UNC CHARLOTTE **ECE 4146/5146 Introduction to VHDL**

Microcontroller design

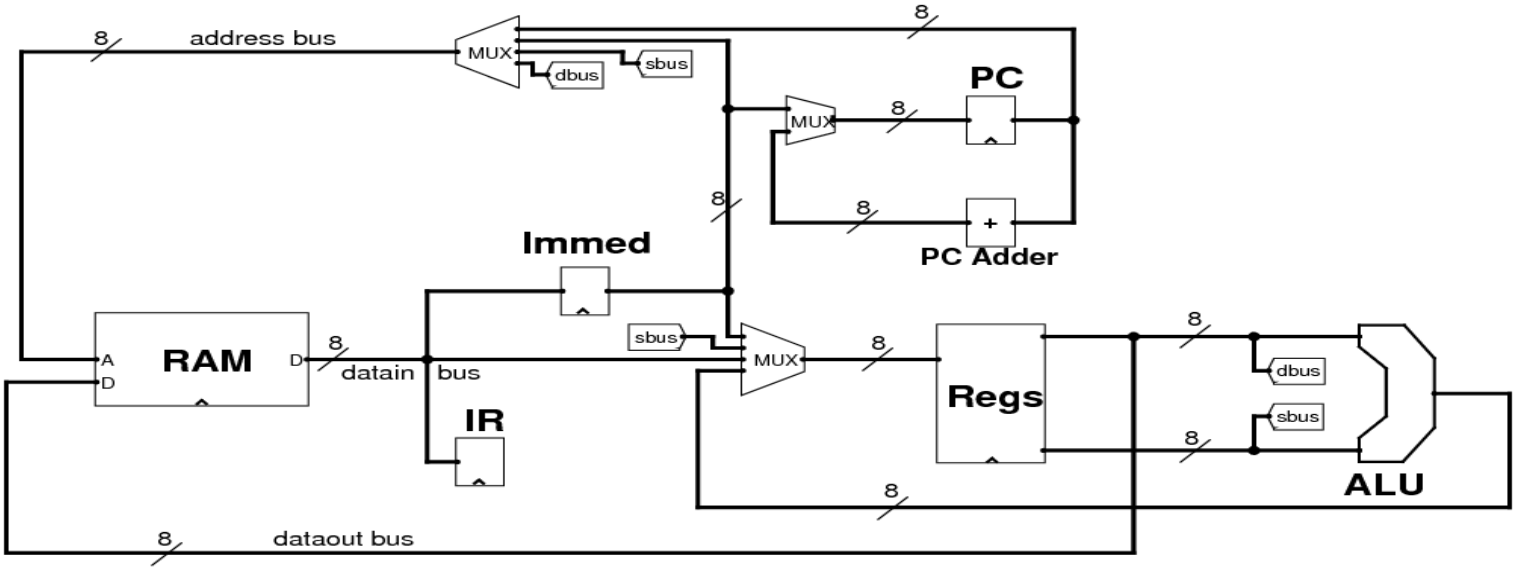
Introduction

The purpose of this project is to design, simulate, and synthesize a behavioral embedded microcontroller. We will design a microcontroller that is compact, capable, and cost-effective fully embedded 8-bit RISC microcontroller core optimized for the many FPGA families.
In this design, the focus is to bring the most significant advantages that a processor can offer to a design environment at minimum cost. For this reason, these small processors have been considered to be "Programmable State Machines" and are referred to as "PSM". A PSM, like any other processor, will execute a program. A program is formed by a set of instructions that are defined by the user and held in a memory. Each instruction is encoded into a machine code.

Architecture

Features of the target CPU are as follows:

- The CPU has an 8-bit data bus and an 8-bit address bus, so it can support 256 bytes of memory to hold both instructions and data.
- Internally, there are four 8-bit registers, R0 to R3, an Instruction Register IR, the Program Counter PC, and an 8-bit immediate register that holds immediate values.
- The ALU is the same one that we designed this week. It performs the four operations AND, OR, ADD, SUB, on two 8-bit values, and supports signed ADDs and SUBs.
- The CPU is a load/store architecture: data must be brought into registers for manipulation, as the ALU only reads from and writes back to the registers.
- The ALU operations have two operands: one register is a source register, and the second register is both source and destination register, i.e. destination register = destination register OP source register. (As you completed in task 2).
- All the jump operations perform absolute jumps; there are no PC-relative branches. There are conditional jumps based on the zero-ness or negativity of the destination register, as well as a "jump always" instruction.
- The *dbus* and *sbus* labels indicate the lines coming out from the register file which hold the value of the destination and source registers.
- Note the data loop involving the registers and the ALU, whose output can only go back into a register.
- The dataout bus is only connected to the *dbus* line, so the only value which can be written to memory is the destination register.
- Also note that there are only 3 multiplexors:
    - The address bus multiplexor can get a memory address from the PC, the immediate register (for direct addressing), or from the source or destination registers (for register indirect addressing).
    - The PC multiplexor either lets the PC increment, or jump to the value in the immediate register.
    - The multiplexor in front of the registers determines where a register write comes from: the ALU, the immediate register, another register or the data bus.
- The following diagram shows the data paths in the CPU:



Instruction Set

- Half of the instructions in the instruction set fit into one byte:

Op1	Op2	Rd	Rs
2	2	2	2

- These instructions are identified by a 0 in the most-significant bit in the instruction, i.e. *op1* = 0X.
- The 4 bits of opcode are split into *op1* and *op2*: details are given below.
- *Rd* is the destination register, and *Rs* is the source register.
- The other half of the instruction set are two-byte instructions. The first byte has the same format as above, and it is followed by an 8-bit constant or immediate value:

Op1	Op2	Rd	Rs	Immediate
2	2	2	2	8

- These two-byte instructions are identified by a 1 in the most-significant bit in the instruction, i.e. *op1* = 1X.
- With 4 operation bits, there are 16 instructions:

op1	**op2**	**Mnemonic**	**Purpose**
00	00	AND Rd, Rs	Rd = Rd AND Rs
00	01	OR Rd, Rs	Rd = Rd OR Rs
00	10	ADD Rd, Rs	Rd = Rd + Rs
00	11	SUB Rd, Rs	Rd = Rd - Rs
01	00	LW Rd, (Rs)	Rd = Mem[Rs]
01	01	SW Rd, (Rs)	Mem[Rs] = Rd
01	10	MOV Rd, Rs	Rd = Rs
01	11	NOP	Do nothing
10	00	JEQ Rd, immed	PC = immed if Rd == 0
10	01	JNE Rd, immed	PC = immed if Rd != 0
10	10	JGT Rd, immed	PC = immed if Rd > 0
10	11	JLT Rd, immed	PC = immed if Rd < 0

11	00	LW Rd, immed	Rd = Mem[immed]
11	01	SW Rd, immed	Mem[immed] = Rd

- Note the regularity of the ALU operations and the jump operations: we can feed the op2 bits directly into the ALU, and use op2 to control the branch decision.
- The rest of the instruction set is less regular, which will require special decoding for certain of the 16 instructions.

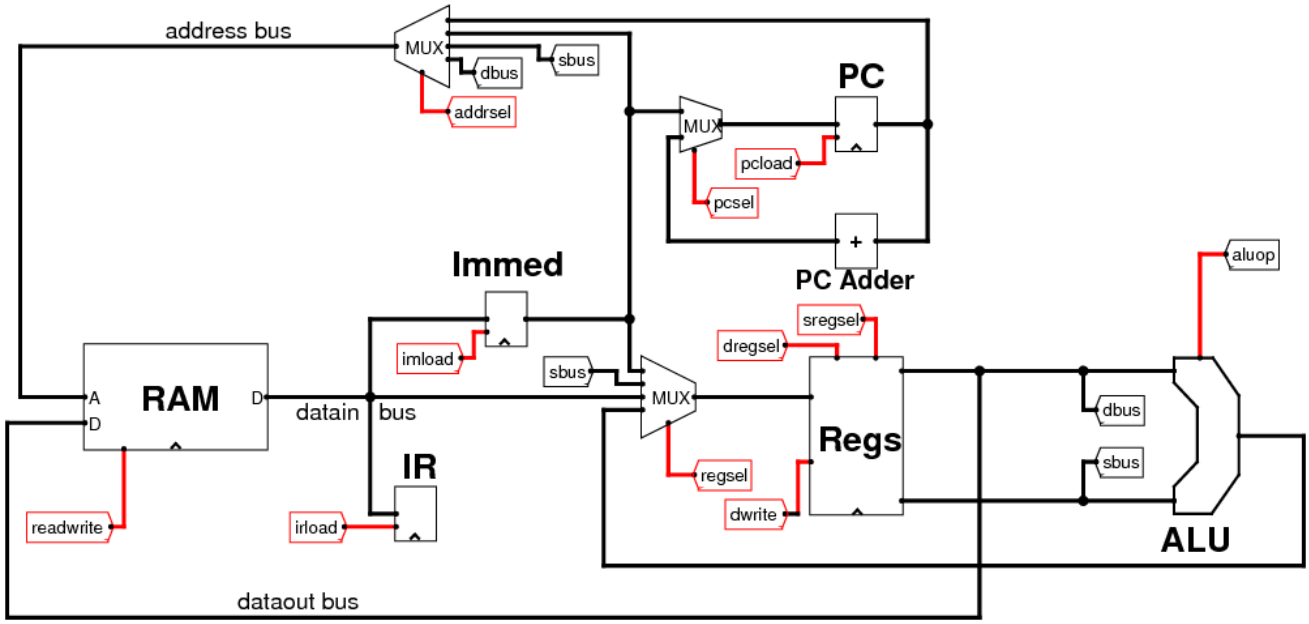
Instruction Phases

- The CPU internally has three stages for the execution of each instruction.
- On stage 1, the instruction is fetched from memory and stored in the Instruction Register.
- On stage 2, if the fetched instruction is a two-byte instruction, the second byte is fetched from memory and stored in the Immediate Register. For one-byte instructions, nothing occurs in stage 2.
- On stage 3, everything else is done as required, which can include:
  - An ALU operation, reading from two registers.
  - A jump decision which updates the PC.
  - A register write.
  - A read from a memory location.
  - A write to a memory location.

- After stage 3, the CPU starts the next instruction in stage 1.

CPU Control Lines

Below is the main controller diagram with the control lines:



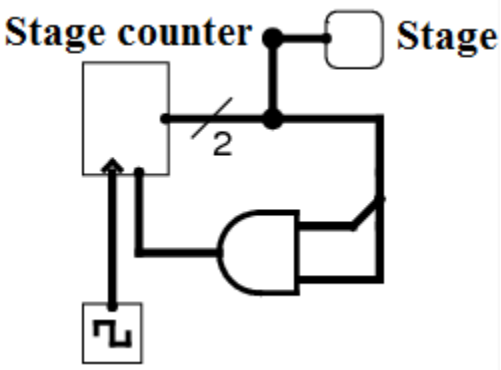
- There are several 1-bit control lines:
  - *pcsel*, increment PC or load the jump value from the Immediate Register.

- *pcload*, load the PC with a new value, or do not load a new value.
- *irload*, load the Instruction Register with a new instruction.
- *imload*, load the Immediate Register with a new value.
- *readwrite*, read from memory, or write to memory.
- *dwrite*, write a value back to a register, or do not write a value.

- There are also several 2-bit control lines:
  - *addrsel*, select an address from the PC, the Immediate Register, the source register or the destination register.
  - *regsel*, select a value to write to a register from the Immediate Register, another register, the data bus or from the ALU.
  - *dregsel* and *sregsel*, select two registers whose values are sent to the ALU.

- aluop, is a 3 bit control pin that control the operation of the ALU.

- The stage counter is used to output the current stage of execution. This is achieved with a simple 2-bit counter which is controlled by the clock cycle, and which outputs a 2-bit *stage* line.



- The stage goes from 0,1,2 and 3. We do not need stage 3, so the two bits of the stage line can be ANDed. When both are true, this resets the counter back to zero.
- For the rest of the decode logic, we need to look at what needs to be performed for the various stages of the CPU, and what needs to be performed for each specific instruction.

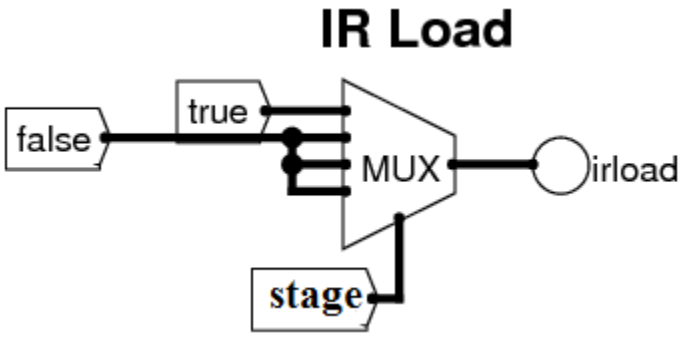
Reset

- Reset PC to zero.

pcsel	pcload	irload	imload	rw	dwrite	jumpsel	addrsel	regsel	dreg	sreg	aluop
0	0	0	0	0	0	0	0	0	0	0	0

Stage Zero (Instruction Fetch)

- On stage zero, the PC's value must be placed on the address bus, so the *addrsel* line must be 0. The *irload* line needs to be 1 so that the IR is loaded from the *datain* bus. Finally, the PC must be incremented in case we need to fetch an immediate value in stage 1.



- All of this can be done using multiplexors which output different values depending on the current phase. Here is the control logic for the *irload* line.
- We only need to load the IR on stage 0, so we can wire true to the 0 input of the *irload* multiplexor, and false to the other inputs. **Note:** input 11 (i.e. decimal 3) to the multiplexor is never used. Another way to look at each stage is the value which needs to be set for each control line, for each instruction.
- For stage zero, these control line values can be set for all instructions:

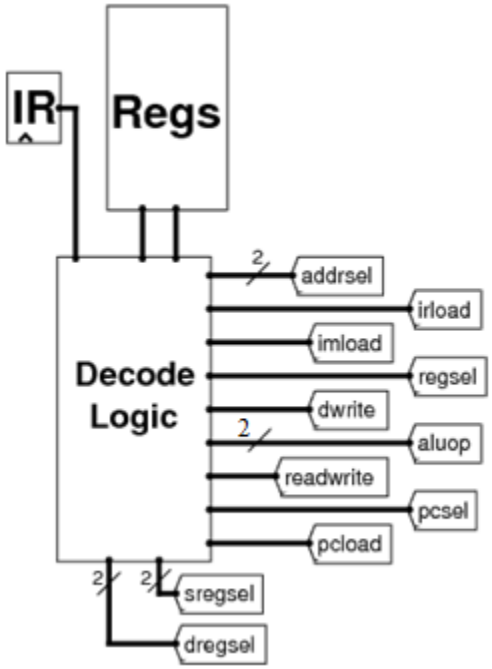
pcsel	pcload	irload	imload	rw	dwrite	jumpsel	addrsel	regsel	dreg	sreg	aluop
1	1	1	0	0	0	0	0	x	x	x	x

Table 1

- 'x' stands for any value, i.e. accept any opcode value, output any control line value.

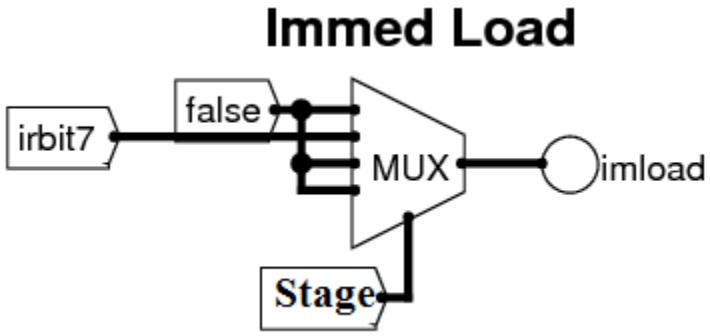
Stage One

- The values for control lines are generated by the Decode Logic, which gets as input the value from the Instruction Register, and the zero & negative lines of the destination register.

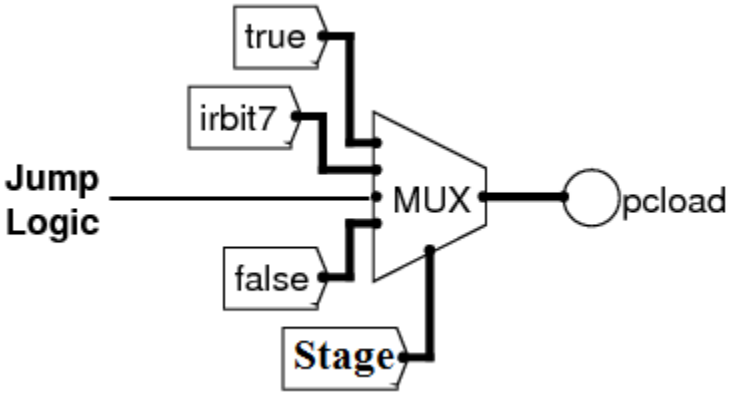


Inside the decode logic

- Inside the Decode Logic block, the value from the Instruction Register is split into individual lines *irbit4*, *irbit5*, *irbit6* and *irbit7*. *op1* and *op2* are split out, with *op2* exported as *aluop*. The 4 opcode bits from the instruction are split out as the *op1op2* line.
- Several of the bits from the instruction register value are wired directly to these 2-bit outputs: *dregsel*, *sregsel* and *aluop*.
- On stage 1, we need to load the Immediate Register with a value from memory if the *irbit7* from the IR is true. The PC's value must be placed on the address bus, so the *addrsel* line must be 0. The *imload* line needs to be 1 so that the Immediate Register is loaded from the *datain* bus. Finally, the PC must be incremented so that we are ready to fetch the next instruction on the next stage 0.



- The *imload* logic is shown above. It is very similar to the *irload* logic, but this time an enable value is output only on stage 1, and only if the *irbit7* is set.



- Some of the *pcload* logic is shown above. The PC is always incremented at stage 0. It is incremented at stage 1 if *irbit7* is set, i.e. a two-byte instruction. Finally, the PC can be loaded with an immediate value in stage 2 if we are performing a jump instruction and the jump test is true.
- We can tabulate the values of the control lines for stage 1. This time, what is output depends on the top bit of the *op1* value:

op1	op2	instruct	pcsel	pcload	irload	imload	rw	dwrite	jumpsel	addrsel	regsel	dreg	sreg	aluop
0x	xx	all	x	0	0	0	0	0	0	0	x	x	x	x
1x	xx	all	1	1	0	1	0	0	0	0	x	x	x	x

Stage Two

Here, the values of the control lines depend heavily on what specific instruction we are performing. Following table shows control line outputs depending on the instruction:

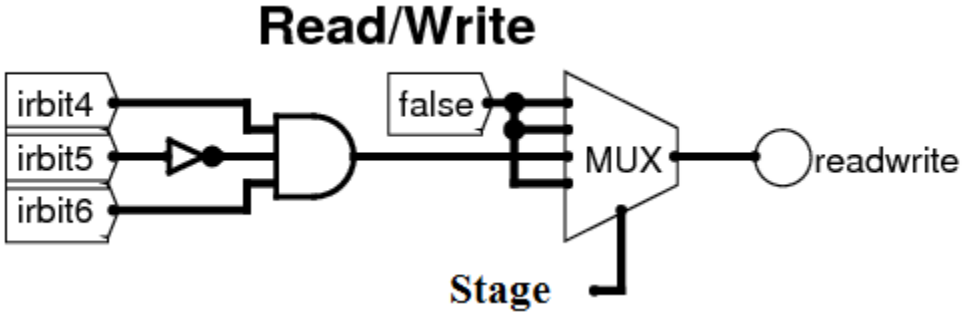
op1	op2	instruct	pcsel	pcload	irload	imload	rw	dwrite	addrsel	regsel	dreg	sreg	aluop
00	00	AND Rd, Rs	x	0	0	0	0	1	x	3	Rd	Rs	op2
00	01	OR Rd, Rs	x	0	0	0	0	1	x	3	Rd	Rs	op2
00	10	ADD Rd, Rs	x	0	0	0	0	1	x	3	Rd	Rs	op2
00	11	SUB Rd, Rs	x	0	0	0	0	1	x	3	Rd	Rs	op2
01	00	LW Rd, (Rs)	x	0	0	0	0	1	2	2	Rd	Rs	x
01	01	SW Rd, (Rs)	x	0	0	0	1	0	3	x	Rd	Rs	x
01	10	MOV Rd, Rs	x	0	0	0	0	1	x	1	Rd	Rs	x
01	11	NOP	x	0	0	0	0	0	x	x	x	x	x
10	00	JEQ Rd, immed	0	j	0	0	0	0	x	x	Rd	x	op2
10	01	JNE Rd, immed	0	j	0	0	0	0	x	x	Rd	x	op2
10	10	JGT Rd, immed	0	j	0	0	0	0	x	x	Rd	x	op2
10	11	JLT Rd, immed	0	j	0	0	0	0	x	x	Rd	x	op2
11	00	LW Rd, immed	x	0	0	0	0	1	1	2	Rd	x	x
11	01	SW Rd, immed	x	0	0	0	1	0	1	x	Rd	x	x
11	10	LI Rd, immed	x	0	0	0	0	1	x	0	Rd	x	x
11	11	JMP immed	0	1	0	0	0	0	x	x	x	x	x

Table 2

- From the table above, the ALU instructions (*op1*=00) and the jump instructions (*op1*=10) are regular. All the *op1*=1x instructions use the Immediate Register, while the *op1*=0x instructions do not.
- We can always tie *dregsel* to *Rd* from the instruction, and the same goes for *sregsel* = *Rs* and *aluop* = *op2*. And *irload* and *imload* are always 0 for stage 2.
- With the remaining control lines, the regularities cease.

Read/Write Logic (rw to memory)

- The read/write line out to memory only needs to be enabled when we are performing SW (store word) operations, and only in stage 2.
- The *op1op2* values for the two SW instructions are 0101 and 1101, so we can treat this as x101, and set *readwrite* true when *irbit6* is on, *irbit5* is off and *irbit4* is on.



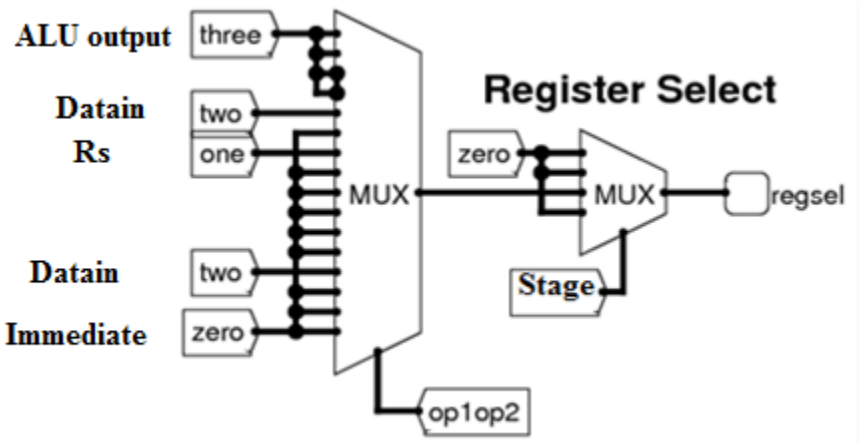
Register Select Logic

- The *regsel* line selects the input to be written into the destination register in register file. Data value can come from any of the following:

00 (zero)	Immediate Register
01 (one)	*sbus*, i.e. the source register
10 (two)	datain bus
11 (three)	ALU output

Table 3

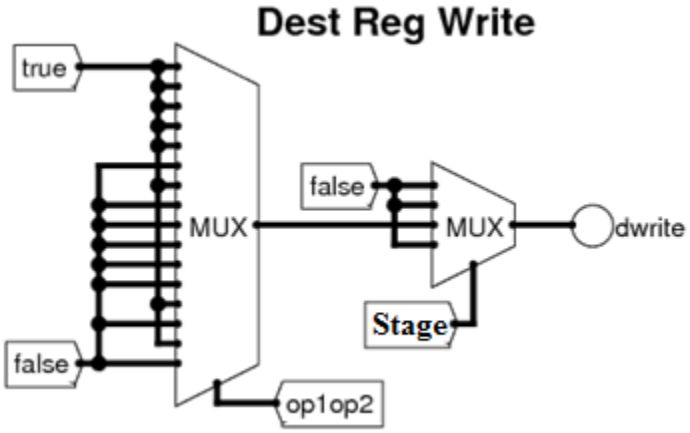
- Consulting Table 3 in the stage Two subsection, all the ALU instructions set *regsel* to 3, but apart from that there is no simple logical rule to output all the possible values.
- Also, for the *regsel* 'x' values in the table, we can choose to output any value, as the register would not be loaded on these instructions.
- One multiplexor for the stage of operation, and a second multiplexor for the instruction's opcode, i.e. *op1op2* is used to select the value to output.



- Each of the 16 inputs to the big multiplexor sets a *regsel* value for a specific instruction based on the *op1op2* value, and this only gets out during stage 2. Otherwise, *regsel* is set to zero.

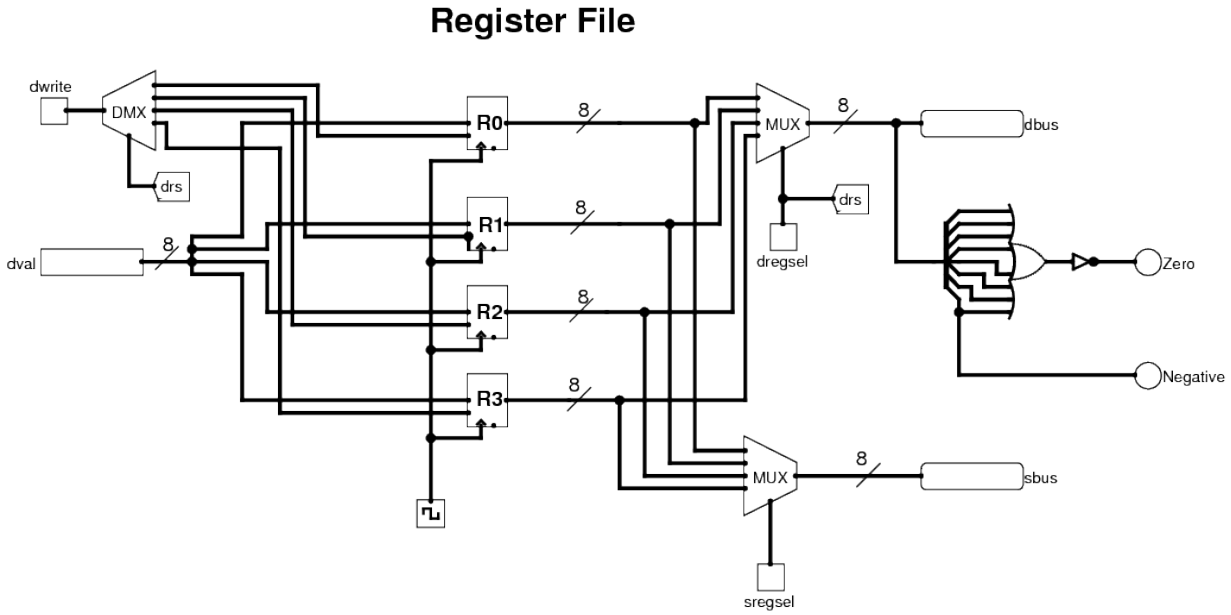
Destination Register Write Logic

- Following on from *regsel*, we need to control *dwrite*, i.e. when the destination register gets written.
- This can occur only in phase 2, and again there is no simple rule that allows us to hardwire the value with simple gates: see the *dwrite* column in Table 3.



- Again, we can use a 16-input multiplexor, with *op1op2* to choose the correct *dwrite* value to output on phase 2.

The Register File



- We have four data registers in this architecture. Each register's value is sent to two multiplexors on the right.
- The top multiplexor selects one of the register values based on the *dregsel* control line (drs), and outputs one value on the *dbus* line to the ALU.
- Similarly, the bottom multiplexor selects one of the register values based on the *sregsel* control line, and outputs one value on the *sbus* line to the ALU.
- There is also logic to test if the *dbus* value is negative. The eight bits on the *dbus* are split out, and the most significant bit is the *Negative* output. This implies that we use twos-complement representation for signed values.
- All eight bits of the *dbus* are ORed together and then negated.

- When the *dbus* is zero, all bits are zero. The OR output is zero, and thus the negated output on the *Zero* line is true.
- On the left, each register takes three inputs:
  - A value to possibly load. All registers are hard-wired to the input *dval* which holds the value to load. (Dval is connected to the outout of multiplexer with *regsel ctrl pin.)*
  - The clock pulse, which tells the registers when to load.
  - A write enable signal. Only when the write enable signal is 1 can a register overwrite its old value.
- The de-multiplexor at the top-left takes as input *dwrite* (i.e. the write signal), and based on the *dregsel* value, selects which register to send the *dwrite* signal to. All other registers will get a write signal of 0, i.e. not to perform a write operation.

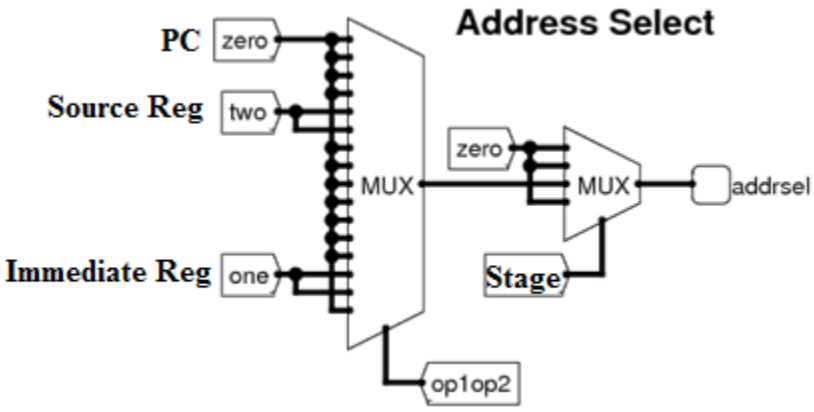
Address Select Logic

- When we want to read data from main memory, the address we want to read from can be selected from these inputs:

00 (zero)	Program Counter
01 (one)	Immediate Register
10 (two)	*sbus*, i.e the source register
11 (three)	*dbus*, i.e the source register

Table 4

- The decode logic needs to output a value for *addrsel* which selects the correct address to assert on the address bus for each instruction during stage 0, stage 1 and stage 2.
- As with the previous two control lines, there is no simple logic to produce the value on this line based on the instruction opcode, so we resort again to a 16-input multiplexor.



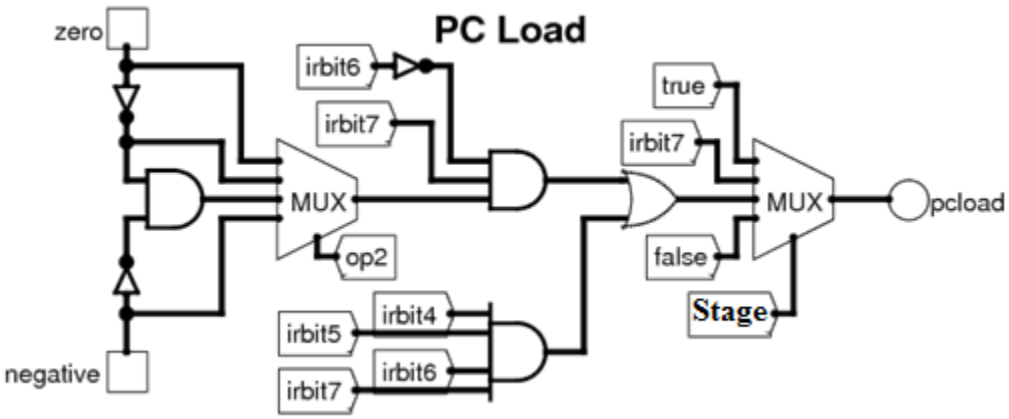
Jump Logic

- The signal *pcload* which determines when the Program Counter is updated.
- One of the jump instructions, *op1op2* = 1111, always sets the PC to the Immediate Register.
- The other jump instructions only set PC to Immediate Register when a specific test is true:

op1op2	Test	Zero	Negative
1000 (JEQ)	Rd == 0	1	x
1001 (JNE)	Rd != 0	0	x
1010 (JGT)	Rd > 0	0	0
1011 (JLT)	Rd < 0	x	1

Table 5

- Here is the logic to set *pcload* for all phases:



- The OR gate which connects to the stage 2 input of the right-hand multiplexor chooses either a successful jump test (top-left input) OR a "jump always" instruction (bottom-right input).
- The "jump always" instruction is *op1op2* = 1111, so a 4-input AND gate is used to select for only this opcode.
- The left-hand multiplexor outputs true when the EQ/NE/GT/LT decision is true, for any instruction (not just a jump instruction). Compare the gate logic here with the truth table above.
- But we must make sure that we only output the EQ/NE/GT/LT decision on jump instructions, i.e. when *op1op2* = 10xx. Thus, we use the 3-way AND gate and also use the *irbit6* and *irbit7* lines as inputs.

Entities

ALU

```
entity ALU is
        port(
        A: in std_logic_vector(7 downto 0);
        B: in std_logic_vector(7 downto 0);
        aluop: in std_logic_vector( 1 downto 0);
        result: out std_logic_vector(7 downto 0)
):
end entity;
```

Memory

```
entity memory is
        port(
        address: in std_logic_vector(7 downto 0);
        dataout: in std_logic_vector(7 downto 0);
        datain: out std_logic_vector(7 downto 0);
        readwrite: in std_logic;
        clk: in std_logic;
        rst: in std_logic
);
end entity;
```

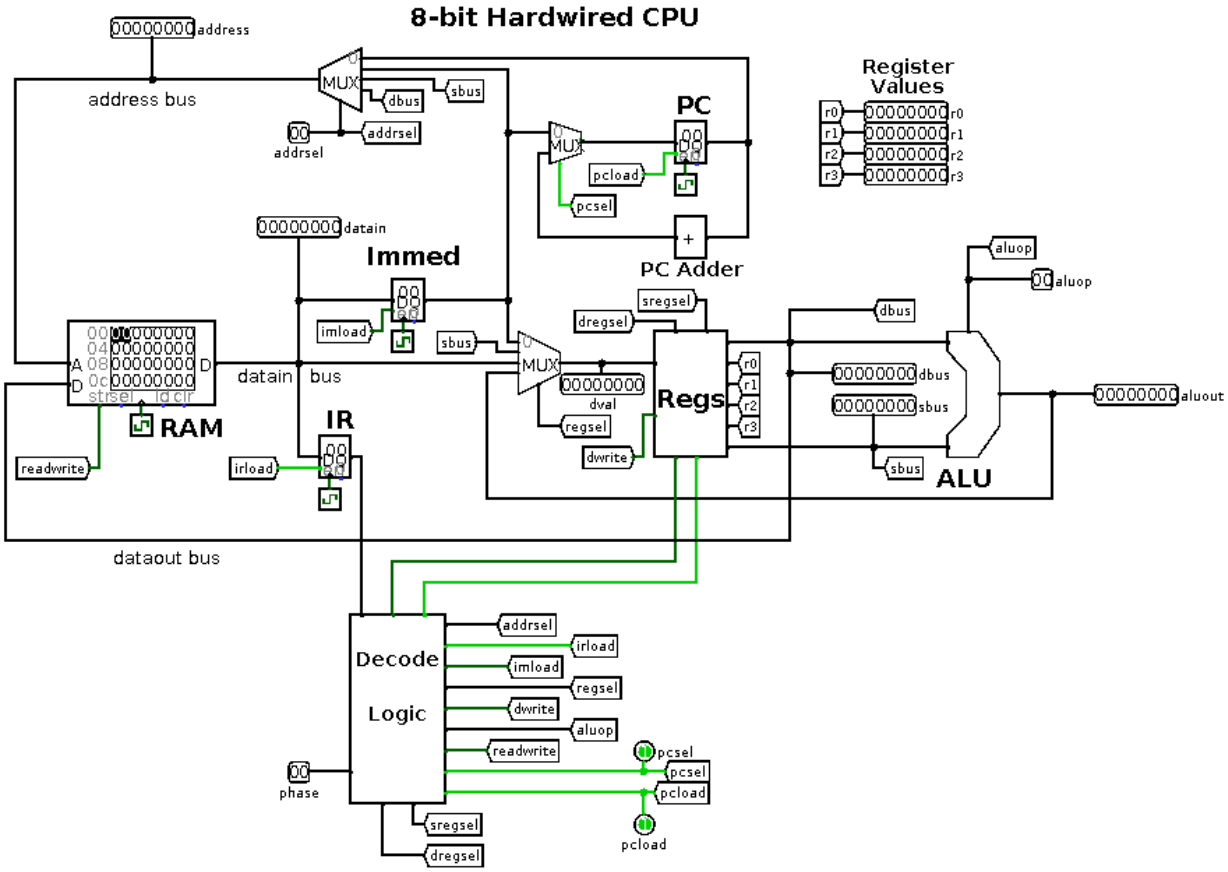
Decode Logic

```
entity decode_logic is
        port(
        instruction: in std_logic_vector(7 downto 0); --instruction input from IR
        zero : in std_logic; --zero line from Register File
        negative: in std_logic; --negative from Register file
        addrsel : out std_logic_vector(1 downto 0);
        irload : out std_logic;
        imload : out std_logic;
        regsel: out std_logic;
        dwrite: out std_logic;
        aluop: out std_logic_vector(2 downto 0);
        readwrite: out std_logic;
        pcsel: out std_logic;
        pcload: out std_logic;
        sregsel: out std_logic_vector(1 downto 0);
        dregsel: out std_logic_vector(1 downto 0);
);
end entity;
```

Stage Counter

```
entity phase_counter is
        port (
        clk: in std_logic;
        rst: in std_logic;
        stage: out std_logic_vector(1 downto 0)
);
end stage_counter;
```

UNC CHARLOTTE **ECE 4146/5146 Introduction to VHDL**



Test Program

- Consider an example program for your CPU.
- In memory starting at location 0x80 is a list of 8-bit numbers; the last number in the list is 0.
- We want a program to sum the numbers, store the result into memory location 0x40, and loop indefinitely after that.
- We have 4 registers to use. They are allocated as follows:
  - R0 holds the pointer to the next number to add.
  - R1 holds the running sum.
  - R2 holds the next number to add to the running sum.
  - R3 is used as a temporary register.
- The machine code, is shown in the hex values to put into memory starting at location 0:

LI R1,0x00	e4 00
LI R0,0x80	e0 80
LW R2, (R0)	48
JEQ R2, end	88 0d
ADD R1, R2	26
LI R3, 0x01	ec 01
ADD R0, R3	23
JMP loop	ff 04

SW R1, 0x40	d4 40
JMP inf	ff 0f

Deliverables

- Using the description provided, implement the CPU in VHDL. Use the entities, as defined here to implement the important design blocks. In addition to these entities described here, add design elements as required.
- In addition to the design, write a detailed report which describes the implementation of each module and any extra design elements that you might have added. Also mention any design, implementation considerations and issues that you ran into for this design.
- Each module must first be designed and verified with a test bench individually.
- It must be noted that you are free to decide on the implementation of a module and the implementation provided in the document is for providing an understanding.
- For the ALU, you can reuse your already implemented design of ALU and modify it for this lab.

- Your report need to include following steps:
    - With the memory loaded with the above data values, start the program.
    - Show the stages of operation. Watch the IR get loaded with an instruction.
    - Show the Immediate Register get loaded with a value.
    - On the LW instruction, watch as the *sbus* value is selected to be placed on the address bus, and the datain value is written to the destination register.
    - On ALU instructions, show the *sbus* and *dbus* values, the *aluop*, and the result which is written back into the destination register.
    - On the JEQ instruction, show the value of N and Z into the Decode Logic, and the resulting *pcsel* and *pcload* values.