Lab 9 - Control Unit

What to hand in

1. Answers to questions in a single .txt file. Named Team"NUMBER"_Lab9.txt
2. Control unit vhdl file

Overview

Now that we have built a register file and an ALU, there are only a few major components we are missing for a full Processor (ignoring random registers, multiplexors, and certain memory devices). The last major component you will build is the control unit. We will start with a basic Control Unit that will only be able to handle R-type instructions. An R-type instruction means that the operation will be register-to-register. R-type instructions that we will implement today are: add, sub, and, or, and xor. We will set up the control unit so that when it detects one of these instructions, the correct control flags will be set. As you move along in the project, you will have to add more functionality to the control unit in order for it to handle more instructions.

Control Unit

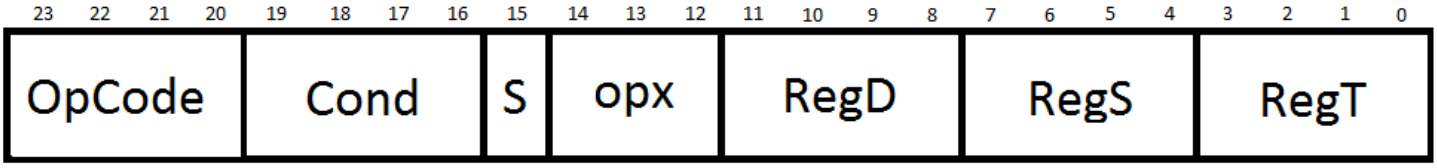
A control unit acts as the interpreter for a processor. It decodes an instruction to determine what needs to happen. This means that the general inputs to a control unit will be the instructions and any external control flags. The NZVC flags we created in the alu are considered external control flags and should be fed into the control unit. Other inputs include the clock, reset, and a flag from the memory system. These inputs will then be decoded and change the value of the outputs. The outputs of the control unit dictate when/if something in the processor happens. The a_inv, b_inv, alu_op, and write_enable bits are examples of control flags.

For example, say I want to do the instruction, add r2, r2, r3. The first part of instruction execution is instruction fetch. So the memory unit will the load the instruction into the instruction register (IR). The second part is then instruction decode. This is where the instruction will be broken down into subparts, and those subparts will be sent to the control unit. During this stage, the control unit will determine what control flags to enable or disable. For this instruction, the control unit will set the a_inv, b_inv, alu_op, and write enable control flags as these flags are important to an add. After the flags are set, the next stage starts. During the third part the ALU performs operations, and program counter (PC) jumps/branches occur. For the instruction above, all we would care about is the ALU operation. So after the ALU performs addition, we then enter stage 4, which is the memory stage, which doesn't matter to an add. Finally, we enter the last stage, where we write the value of R2+R3 to the register file.

Instruction structure

One important part we have yet to talk about is the instruction breakdown. Given an instruction, add r2 r2 r3, we would have to translate this into machine code in order to send it to the processor. For this project, we will give you a designed Instruction Set Architecture, that mimics the NIOS II ISA you were using earlier in the lab.

For our processor, we use 24 bit instructions. The bit values for all R-type instructions is shown below:



The OpCode is what determines the type of instruction. The first two bits of it will say whether it is a R(00), D(01), B(10), or J(11) type instruction. The next two bits determine what type of R/D/B/J. For example, if the instruction is add/sub/or/xor/and, then the OpCode will be 0000. The Cond will determine what type of condition has to be met for the instruction to execute. The S bit will be used to set the NCVZ flags based on the current instruction. You will not have to worry about the Cond or S bits until Part IV of the project. The opx, which is only in the R-type instructions, determines the type of operation to take place in the ALU. The values of opx are; 111 for AND, 110 for OR, 101 for XOR, 100 for ADD, 011 for SUB. Finally, the remaining 12 bits determine what registers the instruction will use. RegD is the target register while RegS and RegT are to source registers. These will match up to the inputs of your register file and are not important to the control unit.

To put it all together, the instruction add r2 r2 r3 will be broken down into the following machine code: 0000 0000 0 100 0010 0010 0011. You can use this for testing of your control unit later.

Part 1 questions

1. What are control flags?
2. Which control flags are set during an Add instruction (Think of flags that control the ALU and Register File)?

Sequential control unit

Our control unit will have to be sequential. We only want it to update the values of the control unit at a specific time, the rising edge of the clock. In order to do this we must make it a sequential device. You can think of a control unit as a bunch of registers that separately hold the values of the control flags.

Getting sequential in VHDL

Now, the easiest way to create sequential circuits is in VHDL. The language provides quick and easy ways to turn a circuit into a sequential circuit by adding a clock input and using a few new methods.

The first new method we will talk about is called Process. Since we want to create a sequential circuit that only updates values **IF** the clock is on the rising edge, we must use a process. Process's are written in the ARCHITECTURE of a VHDL file. Each process must have a beginning (BEGIN) and an end (END). An important part of processes in VHDL is that you can include inputs/signals in a processes sensitivity list. This list will make those values the most important part of the process, and will ultimately depend on them. For a sequential circuit, you will want to include the clock at least in this list. See the example below for how a 16 bit register is created.

```
ARCHITECTURE behaivor OF reg16 IS
BEGIN
  PROCESS(clock, reset)    --Set up the process to be sensitive to clock and reset
  BEGIN                    --Start the process
    IF(reset = '1') THEN   --Check if reset is one
      output <= (OTHERS =>'0');      --If so, the output is zero
    ELSIF(rising_edge(Clock)) THEN   --else Check if the clock is on the rising edge
      IF(enable = '1') THEN          --If so, finally check if enable is 1
        output <= data;              --If so, update the value of the register
      END IF;          --All ifs must end
    END IF;            --All ifs must end
  END PROCESS;         --All processes must end
END behavior;          --All behaivors must end
```

Using the above example, we can see how different processes are from normal VHDL files. We can now use if statements to check values instead of determining the logic ourselves! However, processes are not something you should default to. The main problem a lot of VHDL files have is that the code you write, does not always end up the circuit you need. This is caused by using processes and not correctly defining your circuit yourself. Unless you explicitly design the circuit, using gates, you can never be 100% sure what the circuit will end up looking like.

For this lab, the only new thing you need to learn from processes is the basic structure of them and if statements. If statements are handled differently than from C or Java in that you must use the keywords THEN, ELSIF, and END IF. For each IF statement you must have a THEN statement following the condition to check. The same goes for each ELSIF statement and ELSE statements. Finally, you must end each IF statement with an END IF statement. The equality checker is only one equals sign, not two. In the condition check you can use a few different ways to check the value of a std_logic_vector type. The first is complete check. The second is for bit check. The below code shows how to do both.

```
IF(opCode = "0000") THEN
  --do something
ELSIF(opCode(3) = '0' AND opCode(2) = '0' AND opCode(1) = '0' AND opCode(0) =
    '1') THEN
  --do something
END IF;
```

Part 2 questions

1. How do you end an if statement in VHDL?
2. Why should a control unit be sequential?
3. How do you check if the clock is on the rising edge in VHDL?
4. How would you write the following C Code in VHDL?

```
if(x == 0b01)
  y = 0b00;
else if(x == 0b10)
  y = 0b01;
else
  x = 0b00;
```

   Assume that x is of type std_logic_vector(1 downto 0) and y is of type std_logic. 0b01 would be represented as "01" in VHDL

Creating our control unit

The rest of this document will guide you through writing your own control unit. The writing is broken down into multiple parts and you will have to do no guessing as to what should be done. After you get down with creating the control unit, you will have to answer some more questions and then hand in the files listed at the beginning. You should, however, make sure to test your control unit with the .do file located on blackboard. You may have to change some I/O pin names, but there is an expected output that should be matched.

Going to the Library

The first thing we need to do, like every VHDL file, is specify that we want to use the ieee library. We also need to specify that we want to use the std_logic_1164 part of the

library. If you do not know how to do this, then look at any other VHDL document for an example.

Defining an Entity

Now that we have the library we want included, we need to define the inputs and outputs of our control unit. First decide on a name. If you can't think of anything then use controlUnit. After that we must define the ports of the control unit. Using the below table, with types assigned, you should be able to set up the entity declaration. Be forewarned, there are a lot of extra inputs and outputs in there that might not make to much sense at this time, but include them for now.

INPUTS	SIZE
opCode	4
Cond	4
opx	3
S	1
N,C,V,Z	1(each)
mfc	1
clock, reset	1

OUTPUTS	SIZE
alu_op, c_select, y_select	2
rf_write, b_select	1
a_inv, b_inv	1
extend	2
ir_enable, ma_select	1
mem_read, mem_write	1
pc_select, pc_enable, inc_select	1

Once you have the inputs and outputs declared and the entity declaration set up you are ready to move on.

Creating a behavior

Now that we have the entity declared, we need to define it. We do this in the ARCHITECTURE area. You should now how to set this up for use. Your ARCHITECTURE area should look like the below code (go ahead and copy it if you are lazy enough to not write three lines of code).

```
ARCHITECTURE behavior OF controlUnit IS

BEGIN

END behavior;
```

Lost signals

Now that we have set up the ARHCITECTURE, we have to go back and include a few signals. The table below will list all the signals your control unit will need. One of them you will not understand, unless you read up on VHDL your own, and that is okay. Go ahead and copy the below code to where your signals would normally go.

```
signal wmfc: std_logic;
shared variable stage: integer:= 0;
```

These two signals, or rather one signal and one shared variable, will help your control unit know what stage it is in (1-5). The shared variable will keep a running tab on what stage of each instruction we are on. The wmfc signal will be used for initial start up only. Once you have the lost signals back to their rightful place, you are ready for the next step.

It's something of a process

Now we need to actually write the code that will make the control unit operate the way we want it to. The first step is to begin a process. You should add the clock and the reset inputs as the values that the process is sensitive to. You might have questions on how to do this. If you do, then call a friendly TA over to help. If you don't, its because you looked up at "Getting sequential in VHDL" part of this document and copied the line of code from there. Either way, this is a new topic so feel free to ask questions. If you are still lost, the code for this part is included in the next part.

All you should do for this part is write the opening and end statements of your process for now. That is just three lines of code.

When is the control unit suppose to run again?

After we have a process set up, we need to do an encompassing IF statement. This statement should check for when the clock is on the rising edge. As you see below, the code is given to you to write. Every bit of code after this one should be inside that IF statement.

```
BEGIN
  PROCESS(clock, reset)
  BEGIN
    IF(rising_edge(clock)) THEN
    ....
    ....
    END IF;
  END PROCESS;
END behavior;
```

Everything that the processor does will only happen during the rising edge of the clock. This means that every stage will take 1 clock cycle. Ultimately, that means every instruction will take 5 clock cycles. This is terribly inefficient, however, we don't care. The whole point is to understand how this works first, and then how to make it work faster. We can now start wrtiting code that resembles decoding an instruction.

Pushing the reset button

The first thing we need to check when the clock is on the rising edge is what the value of reset is. If it is a 1, that means we should reset the stage to 0 and then start a new instruction. If it is a 0 then we can carry on with our business. The below code is how we check if we need to reset.

```
IF(reset = '1') THEN
  stage := 0;
END IF;
```

You may be wondering why a lot of the code is given to you for this lab. It's not that it's difficult code to write, it's that this is a somewhat difficult topic, and we would rather you know what it is doing, than spend 6 hours and get frustrated at it.

Moving along

The next thing we need to do is update the stage of our processor. We will update the stage every single time as long as the mfc input is 1 or the mwfc is 0. The mfc input is a signal that will come from the memory interface in stage three, but for now we can assume it is 1.

```
IF((mfc = '1' or wmfc = '0')) THEN
  stage := stage mod 5 + 1;
END IF;
```

Stage 1 and 2

Now that we got the basic stuff out of the way, we can finally do some instruction decoding. In order to do this we will set up an IF-ELSIF structure that checks what stage the processor is in. The first two stages will be given to you. These stages should never be modified throughout the project. The code is located on the next page because it is too long to fit the remainder of the page.

```
-- instruction fetch
IF(stage = 1) THEN

   wmfc <= '1';
   alu_op <= "00";
   c_select <= "01";
   y_select <= "00";
   rf_write <= '0';
   b_select <= '0';
   a_inv <= '0';
   b_inv <= '0';
   extend <= "00";
   ir_enable <= '1';
   ma_select <= '1';
   mem_read <= '1';
   mem_write <= '0';
   pc_select <= '1';
   pc_enable <= mfc;
   inc_select <= '0';


-- register load
ELSIF(stage = 2) THEN

   wmfc <= '0';
   ir_enable <= '0';
   mem_read <= '0';
   pc_enable <= '0';

--ALU, branch, jump operation
ELSIF(stage = 3) THEN

--Memory stage
ELSIF(stage = 4) THEN

--Write back stage
ELSIF(stage = 5) THEN

END IF;
```

As you can see, the first stage is simply giving a default value to the cotnrol flags. It also turns on the instruction register(IR), allows for memory read, and turns on the program counter (PC). This is very important as we want to read an instruction from the memory location that is specified by the PC, update the PC, and then load the instruction into the IR. After we do this, we move on to the second stage. During this stage we want to turn off the IR and PC so that we don't update the PC or IR every clock cycle. The next three stages are what you will mostly be modifying for the project.

Stage 3 - ALU Operation

Finally you will have to write some of your own code. Not really. You will simply have to update the values in the code I have written below. As stated above, for this instruction all you have to do is implement R-type instructions. Specifically only Add, Sub, And, Or, and Xor instructions. As a reminder, the OpCode for R-type instructions is 000L. The L there is a zero or a one depending on the type of R-Type instruction. For Add, Sub, AND, OR, and XOR it is a zero. Reference the table below if you are lost. All you need to do is set the correct values of the control flags alu_op and b_inv.

INSTRUCTION	OpCode	opx
and	0000	111
or	0000	110
xor	0000	101
add	0000	100
sub	0000	011
jr	0001	000

```
ELSIF(stage = 3) THEN
  -- R-type instructions
  IF(opCode(3) = '0' AND opCode(2) = '0') THEN
    IF(opCode(1) = '0' AND opCode(0) = '1') THEN
      --This is for JR, just fill in the values for the if statement
    ELSIF(opCode(1) = '0' AND opCode(0) = '0') THEN
      --This is for the other instructions
      IF(opx = "111") THEN
        --AND instruction
        alu_op <= "";
      ELSIF(opx = "110") THEN
        --OR instruction
        alu_op <= "";
      ELSIF(opx = "101") THEN
        --XOR instruction
        alu_op <= "";
      ELSIF(opx = "100") THEN
        --ADD instruction
        alu_op <= "";
      ELSIF(opx = "011") THEN
        --SUB instruction
        alu_op <= "";
        b_inv <= '';
      END IF;
    END IF;
  END IF;
ELSIF(stage = 4) THEN
```

As you can see, all we care about is the alu_op and b_inv flags in stage three.

Stage 4 - Memory operation

For this stage, there isn't much to write. Just copy the code below. R-type instructions have no use of the memory devices.

```
ELSIF(stage = 4) THEN
  -- R-type instructions
  IF(opCode(3) = '0' AND opCode(2) = '0') THEN
    IF(opCode(1) = '0' AND opCode(0) = '1') THEN
      --This is for JR, just fill in the values for the if statement
      --We will have to set some flags here in the future
    END IF;
  END IF;
ELSIF(stage = 5) THEN
```

Stage 5 - Write back

In the last stage of the processor we need to set one control flag for R-type instructions. That flag is the write enable flag, rf_write. All we need to do is set it to one. You can just copy the code below.

```
ELSIF(stage = 5) THEN
  -- R-type instructions
  IF(opCode(3) = '0' AND opCode(2) = '0') THEN
    IF(opCode(1) = '0' AND opCode(0) = '1') THEN
      --This is for JR, just fill in the values for the if statement
    ELSIF(opCode(1) = '0' AND opCode(0) = '0') THEN
      rf_write <= '1';
    END IF;
  END IF;
END IF;
```

Control unit built and testing

Congratulations, you have successfully built a control unit that can decode 5 different instructions and set the corresponding flags! As for testing, you can download a .do file from piazza that will test an Add instruction and a Sub instruction. You should see the correct flags being set in the correct stages if you built yours correctly. You should test more instructions on your own though. The control unit is the most important part of the processor and needs to work. **Note:** you may need to modify some of the values to fit any naming differences. You should also take the time here to understand how to write the machine code for instructions. For the first part of the project you will have to create your own instruction by hand to test them.

Final questions

1. What components are used in stage 3 of our processor? Remember, stage 3 is the execute stage.
2. What should alu_op be set to for an XOR instruction?
3. What happens when reset is 1?
4. What control signal is assigned in stage 5 for a Sub instruction?