CS 240 – Computer Architecture

Fall 2024

Lab 4:

RISC-V Decoder

&

Verilog Module Implementation

Concepts

Hardware Description Language

Hardware Description Languages are special languages used to define internal signals and behavior of digital hardware. Some of the very common HDL are Verilog, VHDL and SystemVerilog.

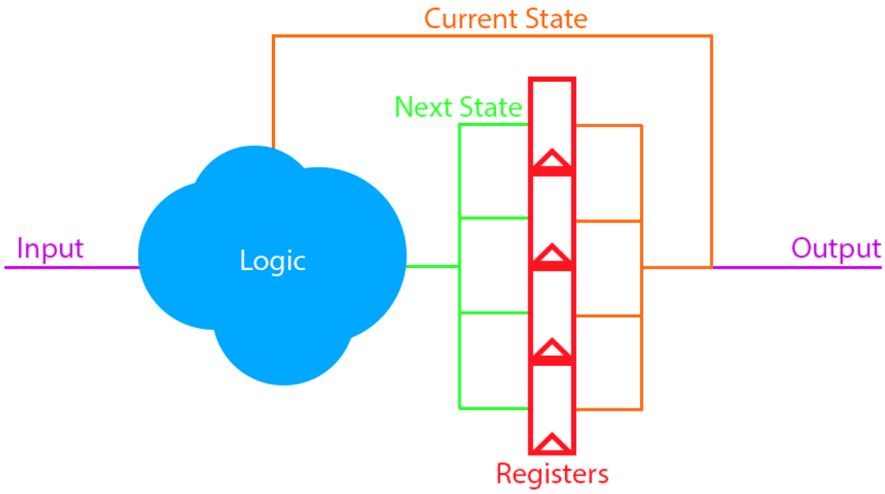
HDLs allow users to easily perform low-level operations over signals like bit manipulations or bit-level arithmetic and logical operations.

Synthesizers

Synthesizers can be considered compilers for HDL. They read HDL code and output a netlist. Netlist is a collection of digital signals, modules, and connections of the related hardware. Like compilers, synthesizers also may warn about possible bugs and unimplementable code. They also perform optimizations over the design.

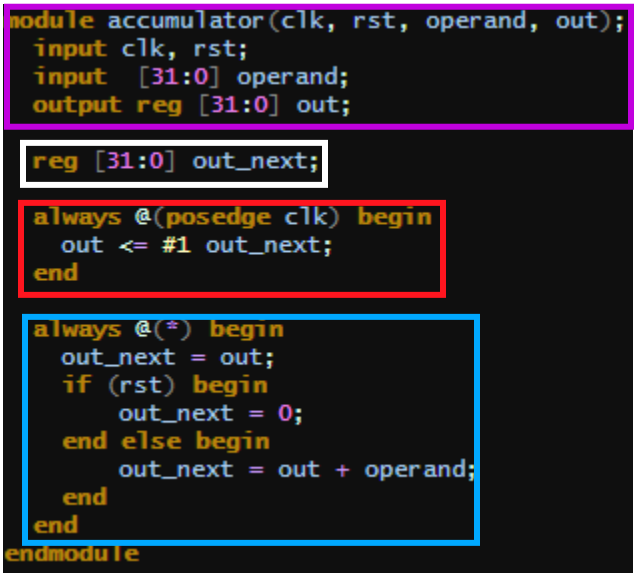
State machines and Logic Design

You can imagine digital hardware as a logic cloud that operates on its registers (or memory). Registers hold the current state of your hardware. Depending on the current state and input, your logic will determine the next state.



This type of logic is called a state machine. CPUs work in a similar way. Every cycle they look at their input (instruction) and current state (register file and memory), then determine their next state. Of course, modern CPUs utilize many advanced concepts which makes them more than simple state machines.

Structure of A Verilog File



You can see the structure of a basic accumulator design on the left.

The purple area is where we define the inputs and outputs of the design.

The white area is where we define all internal signals. Both the signals that represent the current state and the next state can be defined here.

The red area is where we represent our register behavior. In most cases, we do not put any extra logic in there. We just represent the transition from the current state to the next state.

The blue area is where we define all combinational logic. We always modify the signals that represent the next state. Since we only commit our changes at the end of a cycle, any changes to the same signal will override all changes above it. For example, in the above example, if the reset signal is 1, out_next = 0; will always override out_next = out; logic.

There are two basic variable types in Verilog, reg and wire. Reg does not stand for register. Basically, anything you change inside the always blocks should be defined as reg. Everything else should be defined as wire.

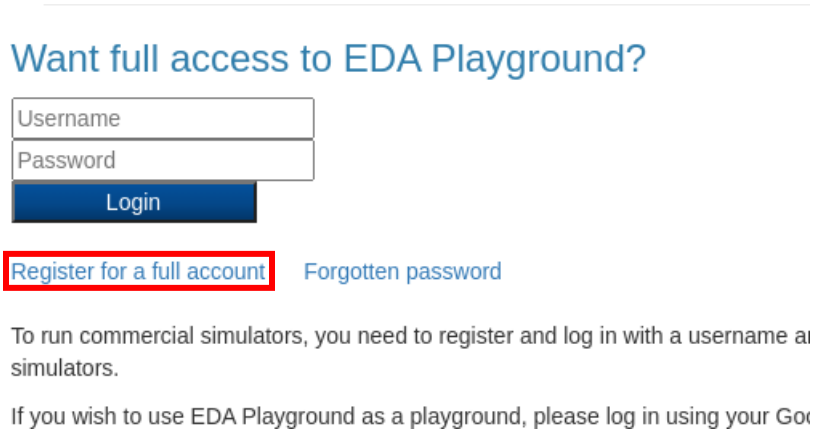
Always blocks represent different logic groups inside your hardware. Every always block runs in parallel. You cannot modify the same signal from different always blocks. This is called a multiple-driver problem. A signal cannot have two values at the same time.

Scope of the Lab

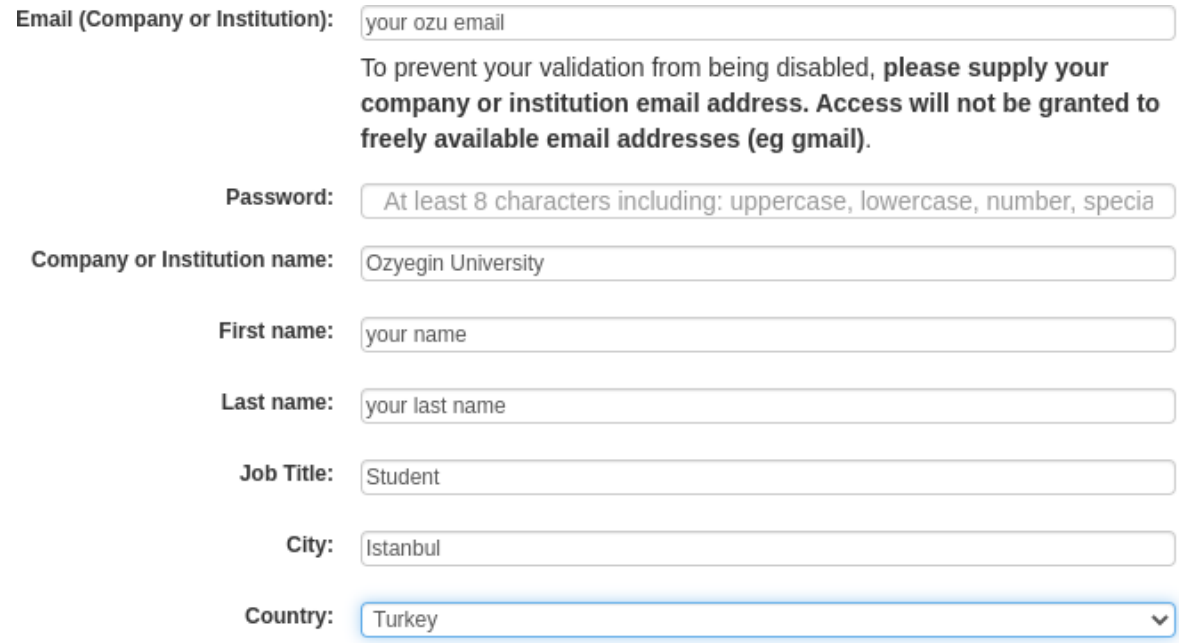
In this assignment, you will implement behavioral logic for two hardware modules in Verilog. For the first task, you will implement a RISC-V Decoder module. The module will only extract related fields and the control signals for each instruction type. The second module will have two different behaviors which will be executeddepending on the input signal fetched from the memory. You do not have to worry about memory implementation for this task.

Resources

For this lab, we will use EDA Playground. EDAP is a website that provides both open-source and commercial tools to develop hardware for educational purposes. To access these tools and run your code, you need to register first. To register to EDAP, first, open the home page, then press login. In the login page, you should see an option to register for full access.



On the registration page, enter your information below.



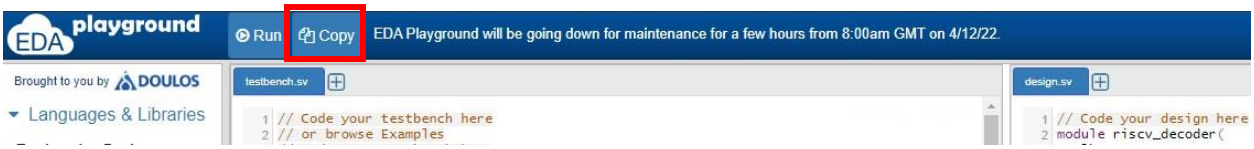
To complete your registration, open your email and confirm your email from the registration mail.

You are given the skeleton for each module implementation. You can reach each task from the following links.

Task1: https://www.edaplayground.com/x/TedX

Task2: https://www.edaplayground.com/x/HMgg

When you first open these tasks, click the copy button on the top left of the page.

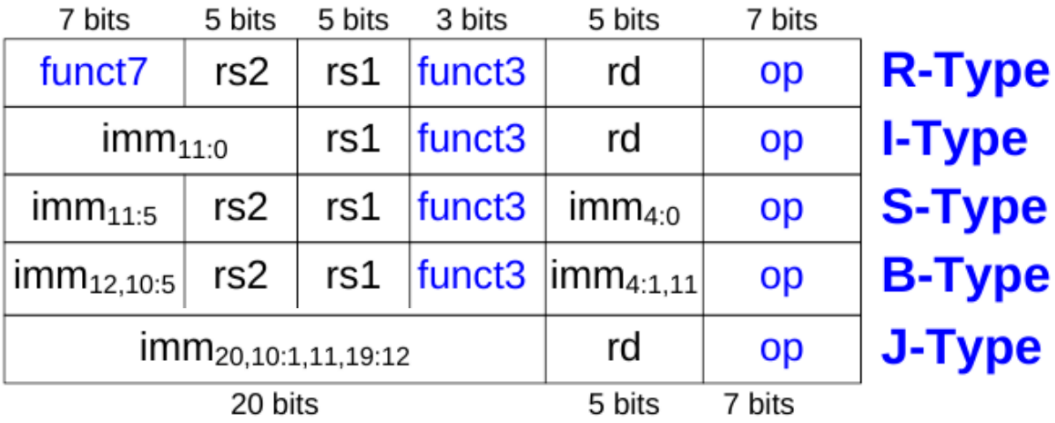


This will create a copy on your account.

Exercise 1

During this lab, you will implement the behavior for RISCV_Decoder module. This module will take a 32-bit instruction and it will return the related fields for each instruction type.

Instruction types to be decoded:



You should decode opcode and operand signals for each instruction types (e.g., for I type: opcode, rd, funct3, rs1, imm12).

In addition to opcode and operand signals, you should also decode the main control signals according to the Table 1; and ALU control signals according to Table 2.

Table 1. Main Control Signal Decoder Truth Table

Instruction	Opcode	RegWrite	ImmSrc	ALUSrcA	ALUSrcB	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw (I-type)	0000011	1	00	0	1	0	01	0	00	0
sw (S-type)	0100011	0	01	0	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	0	00	0	10	0
beq (B-type)	1100011	0	10	0	0	0	xx	1	01	0
I-type ALU	0010011	1	00	0	0	0	00	0	10	0
jal (J-type)	1101111	1	11	x	x	0	10	0	xx	1

Table 2. ALU Control Signal Decoder Truth Table

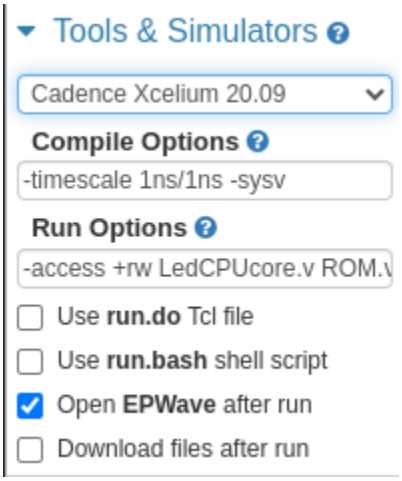
$ALUOp_{1:0}$	$funct3_{2:0}$	$\{op_5, funct7_5\}$	$ALUControl_{2:0}$	Operation
00	x	x	000	Add
01	x	x	001	Subtract
10	000	00, 01, 10	000	Add
	000	11	001	Subtract
	010	x	101	SLT
	110	x	011	OR
	111	x	010	AND

Below is the test program to test your decoder implementation.

```
#       RISC-V Assembly         Description                 Address   Machine
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10        0041F2B3
        add  x5, x5, x4         # x5 = (4 + 7) = 11         14        004282B3
        beq  x5, x7, end        # shouldn't be taken        18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         1C        0041A233
        beq  x4, x0, around     # should be taken           20        00020463
        addi x5, x0, 0          # shouldn't happen          24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5)  = 1         28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12        2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7         30        402383B3
        sw   x7, 84(x3)         # [96] = 7                  34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7             38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18        3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44    40        008001EF
        addi x2, x0, 1          # shouldn't happen          44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18)  = 25       48        00910133
        sw   x2, 0x20(x3)       # mem[100] = 25             4C        0221A023
done:   beq  x2, x2, done       # infinite loop             50        00210063
```

After completing the module, you will first synthesize the module to see if any warnings or errors occur. You can synthesize your module by selecting theMentor Precision tool from the sidebar, then pressing the Run button.

After making sure your module synthesizes, you should simulate the module by selecting the Cadence Xcelium tool from the sidebar.



You can also minimize the EPWave window and check if any warnings for wrong values are present in the terminal window. If you have no warnings in the simulation window, it means that you have completed the task.

Exercise 2

You should implement a module that, in every cycle, reads from the memory and changes its behavior depending on the data fetched from the memory. Each data is 16 bits. Your module should read the rightmost 8 bits of the data and change its behavior. If the rightmost 8 bits are not equal to 0, your module should wait up to that many cycles before getting the next line and in the meantime, it should output the leftmost 8 bits. If the rightmost 8 bits are equal to zero, then your module should treat the rightmost 8 bits as an address and change the PC registers value accordingly.

BIT #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Output	OUTPUT PATTERN								DELAY ≠ 0							
Jump	JUMP ADDRESS								DELAY = 0							

So, we have the following logic:

1) DELAY = 0 => Do Jump operation

   BIT[15:8] is the address we jump to.

2) DELAY ≠0 => Do Output operation

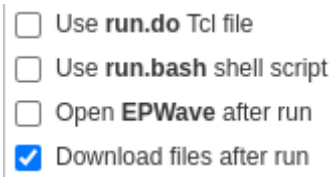
   BIT[15:8] is the pattern shown in the LEDs.

   BIT[7:0] is the amount of time this particular pattern should be shown in the LEDs.

   Then it moves to the next address.

The CPU simply starts from address 0 and executes the following instructions.

After completing the module, you should synthesize your code as mentioned above. If you get no warnings, then you should simulate your code. This time, change your simulation options as shown in the following image.



This will download a zip file containing project files. Find “dump.vcd” file inside this zip file and open it using GTKWave. Observe the wave form.