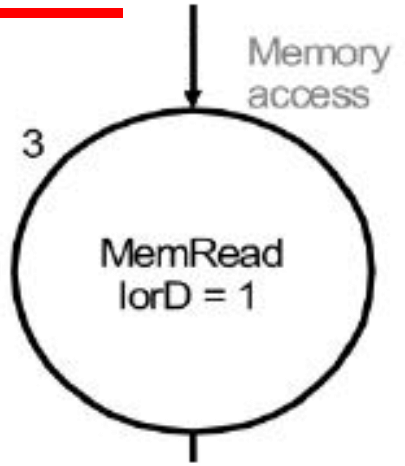
Signal	Value
PCSource	10
PCWrite	1




```
JUMP:    nextstate = FETCH;
JUMP:
   begin
      PCWrite = 1'b1;
      PCSource = 2'b10;
   end
```

4-Memory Access: Load

Signal	Value
MemRead	1
IorD	1
IRWrite	0

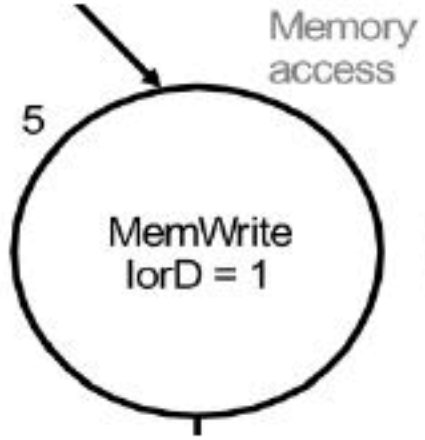



```
MEMACCESSL:     nextstate = MEMREADEND;

MEMACCESSL:
   begin
      MemRead = 1'b1;
      IorD    = 1'b1;
   end
```

4-Memory Access: Store

Signal	Value
MemWrite	1
IorD	1

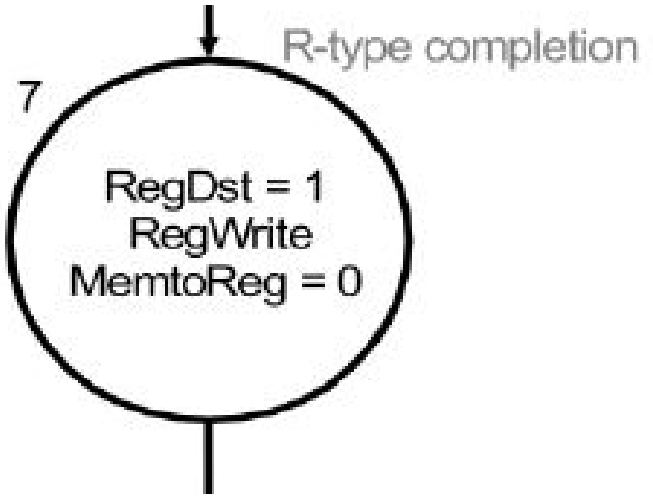



```
MEMACCESSS:     nextstate = FETCH;
```

```
MEMACCESSS:
   begin
      MemWrite = 1'b1;
      IorD     = 1'b1;
   end
```

4-R-Type Completion

Signal	Value
MemtoReg	0
RegWrite	1
RegDst	1

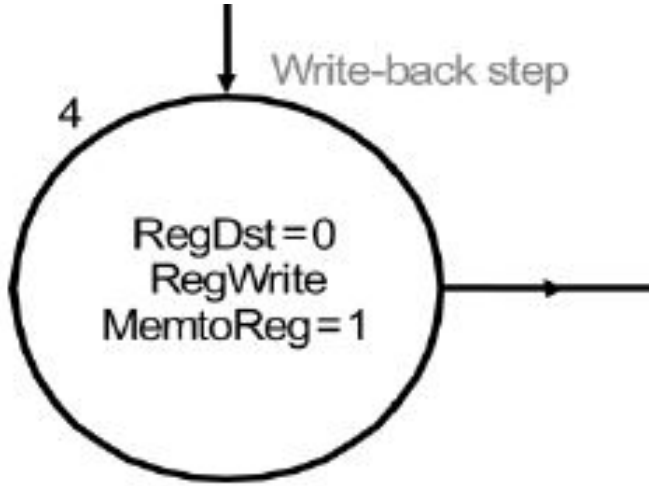



```
RTYPEEND: nextstate = FETCH;
```

```
RTYPEEND:
   begin
      RegDst   = 1'b1;
      RegWrite = 1'b1;
   end
```

5-Read Completion

Signal	Value
RegWrite	1
MemtoReg	1
RegDst	0




```
MEMREADEND:     nextstate = FETCH;
MEMREADEND:
   begin
      RegWrite = 1'b1;
      MemtoReg = 1'b1;
      RegDst = 1'b0;
   end
```

Working of FSM

- Start from Fetch

```
default: nextstate = FETCH;
```

- Three always blocks.
  - For state or opcode change response.
  - For state change response.
  - For clock response.

For State or Opcode Change

```
always@(state or Op) begin
    case (state)
    FETCH:  nextstate = DECODE;
    DECODE:   case(Op)
                 //OpCode

                default: nextstate = FETCH;
              endcase
    MEMADRCOMP:   case(Op)

                default: nextstate = FETCH;
              endcase
                                    MEMACCESSL:     nextstate = MEMREADEND;
                                    MEMREADEND:     nextstate = FETCH;
                                    MEMACCESSS:     nextstate = FETCH;
                                    EXECUTION: nextstate = RTYPEEND;
                                    RTYPEEND: nextstate = FETCH;
                                    BEQ:    nextstate = FETCH;
                                    JUMP:    nextstate = FETCH;
                                    default: nextstate = FETCH;
                                  endcase
                                end
```

For State Change

```
always@(state) begin

    case (state)
    FETCH:

    DECODE:

    MEMADRCOMP:

    MEMACCESSL:

    MEMREADEND:

    MEMACCESSS:

    EXECUTION:

    RTYPEEND:

    BEQ:

    JUMP:

    endcase
end
```

For clock response

```
always@(posedge clk)
   if (reset)
        state <= FETCH;
   else
        state <= nextstate;
```

Final Task

1. Create Module controller.
2. Declare ports and parameters.
3. Initialize output ports.
4. Include specify always blocks.
5. Create testbench.

Thank You

innovate achieve lead

BITS Pilani

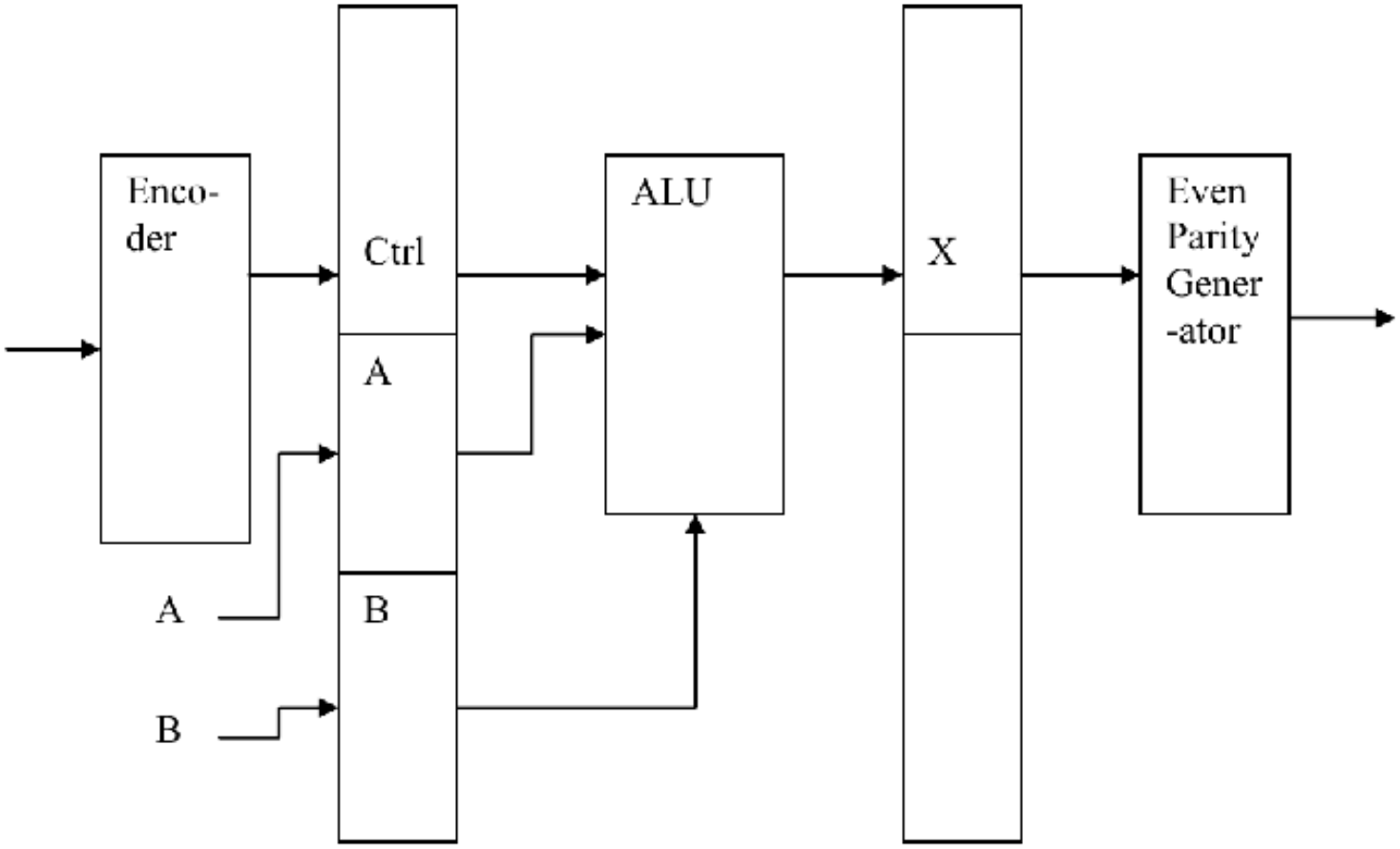
Pilani|Dubai|Goa|Hyderabad

• Lab – 8

- **Modeling 3-stage pipeline design.**

3-Stage Pipeline

- Today we have to implement a simple 3-stage pipeline design.



3-Stage Pipeline

- To accomplish this,
  1. First identify the hardware components of the pipeline design.
  2. Write separate modules for each component.
  3. Integrate these modules to realize the complete pipeline design circuit.
  4. Test the correctness of the complete implementation by supplying a sequence of instructions as input.
- It consist of three pipeline stages: fetch, execute and generate parity and two pipeline registers.

Fetch Stage (F)

- In first stage (Fetch stage (F)), following operations are done:

1. 8-bit function code of the operation to be performed in the instruction is given as input to the encoder, encoder encodes the function code and outputs the corresponding 3-bit op-code. This op-code is stored in the first pipeline register.
2. Two 4-bit operands A and B are stored in first pipeline register which will be used in second stage.

Fetch Stage (F)

- Let us consider, there are 8 instructions in ISA each require 3-bit opcode. The encoder takes 8-bit function code of the instruction as input and produces a 3-bit op-code (or Ctrl) as output.

1. ctrl = 3'b000 : add
2. ctrl = 3'b001 : sub
3. ctrl = 3'b010 : xor
4. ctrl = 3'b011 : or
5. ctrl = 3'b100 : and
6. ctrl = 3'b101 : nor
7. ctrl = 3'b110 : nand
8. ctrl = 3'b111 : xnor

```
module ENCODER(OpCode, FuncCode);
  input [7:0] FuncCode;
  output  [2:0] OpCode;
  reg [2:0] OpCode;
  always  @ (FuncCode)  begin
    if  (FuncCode[7])
      OpCode = 3'd0;
    ......
    ......
    ......
  end
endmodule
```

Fetch Stage (F)

```
module FIRSTPIPE(clock, OpCode, A, B, OpOut, AOut, BOut);
  input [2:0] OpCode;
    ......
    ......
    ......
  always  @ (posedge clock) begin
    OpOut <= OpCode;
    AOut  <= A;
    BOut  <= B;
  end
endmodule
```

Execute Stage (E)

- In second stage (Execute stage (E)), following operations are carried out:

1. Op-code, operand A and operand B are read from the first pipeline register and given as input to the ALU.
2. Based on the op-code, ALU performs operation on two operands A and B.
3. Output of ALU is stored in second pipeline register.

Execute Stage (E)

```
module ALU(Carry, X, A, B, OpCode);
  input [2:0] OpCode;
  input [3:0] A, B;
  output  [3:0] X;
  output  Carry;
  assign  {Carry, X} = (OpCode == 3'b000) ? (A + B) :
  ......
  ......
  ......
  (OpCode == 3'b110) ? (~(A & B)) : (~(A ^ B));
endmodule
```

Execute Stage (E)

```
module SECONDPIPE(clock, X, XOut);
  input clock;
    ......
    ......
    ......
  always  @ (posedge clock) begin
    XOut = X;
  end
endmodule
```

Generate Parity Stage (GP)

- In third stage (Generate Parity (GP)), following operations are done:

1. Output of ALU X is read from second pipeline register and given as input to Even Parity Generator.
2. Even parity generator generates parity of the input 4-bit number.

Generate Parity Stage (GP)

```
module PARITYGENERATOR(X, Out);
  input [3:0] X;
  output  Out;
  assign  Out = ~(X[0] ^ X[1] ^ X[2] ^ X[3]);
endmodule
```

Final Task : Integrate Modules

```
module PIPELINE(clock, FuncCode, A, B, Out);
  input clock;
    ......
    ......
    ......
  ENCODER mod1(OpCode, FuncCode);
  FIRSTPIPE mod2(clock, OpCode, A, B, OpOut, AOut, BOut);
  ALU mod3(Carry, X, AOut, BOut, OpOut);
  SECONDPIPE  mod4(clock, X, XOut);
  PARITYGENERATOR mod5(XOut, Out);
endmodule
```

Validate Design

```
module TESTBENCH;
  reg clock;
    ......
    ......
    ......
  PIPELINE  mod(clock, FuncCode, A, B, Out);
  initial begin
    $monitor($time, " A = %b,  .....", A,.....);
    #0  clock = 1'b1;
    #4  A = 4'b0101; B = 4'b1110;  FuncCode = 8'b10000000;
    ......
    ......
    ......
    #20 FuncCode = 8'b00000001;
    #50 $finish;
  end
  always
    #2  clock = ~clock;
endmodule
```

Thank You