Lab 11. Instruction Decoder

The most important module of any processor is an Instruction Decoder. The module takes a 32-bit long instruction as an input, and it has to determine the nature of instruction and generate appropriate control signals. Figure 1 displays fields of I, J and R type instructions.

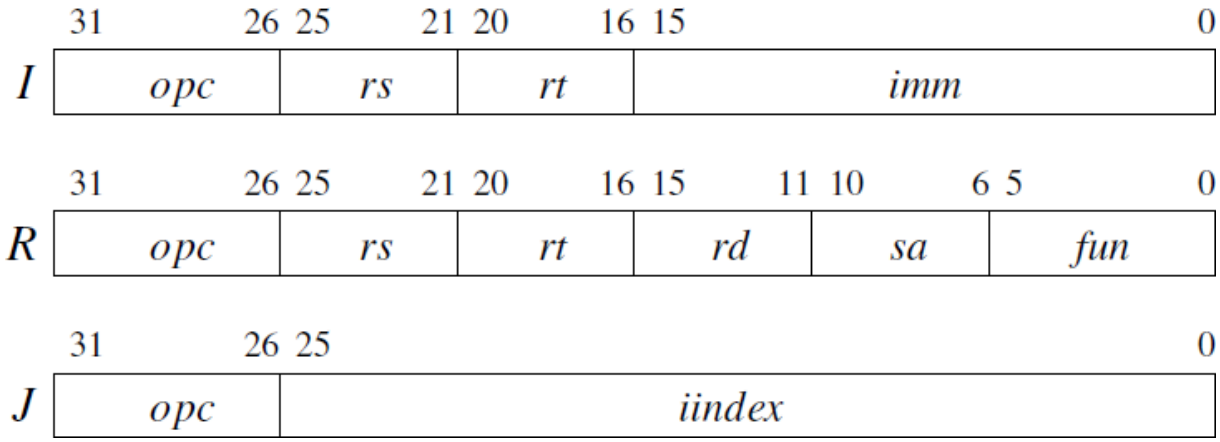

Figure 1

The left most 6 bits, called OPC bits determine what instruction needs to be executed (For R type instructions, opc is 000000 and the last 6 bits called fun determine what instruction needs to be executed).

RS is 5-bit long field and specifies address of register from which left operand is read.

RT is 5-bit long field. In case of R type instruction, RT is address of register from which right operand is taken. In case of I type instruction, RT is address of register where the result of the operation is saved.

RD is 5-bit long field of R type instruction. It is address of destination register. The result of the operation is saved in rd register.

Imm is 16-bit long field of I type instruction. In case of I type instruction, the right operand is an immediate and that immediate is represented with imm field.

SA is 5-bit long field of R type instruction. SA has meaning in shift instructions and it specifies shift amount.

IINDEX is 26-bit long field of J type instruction and it helps us calculate address of instruction where the PC needs to jump.

You are asked to write module IDecoder.

Input:

Instruction – 32-bit long input signal. That is the instruction that needs to be executed

Outputs:

- Related to ALU:
  1) Af – 4-bit control signal that determines what the alu should perform
  2) I – 1-bit signal that tells ALU whether the second operand is immediate or taken from register
  3) ALU_MUX_SEL – 1-bit multiplexer select signal. The second operand of ALU instructions can be either immediate or taken from register. We need a multiplexer before the input of the second ALU operand to choose between immediate fields of instruction and the value read from general purpose registers. If we have Rtype instruction, the second ALU operand is read from registers. Otherwise, it is immediate
- Related to General Purpose Registers:
  1) Cad – 5-bit long signal that determines address of register where the result needs to be stored.
  2) GP_WE – 1-bit long signal. This is general purpose register write enable. The processor updates general purpose registers if and only if GP_WE is 1.
  3) GP_MUX_SEL – 2-bit long multiplexer select signal. Data that should be written to register at address cad can come from 4 different places: from ALU, from MEMORY, from SHIFTER, and from Program Counter (PC). GP_MUX_SEL chooses which of those 4 data is multiplexed and delivered to general purpose registers
- Related to BCE:
  1) Bf – 4-bit long control signal that determines which condition should be tested by BCE.
- Related to MEMORY:
  1) DM_WE – 1-bit signal. This is MEMORY write enable signal. The processor updates MEMORY if and only if DM_WE is 1.
- Related to SHIFTER:
  1) Shift_type – 3-bit long control signal that specifies the type of shift.
- Related to Program Counter (PC):
  1) PC_MUX_Select – 2-bit long multiplexer select signal. PC calculates the address of the next instruction that needs to be executed. PC_MUX_Select chooses between 4 different possible values of next instruction address.

     If we have jump from register (jr) or jump and link from register (jalr) instruction, the addres of next instruction is read from general purpose register.

If we have branch instruction, the address of the next instruction is calculated by adding current PC and sign extended branch distance.

If we have jump (j) or jump and link (jal) instruction, the next instruction is calculated based on jump index and current PC.

PC_MUX_Select chooses between these 4 values.

LAB TASKS

1) Implementation: Write a single module with the abovementioned inputs and outputs. Your module should be able to calculate the appropriate values of the outputs based on the instruction.
2) Simulation & Verification: You are REQUIRED to write a testbench and test if your design generates correct outputs for the following input instructions:

```
initial begin
////////////////// TEST ITYPE INSTRUCTIONS
instruction = 32'b10001100100001010000000000000100; // LW
#10;
instruction = 32'b10101100100001010000000000000100; // SW
#10;

instruction = 32'b00100000100001010000000000000100;  // ADDI
#10;
instruction = 32'b00100100100001010000000000000100; // ADDIU
#10;

instruction = 32'b00101000100001010000000000000100; // SUBI
#10;
instruction = 32'b00101100100001010000000000000100;  // SUBIU
#10;
```

```
instruction = 32'b00110000100001010000000000000100;  // ANDI
#10;
instruction = 32'b00110100100001010000000000000100; // ORI
#10;
instruction = 32'b00111000100001010000000000000100; // XORI
#10;
instruction = 32'b00111100100001010000000000000100; // LUI
#10;



//////BRANCH

instruction = 32'b00000100100000000000000000000100; // bltz
#10;
instruction = 32'b00000100100000001000000000000100; // bGEz
#10;
instruction = 32'b00010000100000001000000000000100; // beq
#10;
instruction = 32'b00010100100000001000000000000100; // bne
#10;
instruction = 32'b00011000100000000000000000000100; // blez
#10;
instruction = 32'b00011100100000000000000000000100; // bgtz
#10;

////////////////// TEST RTYPE
instruction = 32'b00000000100001010010000001000000;  // SLL
```

```
#10;
instruction = 32'b00000000100001010010000001000010; // SRL
#10;
instruction = 32'b00000000100001010010000001000011; // SRA
#10;

instruction = 32'b00000000100001010010000000000100; // SLLV
#10;
instruction = 32'b00000000100001010010000000000110; // SRLV
#10;
instruction = 32'b00000000100001010010000000000111; // SRAV
#10;

instruction = 32'b00000000100001010010000000100000; // ADD
#10;
instruction = 32'b00000000100001010010000000100001;  // ADDU
#10;
instruction = 32'b00000000100001010010000000100010; // SUB
#10;
instruction = 32'b00000000100001010010000000100011; // SUBU
#10;

instruction = 32'b00000000100001010010000000100100; // AND
#10;
instruction = 32'b00000000100001010010000000100101; // OR
#10;
instruction = 32'b00000000100001010010000000100110; // XOR
#10;
```

```
instruction = 32'b00000000100001010010000000100111; // NOR
#10;

instruction = 32'b00000000100001010010000000101010; // SLT
#10;
instruction = 32'b00000000100001010010000000101011; // SLTU
#10;

instruction = 32'b00000000100001010010000000001000; // JR
#10;
instruction = 32'b00000000100001010010000000001001; // JALR
#10;

////////////// JTYPE
instruction = 32'b00001000000000000000000000001001; // J
#10;
instruction = 32'b00001100000000000000000000001001; // JAL
#10;
```

Our advice is to review Chapter 8 in Pr. Wolfgang's book (One that you used last semester). That chapter would help you understand how to calculate corresponding values for the outputs.

Good Luck!