Project 1 Pointers

CSE490-590 Spring 2024

Source: From Prior semesters of CSE490/590 offerings

Project 1

Design an 8-bit microprocessor using Verilog HDL by using Structural Verilog modelling.

The individual components can be designed using **behavioral modelling**

Brief Overview

- Verilog control flow keywords are `always @(...)`, `begin/end`, `case/endcase`, `initial`, `if/else`, and `assign`
- Verilog datatypes can be declared as `input/output`, `reg/wire`, array with `[MSB:LSB]`
- Modules in Verilog can be created by declaring a module with a port list of inputs and outputs.
- Behavioral Verilog is used to describe a module at a high level (think of writing a program in C).
- Structural Verilog is used to describe a module at a low level by using primitive types (and/or/not...) and can be used to connect modules together.
- The datapath of a MIPS CPU is described in detail here.

Verilog Resources

- A verilog guide is given in Class webpage (https://cse.buffalo.edu/~rsridhar/cse490-590/lec/verilog_manual.pdf)

- "Digital Systems Design Using Verilog" by Charles Roth , Lizy K. John, Byeong Kil Lee

- IEEE Standard Verilog Hardware Description Language (Manual for Verilog) (https://ieeexplore-ieee-org.gate.lib.buffalo.edu/stamp/stamp.jsp?tp=&arnumber=954909)

- Piazza/ Office hours

Important Verilog Keywords/ Concepts / Usages

- always @
- assign
- case
- reg
- wire
- initial
- if else
- arrays
- Blocking vs non-blocking
- Module instantiations
- Port connections

Control Flow: `always@`

If there is any change in the sensitivity list (event) the code inside the block will be executed. Code inside an always block is executed sequentially.

Syntax:

```
always@(<sensitivity list>)
begin

// code to be executed

end
```

```
module always_eg(input[2:0] any_input);

always@(any_input)
begin
    // some codes...
end
endmodule
```

Control Flow: `assign`

An `assign` statement can drive wires or other similar wire data-types *continuously* with a value. However, the `assign` statement can't assign value to `reg` data-types.

Syntax:

`assign <signal> = <expression of different signals or constant value>`

```
module assign_eg(
input[2:0] any_input,
wire [2:0] output_1
);

wire[2:0] signal = 101;
assign output_1 = signal;

endmodule
```

```
module assign_eg(
input[2:0] any_input,
wire [2:0] output_1
);


assign output_1 = 100;

endmodule
```

Control Flow: `case`

The `case` statement is similar to the `if-else` statement; it will check if the value of the expression matches one of the case items in the list. The statement in the matched expression will be executed.

Syntax:

```
case(<expression>)
  <case item1>: <statement>
  <case item2>: <statement>
  default: <statement>
endcase
```

```
reg[2:0] reg_a;

always@(any_input)
begin
case(any_input)
    2'b001: reg_a = 2'b001;
    2'b010: reg_a = 2'b010;
    default: reg_a = 0;
endcase
end
```

Control Flow: `initial`

There are 2 types of procedural blocks: `always` and `initial`. Statements in both these types of blocks are executed sequentially. However, unlike the always block, statements in the initial block will only execute once.

The initial block will be executed sequentially.
Eventually reg_a = 3'b001

```
reg[3:0] reg_a;
initial
begin
    // some instructions..
    reg_a = 3'b101;
    reg_a = 3'b011;
    reg_a = 3'b001;
    // some instructions..
end

assign output_1 = reg_a;
```

Control Flow: `if-else`

Similar to other programing languages.

Syntax:

```
if (<conditional statement>) begin
        ……;
end
else if (<conditional statement>) begin
        ……;
end
else begin
        ……;
end
```

```
module example(input_value, output_value);
    input input_value;
    output output_value;

    reg output_value;
    always @(input_value) begin
        if (input_value == 1) begin
            output_value = 1;
        end
        else if (input_value == 2) begin
            output_value = 2;
        end
        else begin
            output_value = 10;
        end
    end
endmodule
```

Data Types: `reg`

`reg` is the most common variable data type, it stores information.

A `reg` type cannot be used for inputs to a module

```
module reg_wire_eg(
input[2:0] any_input,
input reg[2:0] output_1
);
```

Error: non-net port 'output_1' cannot be of mode input

A `reg` type can be used for the outputs of module

```
module reg_wire_eg(
input[2:0] any_input,
output reg[2:0] output_1
);
```

a `reg` type on the left-hand side (LHS) of an assignment is *only* legal in `always` and `initial` blocks:

```
reg[2:0] reg_a;
always@(any_input)
begin
reg_a = 2'b011;
end
```

```
reg[2:0] reg_a;
reg_a = 2'b011;
```

Error: Syntax error near "=".
Error: 'reg_a' is an unknown type

Data Types: `reg` (Continued)

a `reg` type on the LHS not legal in an `assign` statement (continuous assignment):

```
reg[2:0] reg_a;
assign reg_a = 2'b011;
```

Error: concurrent assignment to a non-net 'reg_a' is not permitted

a reg type on the Right Hand Side (RHS) is legal in an `assign` statement

```
module reg_wire_eg(
input[2:0] any_input,
wire[2:0] output_1
);

reg[2:0] reg_a;
assign output_1 = reg_a;
```

Data Types: `wire`

`wire` can used to connect the input ports and output ports between different modules.

`wire` can be used with both the inputs and outputs of a module

```
module reg_wire_eg(
input wire[2:0] any_input,
output wire[3:0] output_1
);
```

`wire` is the only legal type on the LHS in an `assign` statement

```
module reg_wire_eg(
input wire[2:0] any_input,
output wire[3:0] output_1
);

reg[3:0] reg_a;
initial
begin
    reg_a = 3'b101;
end

assign output_1 = reg_a;

endmodule
```

Data Types: `wire` (Continued)

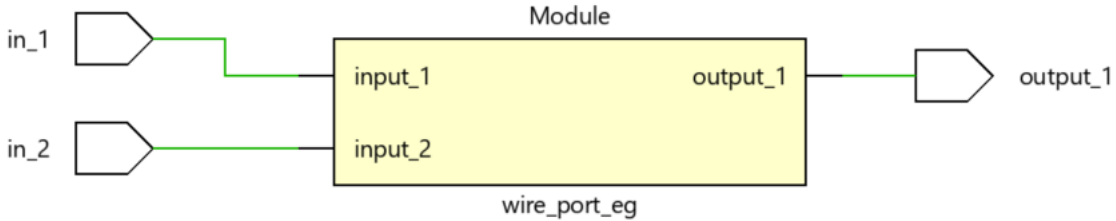
`wire` is stateless and doesn't store data

```
module reg_wire_eg(
input wire[2:0] any_input,
output wire[3:0] output_1
);

initial
begin
output_1 = 3'b111;
end
```

Error: procedural assignment to a non-register output_1 is not permitted, left-hand side should be reg/integer/time/genvar

Mostly used in combinational logic design



Data Types: Arrays

Syntax:

Declare an array:
<*data-type*> <*array*>[<*range*>]

Update an element:
<array>[<*index*>] = <*new value*>

Access an element
<*array*>[<*index*>]

```
// array of integers with length 3
integer array_of_int[0:2];


array_of_int[1] = 1;


// access element in location 1
temp = array_of_int[1];
```

Data Types: Arrays (continued)

Other types of array indexing:

`reg [7:0] memory [0:3];` // declare of an array with length of 4 and 8-bit width

Mostly used for holding values:

```
memory[0] = 8'hA0;                  // assign hex-value 0xA0
memory[1] = 8'b10001000;        // assign binary value 1000 1000
……
```

Note: `[0:7]` (0 to 7) and `[7:0]` (7 down to 0). Both can hold 8-bit width data, but the indices order are different.

Blocking and Non-Blocking Assignments

Syntax:

Blocking	Non-blocking
A = 1	A <= 1

- In blocking, the statements are executed sequentially.
- In non-blocking, the statements are executed simultaneously
- For example:

initially: `a = 1, b = 0`

run:
```
a <= a + 2
b <= a
```

We have a = 3, and b = 1. Each operation takes two steps to finish.

1. Compute the RHS
2. Assign value of LHS to RHS

In both a and b, the statements compute the RHS simultaneously, and assign the RHS value to LHS simultaneously.

Blocking and Non-Blocking Assignments (continued)

Consider the following set up:

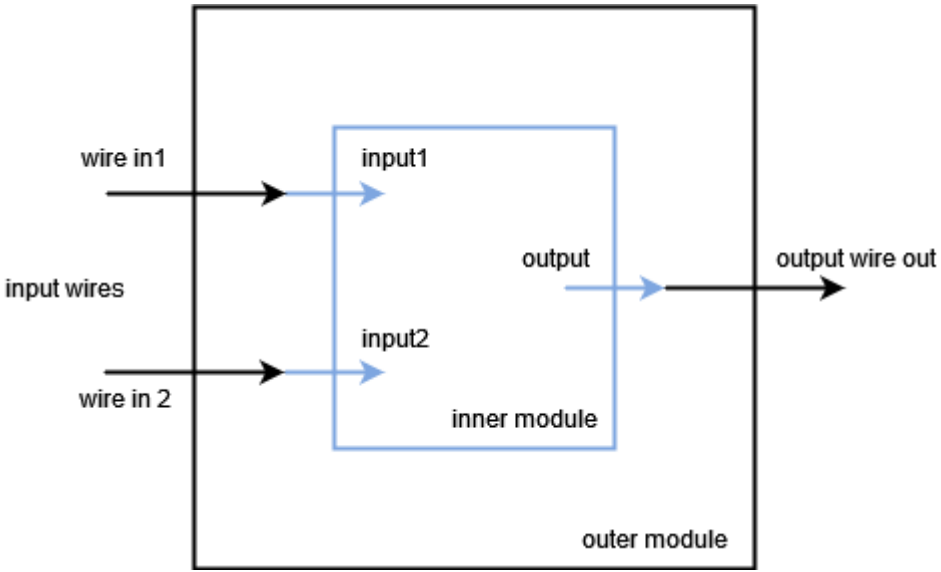
Initially a = 1, b = 2, c = 3. Execute the following statements twice:

Blocking	Non-blocking
a = c, b = a, c = b	a <= c, b <= a, c <= b

Result:

Blocking	Initial	1st	2nd
a	1	3	3
b	2	3	3
c	3	3	3
Non-blocking	Initial	1st	2nd
a	1	3	2
b	2	1	3
c	3	2	1

Modules: Instantiation



Instantiating another module inside a module by connecting their interfaces with wires.

This hierarchy structure allows the outer module to pass through input(s) to the inner module and to get output(s) from it.

Modules: Instantiation (continued)

```
1   `timescale 1ns / 1ps
2
3   module inner_module(input1, input2, out_put);
4       input input1, input2;
5       output out_put;
6       // some operations here
7   endmodule
```

```
1   `timescale 1ns / 1ps
2
3   module outer_module(in1, in2, out);
4       input in1, in2;
5       output out;
6
7       inner_module IM(in1, in2, out);
8   endmodule
```

Inner module(left) and outer module(right).

There are two instantiation methods:

1. by position(above): corresponding variables must be at the correct positions.
2. by name: positions doesn't matter. Different syntax.

```
inner_module IM(.input1(in1), .input2(in2), .out_put(out))
```

Modules: Port Connections

Recall from the last section, multiple modules can be instantiated in the same module.

Modules have input and output interfaces. Different modules can be connected by linking the interfaces with wires.

Once you have finished designing and implementing all the modules, you need to connect them together to simulate the MIPS processor.

Modules: Port Connections (continuous)

Using the example in module instantiations section with 2 additional inner modules. For simplicity, these two are the same as the first module but with different names.

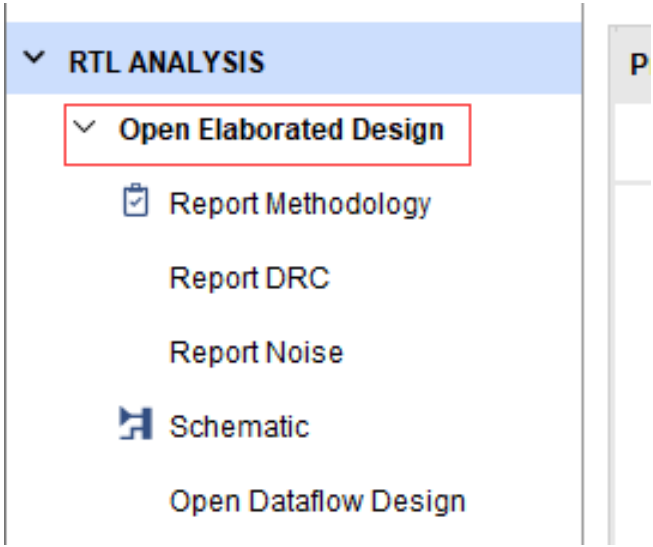
```
1  `timescale 1ns / 1ps
2
3  module outer_module(in1, in2, out);
4      input in1, in2;
5      output out;
6
7      wire IM1_output, IM2_output;
8
9      inner_module IM(in1, in2, IM1_output);
10     inner_module2 IM2(in1, in2, IM2_output);
11     inner_module3 IM3(IM1_output, IM2_output, out);
12 endmodule
```

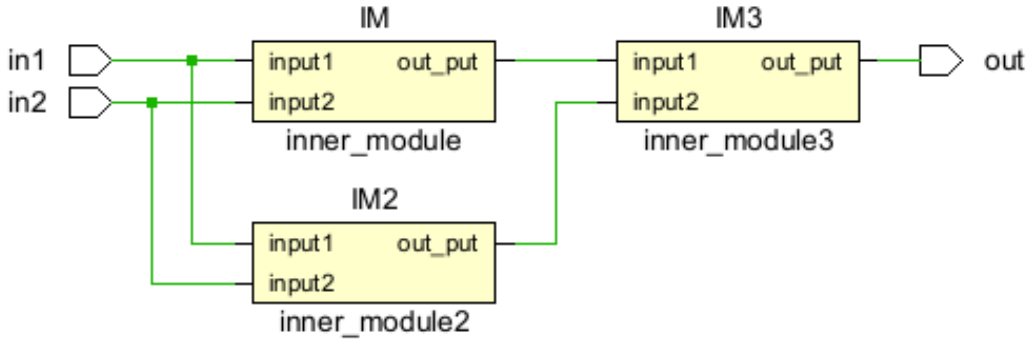
1. module 1 and module 2 take inputs in1 and in2
2. module 3 takes output from module 1 and module 2, producing the output for the outer module

We need 2 wires to connect module 3's input ports with module 1's and 2's output port, which is `IM1_output` and `IM2_output` in the code.

Modules: Port Connections (continued)

You can use the function ‘open elaborated design’ in the Vivado IDE navigator window to view your schematic and compare it with your project design.



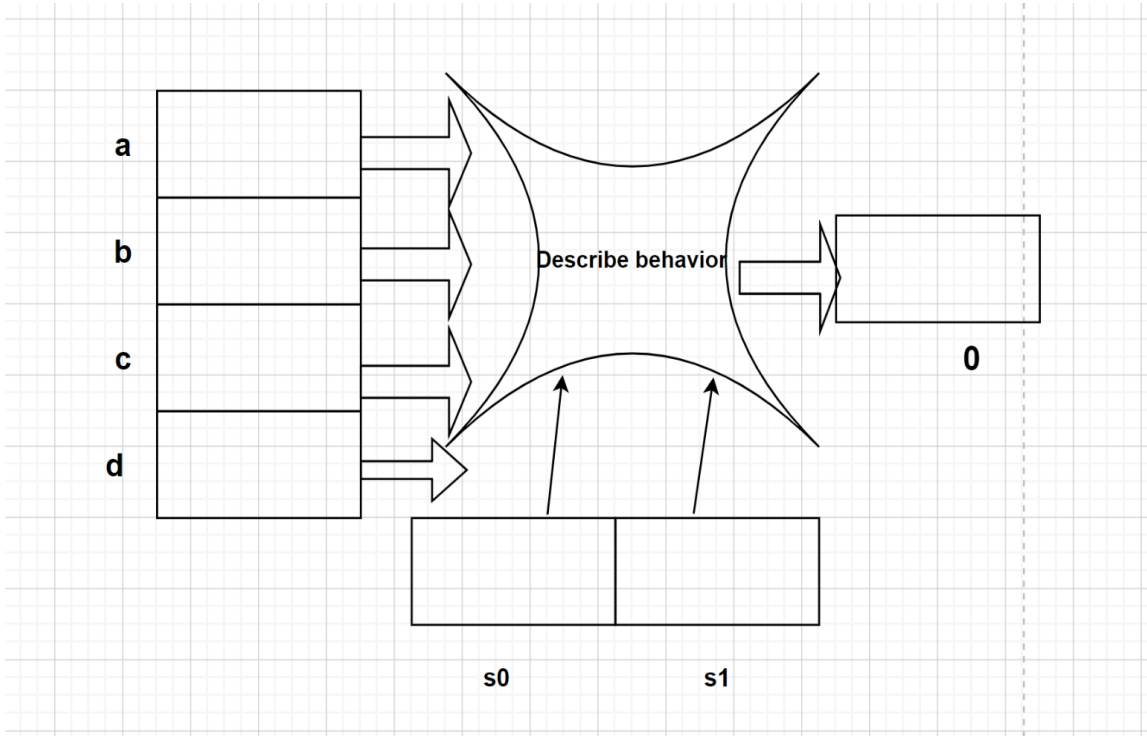


Behavioral vs Structural Verilog

Behavioral Verilog

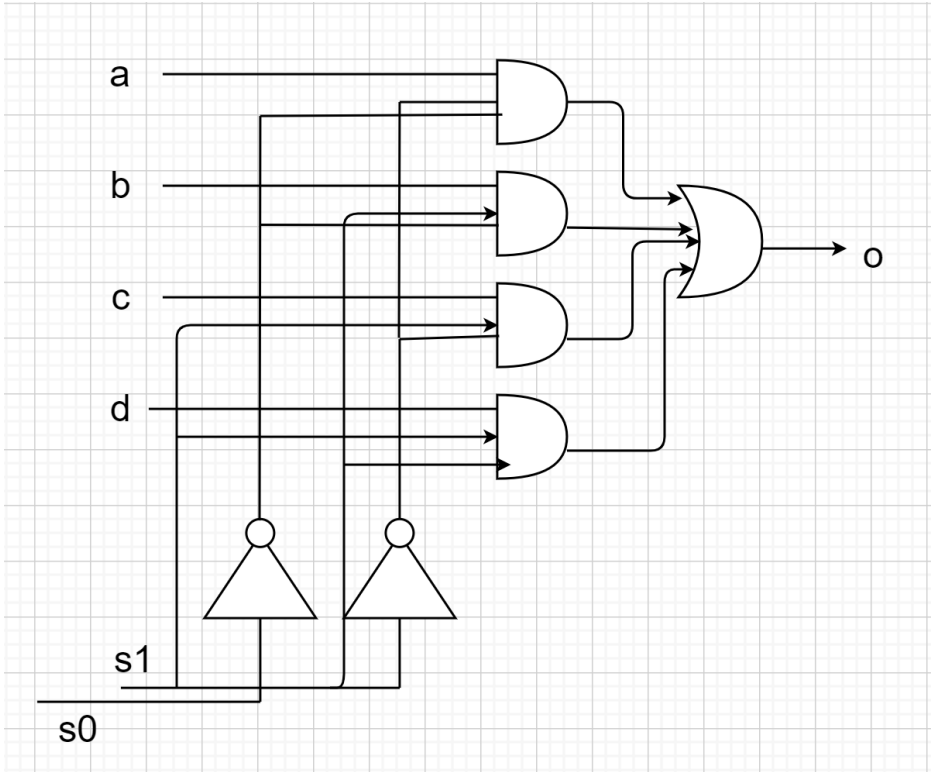
```
module mux_4to1 ( a, b, c, d, s0, s1, out);

input wire a, b, c, d;
input wire s0, s1;
output reg out;
always @ (a or b or c or d or s0, s1)
begin
case (s0 | s1)
2'b00 : out <= a;
2'b01 : out <= b;
2'b10 : out <= c;
2'b11 : out <= d;
endcase
end
endmodule
```



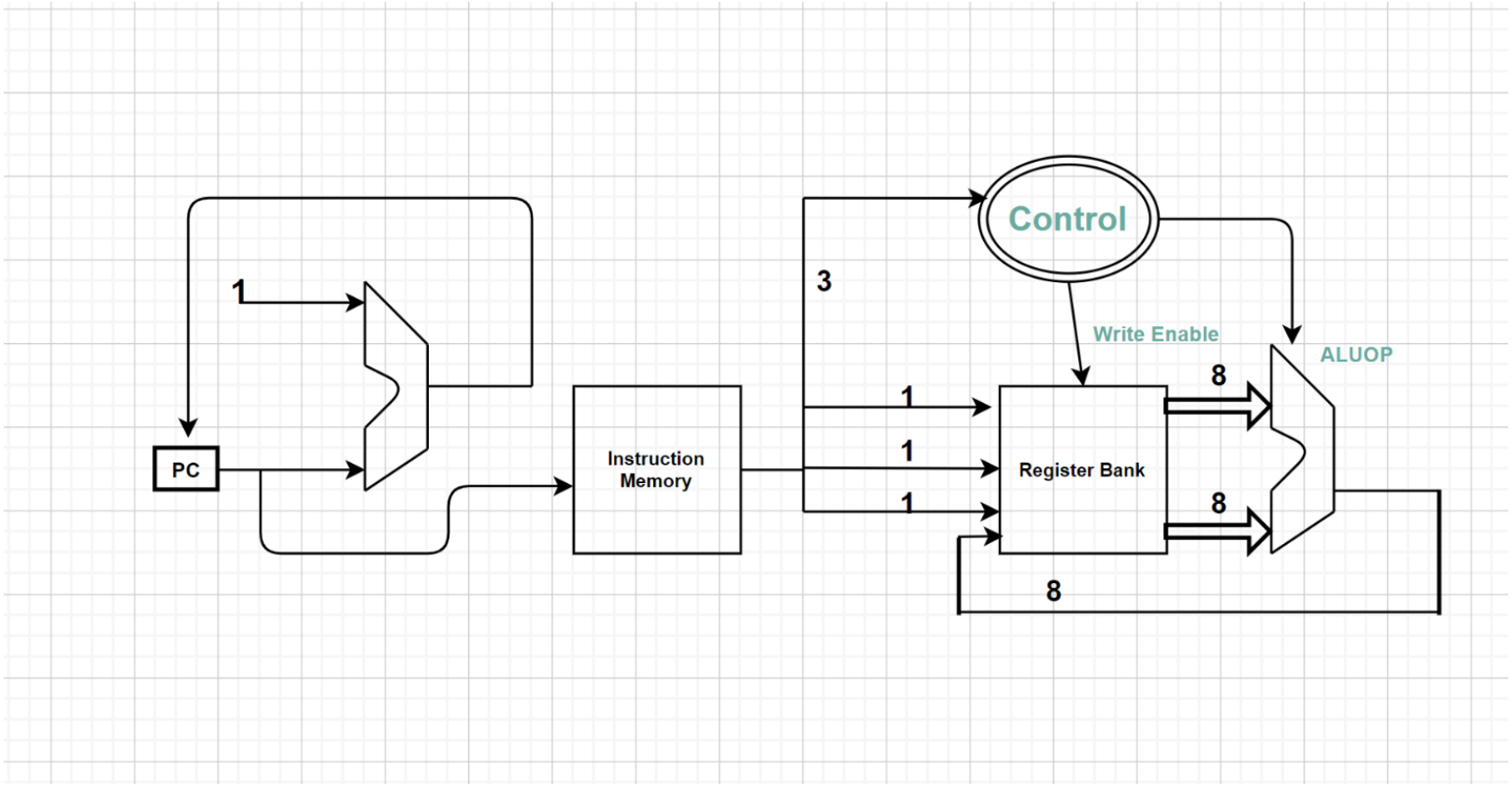
Structural Verilog

```
module mux_4to1(out, a, b, c, d,
s0, s1);
output out;
input a, b, c, d, s0, s1;
wire s0bar, s1bar, T1, T2, T3;
not u1(s1bar, s1);
not u2(s0bar, s0);
and u3(T1, a, s0bar, s1bar);
and u4(T2, b, s0, s1bar);
and u5(T3, c, s0bar, s1);
and u6(T4, d, s0, s1);
or u7(out, T1, T2, T3, T4);
endmodule
```

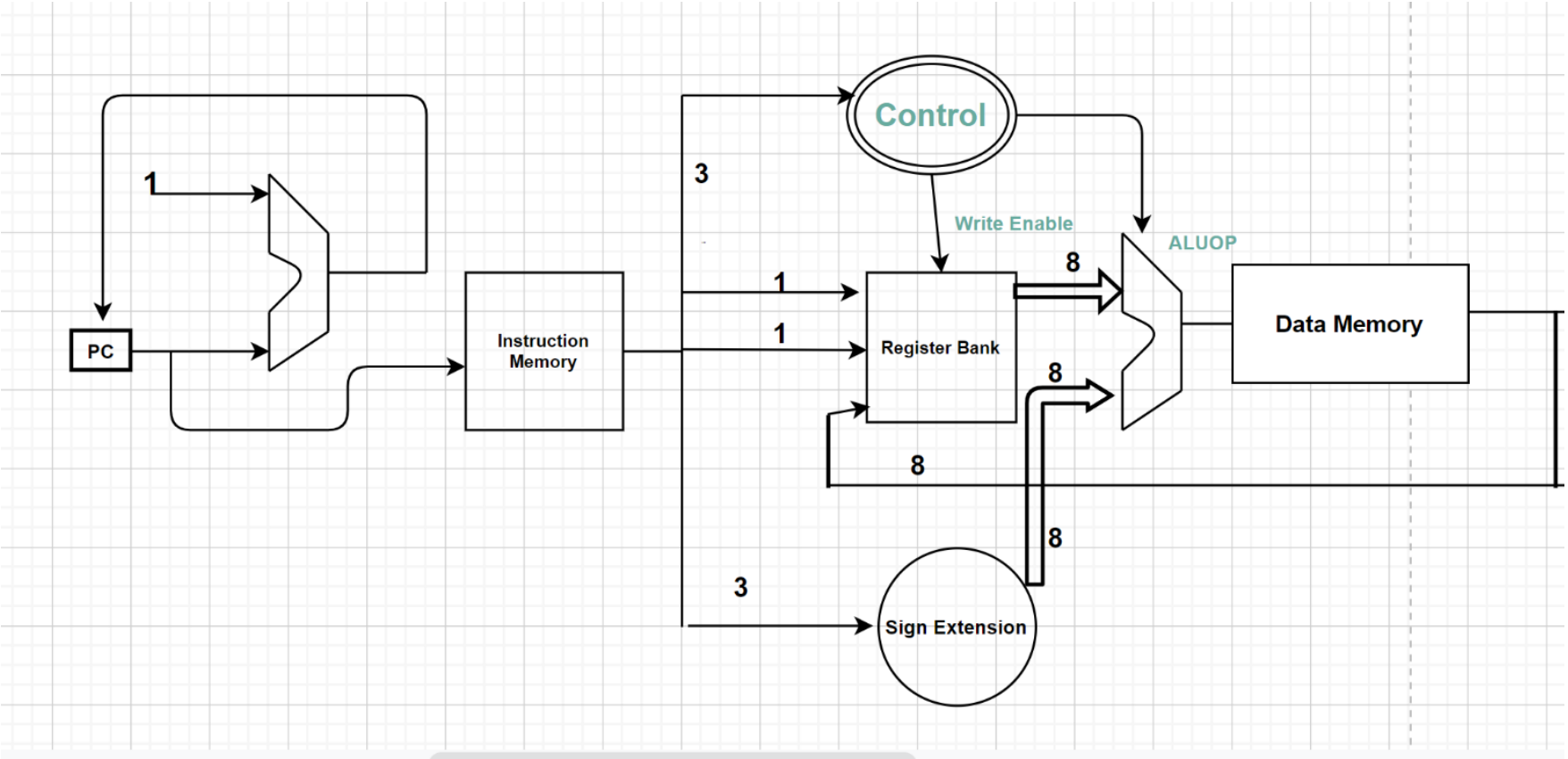


Datapath Diagram

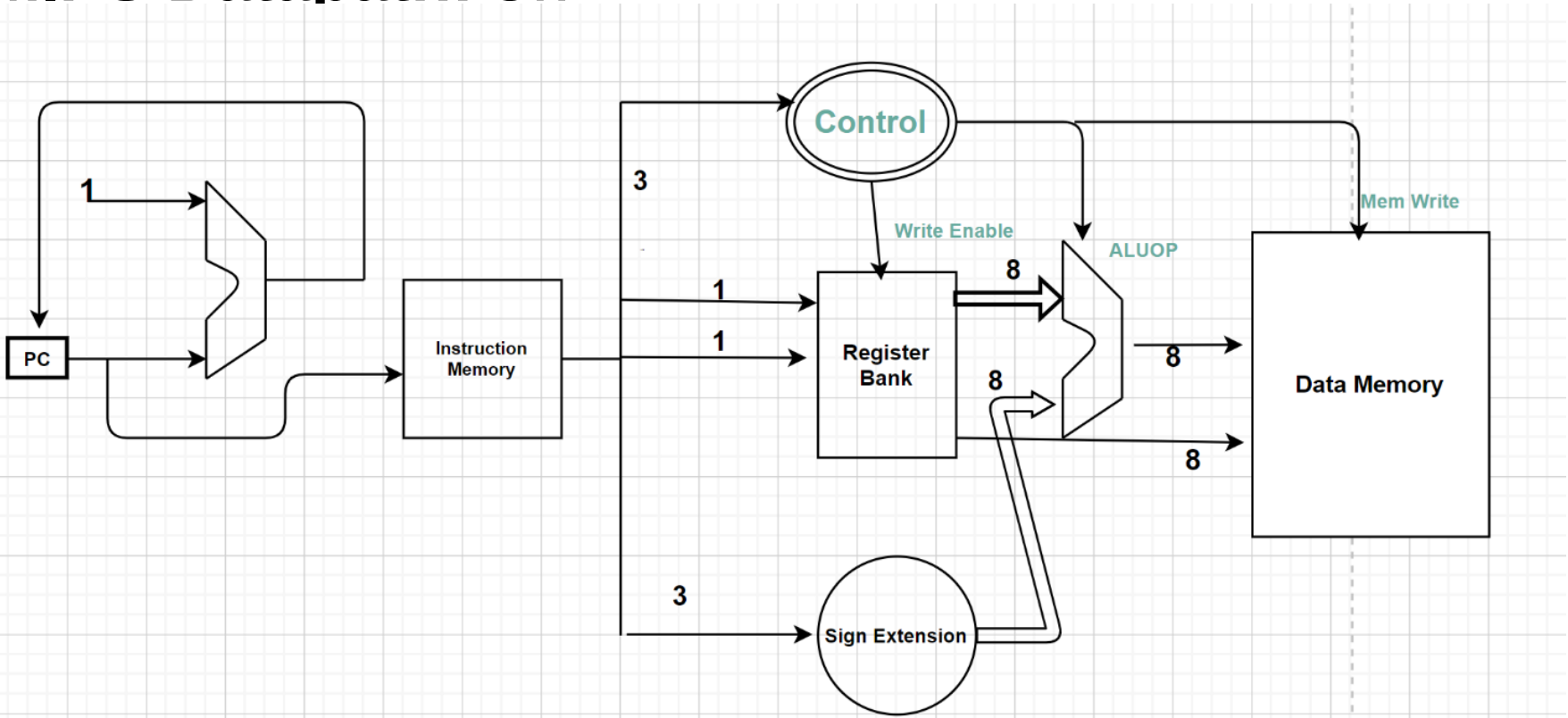
MIPS Datapath: add



MIPS Datapath: lw

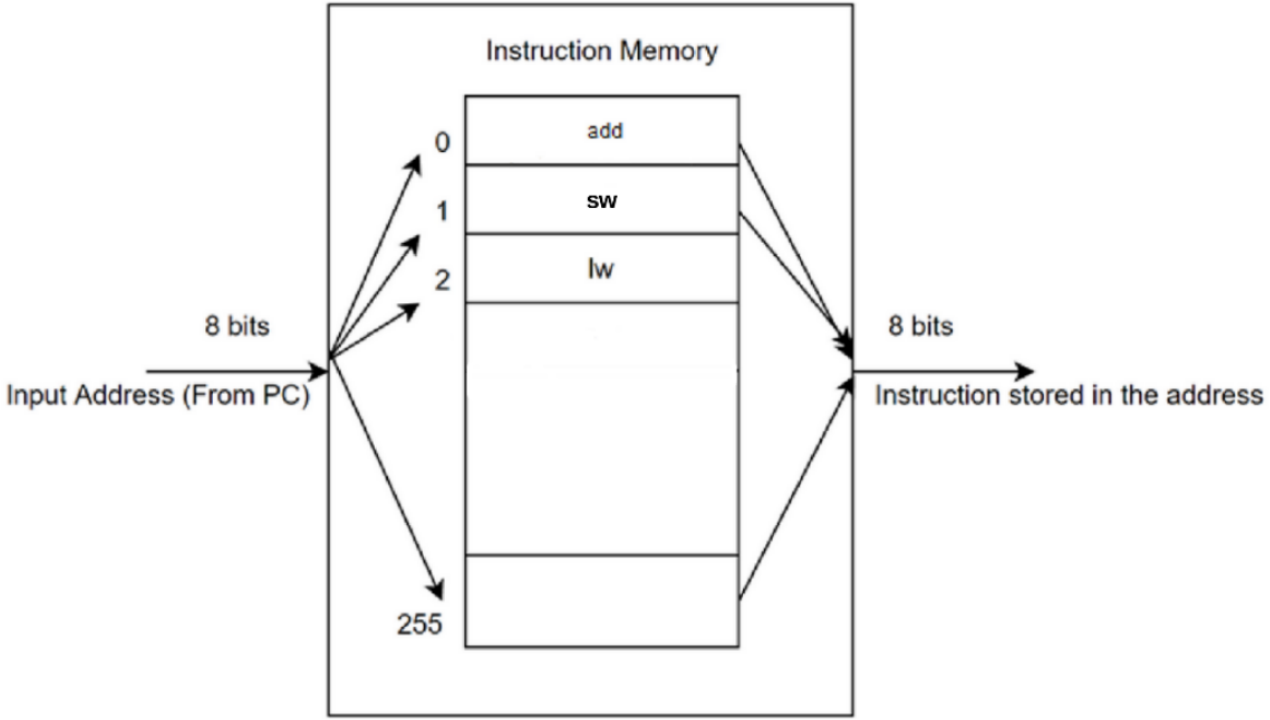


MIPS Datapath: sw

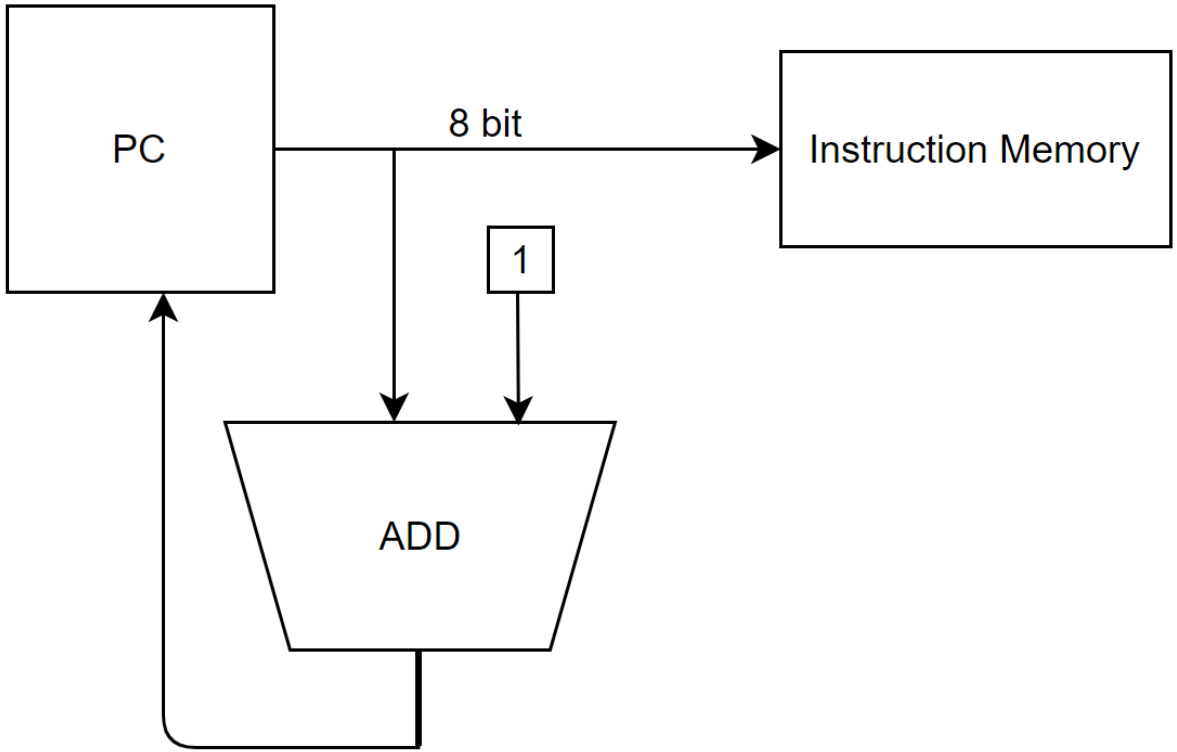


Design Ideas of Individual Components

DATAPATH


Instruction Memory
0
1
2
255
add
sw
lw
8 bits
Input Address (From PC)
8 bits
Instruction stored in the address

DATAPATH



R-type instruction:

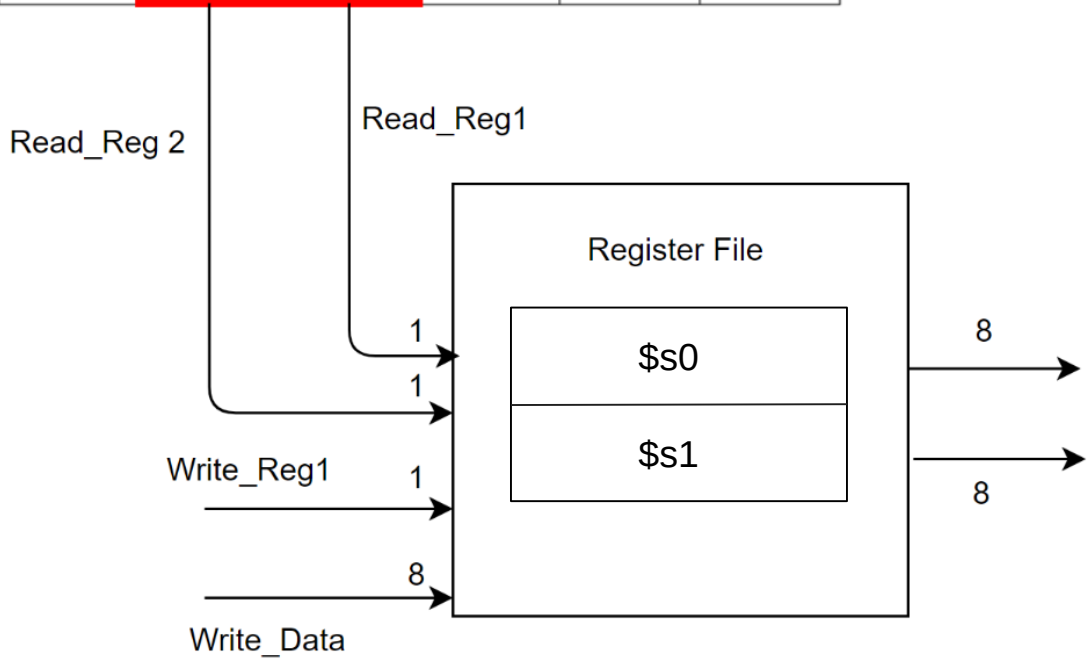
Opcode			Rt/Rd	Rs	Unused		
7	6	5	4	3	2	1	0

I – type Instruction:

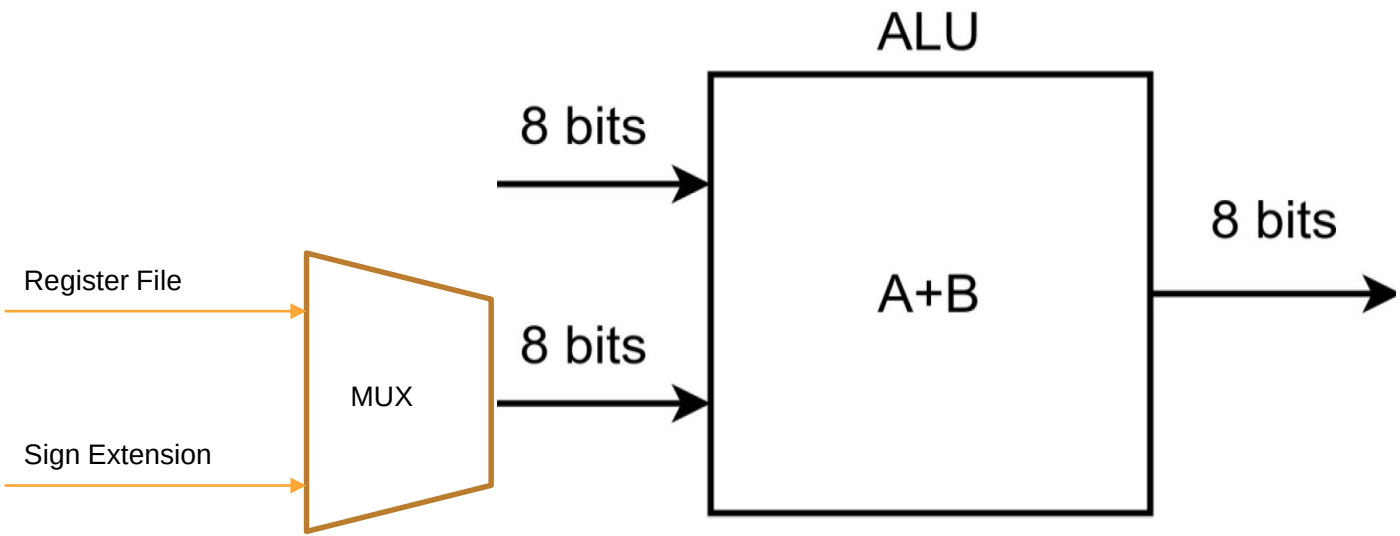
Opcode			Rd	Rs	Immediate		
7	6	5	4	3	2	1	0

Given: 2 Registers $s0 , $s1 (each 8 bits)

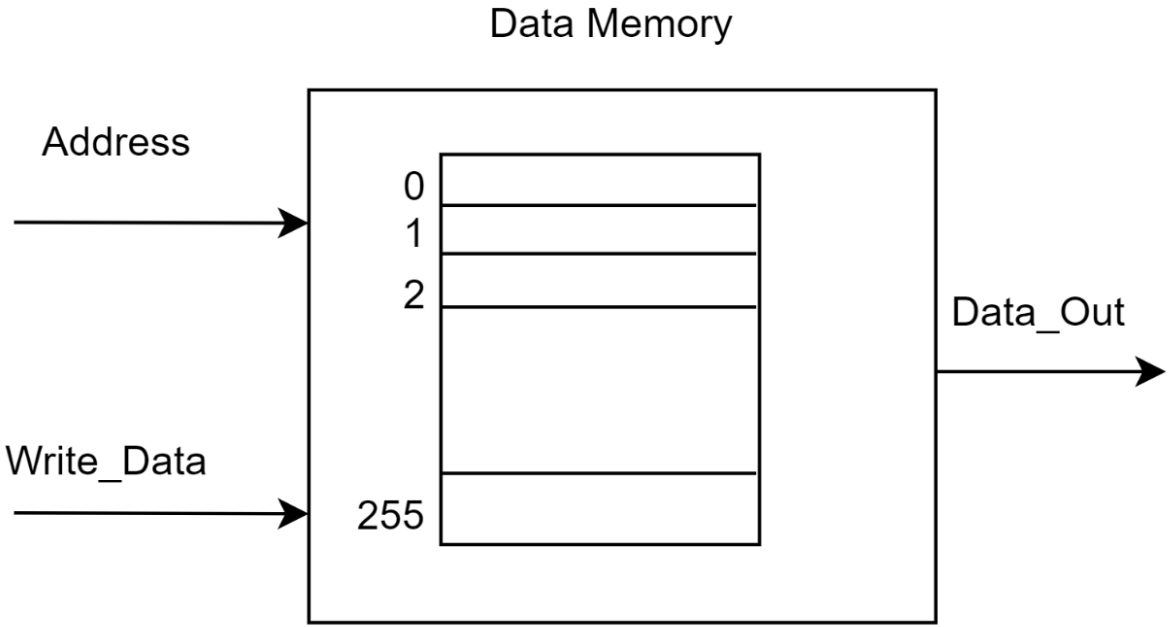
DATAPATH



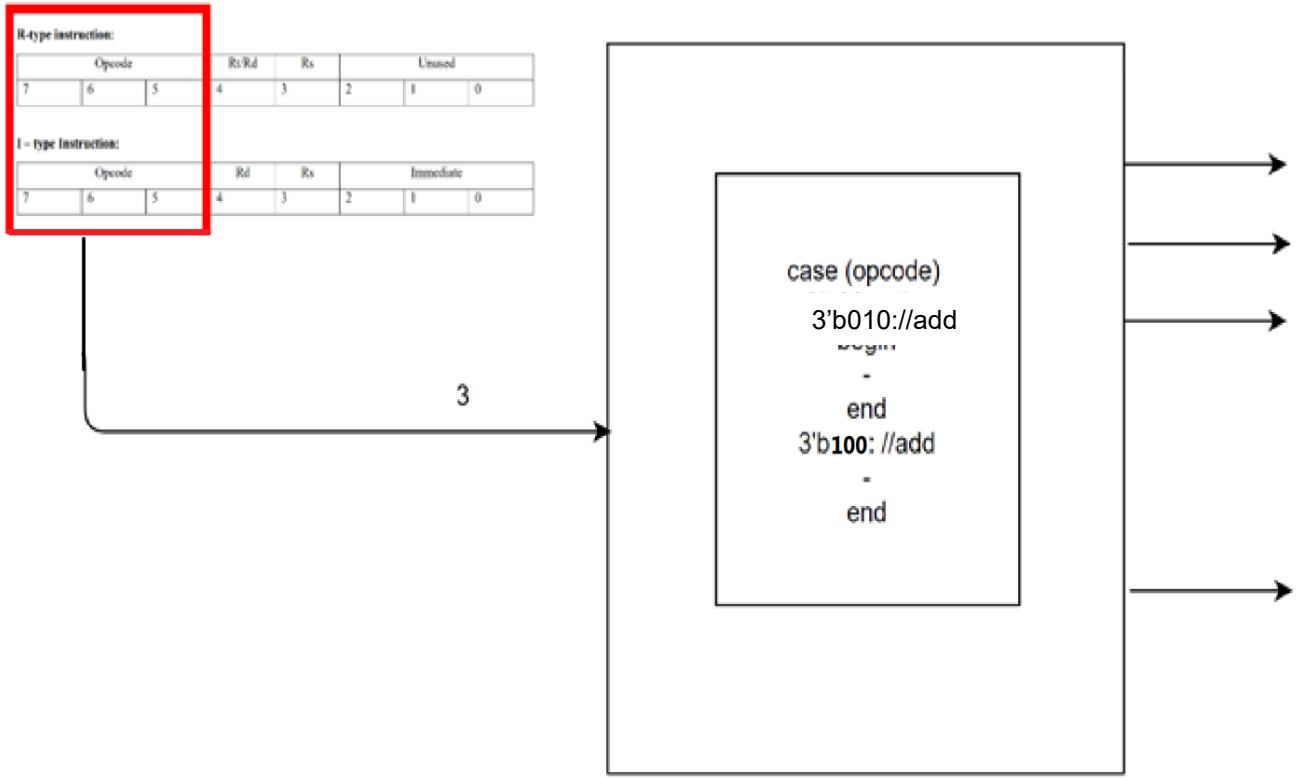
DATAPATH



2 Modes – Read from Memory (lw)
– Write into Memory (sw)



CONTROL UNIT



Where to Start

- Refer to FAQ in project description
- Partial Credits – Individual Modules will be graded
    - Test each module
- Start with PC, ALU and build from there
- Attend TA Office Hours
- DO NOT COPY!!!
    - We have access to src code from previous years, Github and other online materials as well
    - We use multiple plagiarism check resources, such as MOSS to check for academic integrity violations

General Pointers

- Look out for Submission Instructions on Piazza
- Attend Office hours and post on Piazza to clarify your doubts
- Read the piazza posts related to Project. It may help you with the design
- Include any reference that you used in your report