Lab 11. TOP MODULE TO UNITE THEM ALL

We have already written all of the components of MIPS Processor: Branch control evaluation unit, general purpose registers, instruction decoder, ALU MEMORY, shifter, signal extension. Now it is time to combine all of our modules and obtain MIPS Processor depicted on figure 1.

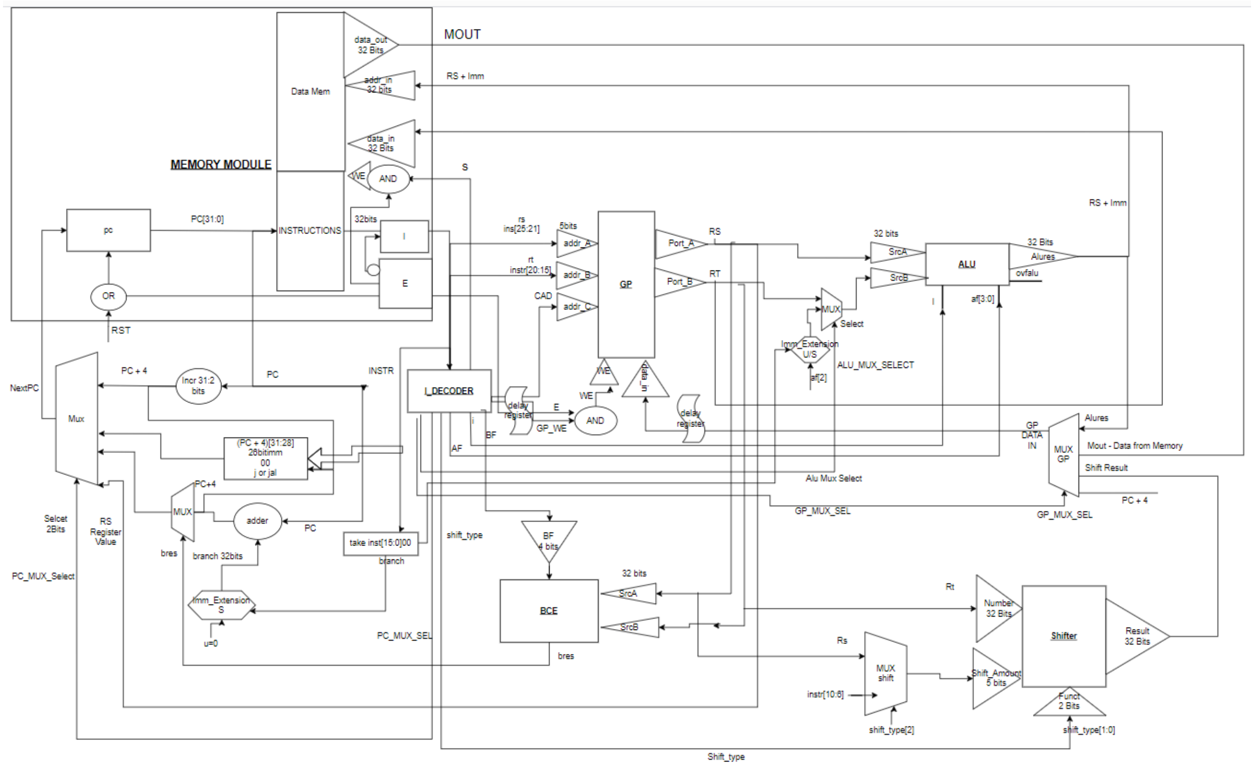


Figure 1

Generally, when we are designing a multimodule hardware in Verilog, we write each of the modules separately and then instantiate all of them in one **TOP MODULE**. We are going to do the same for our processor modules as well.

Figure 1 can be divided into X parts:

NEXT PC :

Figur 2 depicts part of the processor that calculates the next pc value.

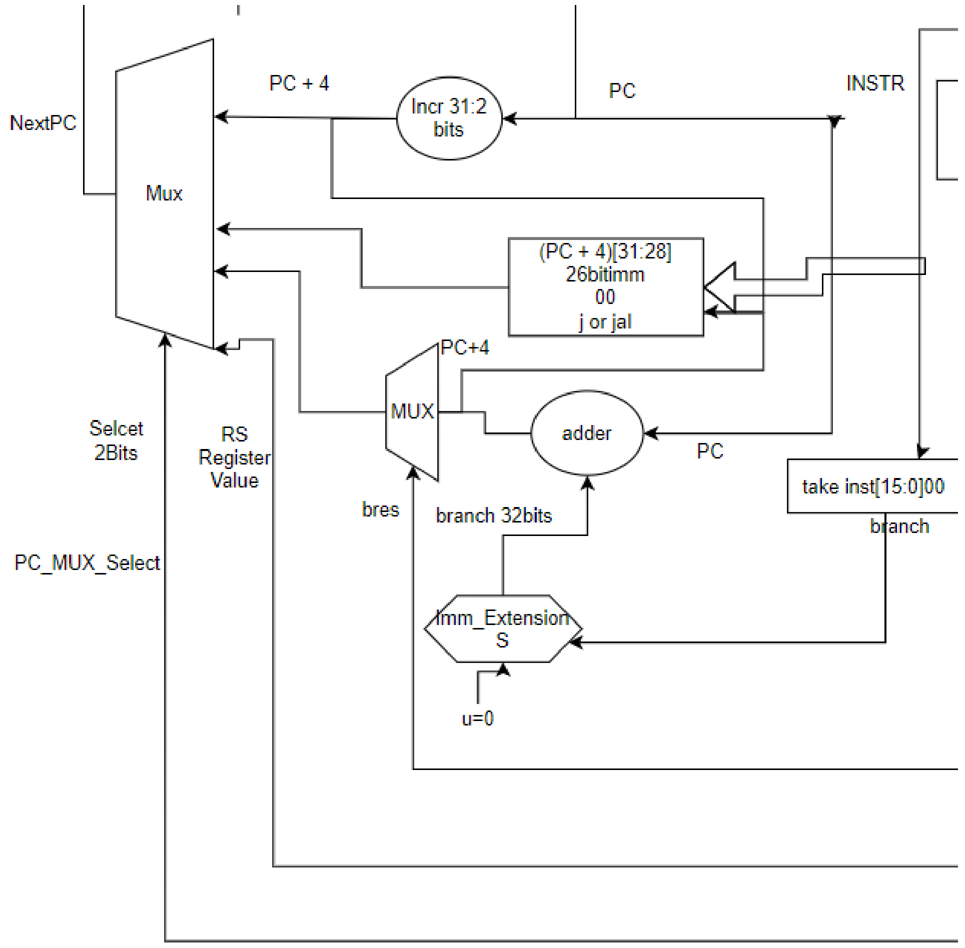


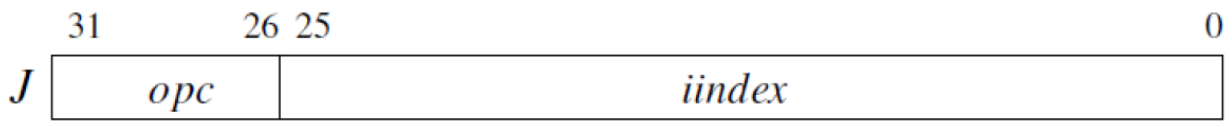
Figure 2

Next PC can have 4 possible values. That is why we use a MULTIPLEXER controlled by PC_MUX_SEL signal (generated by instruction decoder).

<u>**Possible values for NEXT PC:**</u>

- If we have jump from register (jr) or jump and link from register (jalr) instruction, the address of next instruction is read from general purpose register whose address is specified by RS.

- If we have branch instruction and the branch condition is TRUE, the address of the next instruction is calculated by adding current PC and sign extended branch distance. If the branch condition is false, the next pc is current pc plus 4.
- If we have jump (j) or jump and link (jal) instruction, the next instruction is calculated based on jump index and current PC.



The last 26 bits of jump instruction is an immediate. The first four bits of the next instruction is the first four bits of PC+4, concatenated with iindex and two 0s.

$$Next\ PC = (PC + 4)[31:28]iindex00$$

- If we have neither jump nor branch instructions, Next PC is PC + 4.

<u>General Purpose Registers</u>:

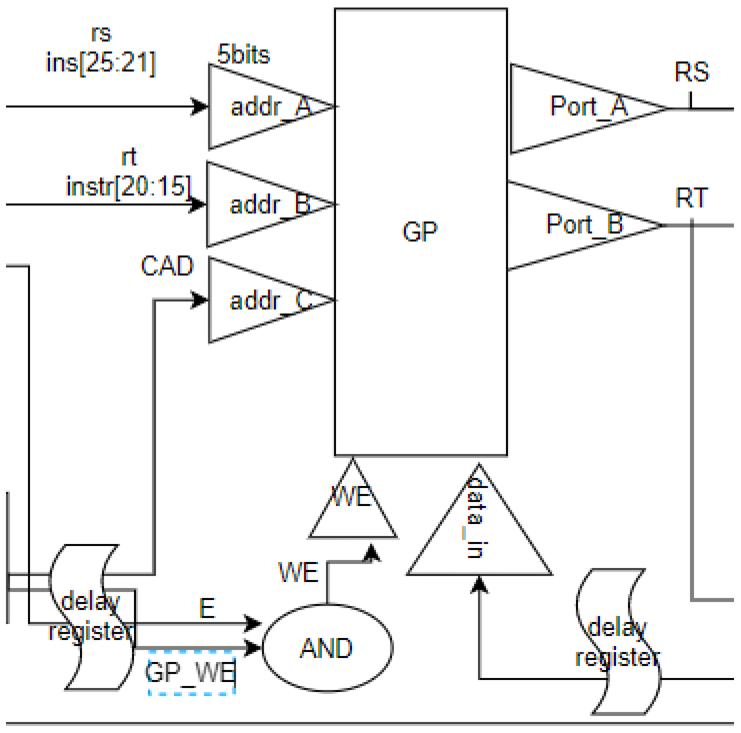


Figure 3

Figure 3 depicts circuits associated with general purpose registers (GP). As we can see, the address of PortA and PortB are directly RS (inst[25:21]) and RT (inst[20:15]) fields of instruction.

CAD comes from instruction decoder and specifies the address of register where data should be written to.

E is output of E register from the memory and GP_WE is register write enable from instruction decoder. E and GP_WE specifies write enable for general purpose registers. BEAR IN MINDS, that E, GP_WE, data_in and CAD should be delayed by 1 clock cycle before they enter into GP module. (See HINT for signal delay).

ALU Circuit:

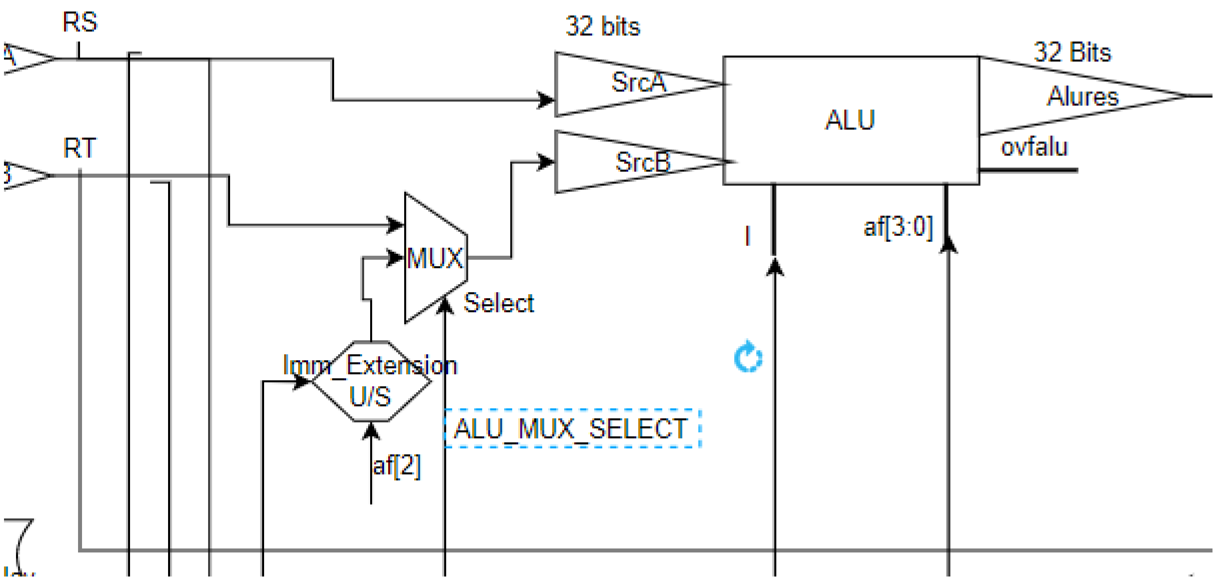


Figure 4

Figure 4 displays circuits associated with ALU. The second operand of ALU operations is either immediate embedded in instruction or value read from RT register. The multiplexer chooses between those two values. Also, we have immediate extension unit to extend 16 bit immediate to 32 bit number. Type of extension (signed or zero) depends on af[2] bit.

GP_MULTIPLEXER:

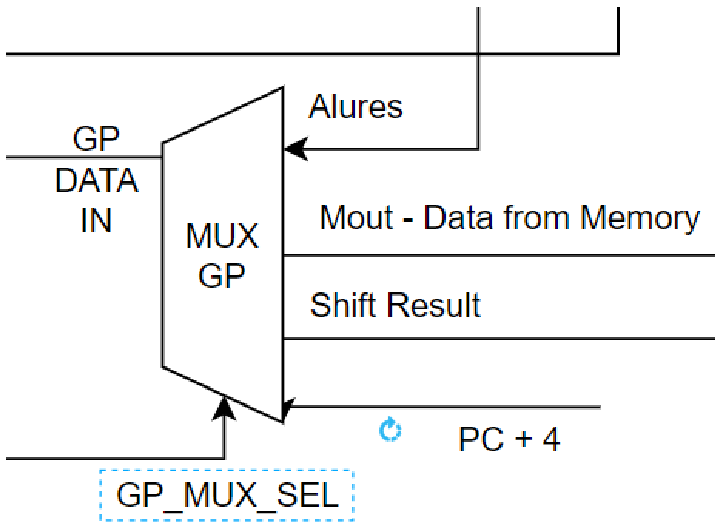


Figure 5

There are 4 possible values for data that can be written in general purpose registers: Alu result, data from memory, shift result, and address of the next instruction. GP_MUX_SEL (generated in instruction decoder) chooses between those 4 values.

Input:

Clock and reset – 1-bit long input signals.

Outputs:

aluresout,shift_resultout,GP_DATA_INout – 32 bit long signals

LAB TASKS

1) **Implementation**: You already have separate modules for BCE, GP, ALU, MEMORY, Shifter, Instruction decoder, immediate extension. Your task is to instantiate those 6 modules in one top module to obtain processor depicted on figure 1. You need to implement multiplexers in your top module
2) **Simulation & Verification**: You are REQUIRED to write a testbench and test if your Processor works correctly. You are suggested to read the following instructions in MEMORY from text file.

```
10001100100001010000000000100010 // LW
10101100100001010000000000100010  // SW
00100000100001010000000000000100  // ADDI
00100100100001010000000000000100  // ADDIU
00101000100001010000000000000100  // SUBI
00101100100001010000000000000100  // SUBIU
00110000100001010000000000000100  // ANDI
00110100100001010000000000000100  // ORI
00111000100001010000000000000100  // XORI
00111100100001010000000000000100  // LUI
00000100100000000000000000000100  // bltz
00000100100000010000000000000100  // bGEz
00010000100000010000000000000100  // beq
00010100100000010000000000000100  // bne
00011000100000000000000000000100  // blez
00011100100000000000000000000100  // bgtz
00000000100001010010000001000000  // SLL
00000000100001010010000001000010  // SRL
00000000100001010010000001000011  // SRA
00000000100001010010000000000100  // SLLV
00000000100001010010000000000110  // SRLV
00000000100001010010000000000111  // SRAV
00000000100001010010000000100000  // ADD
00000000100001010010000000100001  // ADDU
00000000100001010010000000100010  // SUB
00000000100001010010000000100011  // SUBU
00000000100001010010000000100100  // AND
00000000100001010010000000100101  // OR
```

00000000100001010010000000100110 // XOR

00000000100001010010000000100111 // NOR

00000000100001010010000000101010 // SLT

00000000100001010010000000101011 // SLTU

00000000100001010010000000001000 // JR

00000000100001010010000000001001 // JALR

00001000000000000000000000001001 // J

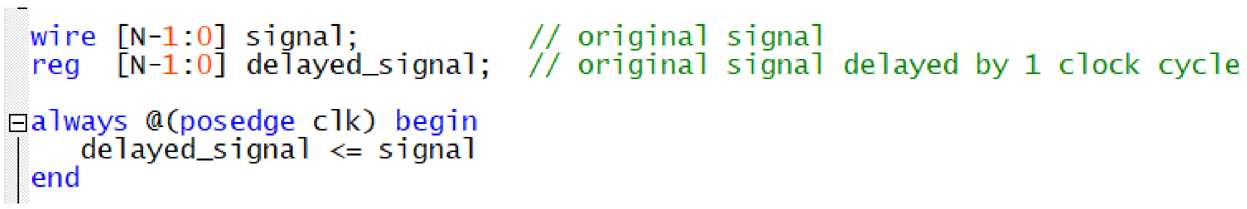
00001100000000000000000000001001 // JAL

00000000000000000000000000001111 // data

With this module, you finish designing your first processor. We are proud of you.

Good Luck in The Future

HINT How to Delay a Signal



```
wire [N-1:0] signal;          // original signal
reg  [N-1:0] delayed_signal;  // original signal delayed by 1 clock cycle

always @(posedge clk) begin
    delayed_signal <= signal
end
```