Project: An 8-bit Processor

The first checkpoint is due by 5pm on Thursday, April 19th, 1999.
The second checkpoint is due by 5pm on Thursday, April 26th, 1999.
The complete project must be demo’ed by 5pm on Thursday, May 3, 1999.
NO LATE HOMEWORK WILL BE ACCEPTED.

1 Overview

In the project, you will be designing and implementing a very small and simple eight-bit processor. Before you panic, this is a very simple processor! It has only 8 instructions, and we’ll walk you through much of the design. You will also be working in teams of two on this project so find a partner. The entire processor will be implemented using the Xilinx FPGAs so you won’t have to spend time wiring up that part of the circuit. You will, however, have to wire the EPROM for instruction memory, a RAM for the data memory, a switch for reset, a pushbutton for CLK, lights to view the data bus in the processor, and the Xilinx board itself to connect it to the lights and switches. You will also need to use the backplane bus in your lab kit so that the processor will be built on two boards: one for the Xilinx chip, and one for the EPROM, RAM, switches, and lights.

The processor can be split into two large sections: a data path circuit, and a control circuit. The data path will be built using parts from the xc3000 library, and the control circuit will be a finite state machine designed using VHDL synthesis. These two circuits interact to make the processor execute tiny programs. The control circuit uses signals from the data path as inputs to the state machine. These inputs determine the operation of the state machine. The outputs of the state machine are the control inputs to the data path. They control, for example, the incrementing of the PC, or the latching of the results into a register.

This project has two main purposes. The first is to offer a (slightly) more realistic circuit than the previous labs. This processor, while ridiculously small and simple, shows off many ideas that are found in any real computer processor. The second is to give you a more realistic view of what state machines are used for in digital systems. Instead of being used to solve the problem on their own, state machines are usually used as controlling circuits for other parts of a complete system. The state machine in this processor is a controller for the data path. Its job is to make sure that operations occur in the data path in the correct order so that the instructions are executed correctly.

The circuits that make up this processor are a little bigger, but otherwise no different from, the circuits you have built already, and that we have talked about in class. It may look daunting at first, but with a little planning, and attention to detail, you should be able to complete the project by the end of the semester.

2 Working With a Partner

This lab will be completed in teams of two. If you do not already share your lab kit, you should find a partner that you can work with (although, you do not have to). There are two reasons to team up on this lab. The first is that we only have enough Xilinx boards for half the class so you have to work in teams of two so that each team can have a Xilinx board. The other reason is that the lab is admittedly a little larger and more complicated than previous labs. It should help to have someone to work with. You will only need to submit one project for the team. Make sure to put both names on the documentation that you hand in.
3 Instruction Set

Because this is a tiny processor, it has an equally tiny instruction set. It includes only 8 instructions. This processor is an register/memory machine. That is, most instructions reference one register and memory. Instructions consist of either one or two eight-bit words. Remember that this is an eight bit processor, so when you get a word from memory (more on memory later), you get eight bits. In the project's processor code, the first eight bit word is the opcode and register specifier and the second eighth bit word is either an immediate or address.

This machine has two general purpose registers (R0 and R1). The first set of instructions operate on one register and one memory location. These instructions therefore require 1-byte for the operation (7 bits for opcode and a 1 bit register specifier), and one byte for the memory address. These instructions are (note that Rx below can be either R0 or R1):

- **ADD Rx Addr**: Add the value in register Rx to the value of the contents of memory location Addr. Store the result back in Rx. (Rx = Rx + Mem[Addr])
- **SUB Rx Addr**: Subtract the value in memory location Addr from the contents of register Rx. Store the result back in Rx. (Rx = Rx - Mem[Addr])
- **LOAD Rx Addr**: Load the contents of memory location Addr into Rx. (Rx = Mem[Addr])
- **STORE Rx Addr**: Store the value in Rx into memory location Addr. (Mem[Addr] = Rx)

The second set of instructions operate only on a register and one additional byte of instruction which is either an immediate value or an program address in the case of BGE. These instructions are (note that # below denotes an immediate value or address in hexadecimal):

- **ADDi Rx #**: Add the immediate # to the value in Rx. Store the result back in Rx. (Rx = Rx + #)
- **SUBi Rx #**: Subtract the immediate # from the value in Rx. Store the result back in Rx. (Rx = Rx - #)
- **LOADi Rx #**: Copy # into Rx. (Rx = #)
- **BGE Rx #**: Branch to address # if the contents of Rx is greater than or equal to zero. If the value of Rx is greater than or equal to zero, jump to location # (load the program counter (PC) with #) and start executing instructions from there. If the accumulator is less than zero, continue to the next sequential instruction. (If Rx[7] = 0 then PC = # else PC = PC + 1)

The operation codes (opcodes) for the instructions in this computer are given in Table 1.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>00000000[Rx]</td>
</tr>
<tr>
<td>SUB</td>
<td>00000001[Rx]</td>
</tr>
<tr>
<td>LOAD</td>
<td>00000010[Rx]</td>
</tr>
<tr>
<td>STORE</td>
<td>00000100[Rx]</td>
</tr>
<tr>
<td>ADDi</td>
<td>00010000[Rx]</td>
</tr>
<tr>
<td>SUBi</td>
<td>00100000[Rx]</td>
</tr>
<tr>
<td>LOADi</td>
<td>01000000[Rx]</td>
</tr>
<tr>
<td>BGE</td>
<td>10000000[Rx]</td>
</tr>
</tbody>
</table>

Table 1: Opcode Values for the Instructions

Because the program counter (PC) and memory locations (Addr) are only eight bits wide, this means that the project processor can only address 256 memory locations for instructions and 256 for data! Programs written for this machine will have to be pretty compact!
4 Execution Model

The previous section describes each of the instructions of the project processor. Let’s be a little more specific about how these instructions are executed.

The first thing that must happen is that the instruction must be fetched from memory so that the processor knows what instruction is being executed. The Program Counter is a register that holds the address of the instruction that is about to be executed. So, the first thing the project processor does is put the Program Counter (PC) value on the bus to the memory. The memory sees the address and responds with the data that are stored in that address of the memory. That is, it sends the instruction back to the processor on its data lines. Once the instruction comes back from the memory, it is stored in a register inside the processor called the Instruction Register (IR). This is so the processor’s state machine can refer to this instruction when it’s deciding what to do next.

At this point, the processor must fetch the second byte of the instruction. To do this, it must increment the value in the PC by one so the PC points to the next memory location. I suggest you use a counter to build the PC as explained later. If you do, then your state machine will enter a state where the counter is enabled to count so that on the next clock edge the PC counts up by one. When you have the new PC value, you send that to the memory over the bus (really you aren’t “sending” anything, the PC output is connected to the bus so the memory always sees it. A real bus has a more complicated protocol, but why complicate things?). The memory sees the address and responds (after some propagation delay) with the second byte of the instruction. This data should be latched into the address/immediate register (AIR).

After the instruction is loaded, the decoding is completed. If it is an ADD, SUB, or LOAD the state machine should put the contents of the AIR onto the data address bus, and latch the data coming from the data memory into the memory data register (MDR). If it is an ADD or SUB, the state machine should now obtain one of its operands from R0 or R1. The second operand should come from MDR. The state machine should also select whether the ALU is doing an add or subtract. In the next cycle, the state machine should latch the result from the ALU into the appropriate register. If it had been a LOAD, the contents of the MDR should be copied directly into the appropriate register. Finally, if the instruction had been a STORE, then the data in the register should be put onto the data memory bus, the contents of AIR should be put on the data address bus, and the data memory should be signaled to write.

The execution is a bit different for ADDi, SUBi, and LOADi. For these instructions, the contents of AIR is used as the second operand for the ADD or SUB and copied into the appropriate register for a LOADi.

In the case of a branch (BGE), the high order bit (i.e., the sign bit) of the appropriate register should be tested. If it is low (i.e., Rx is greater than or equal to 0), the value stored in the AIR should be moved to the PC. If it is high, then the PC should be incremented normally.

After the instruction has been executed, the state machine should increment the PC again so that you can fetch the next instruction from memory and start all over again. This repeats forever as the processor executes the instructions that make up the program.

5 Example Programs

The purpose of this lab is not to build a machine to replace your workstation in the CADE lab, or your PC at home, or even the microprocessor in your toaster. Instead, it is meant to demonstrate some principles of computer organization, and to demonstrate using state machines to control other circuits. Still, it will run very simple programs. The example below computes the gcd of two numbers:
/* Note that m must start greater than n */
int GCD(R0 = m, R1 = n) {
    if (m - n == 0) {
        return m;
    }
    else return GCD(max(m-n,n), min(m-n,n));
}

LOADi R0 m ; R0 = m
LOADi R1 n ; R1 = n
GCD: STORE R0 0 ; mem[0] = m
       STORE R1 1 ; mem[1] = n
       SUB R0 1 ; R0 = m - n
       SUB R0 1 ; R0 = (m - n) - n
       BGE R0 TestEquiv ; Test (m - n) - n >= 0
       LOAD R0 1 ; R0 = n
       LOAD R1 0 ; R1 = m
       SUB R1 1 ; R1 = m - n
       BGE R0 GCD ; Uncondional Jump back to GCD
TestEquiv: ADD R1 1 ; R1 = n + n
          SUB R1 0 ; R1 = n + n - m = n - (m - n)
          BGE R1 FoundGCD ; Test R1 >= 0
          LOAD R0 0 ; R0 = m
          SUB R0 1 ; R0 = m - n
          LOAD R1 1 ; R1 = n
          BGE R1 GCD ; Uncondional Jump back to GCD
FoundGCD: ; m-n & n are equal so found GCD
          ; Result is in memory address 0 and 1
6 Memory

Your processor has a generous instruction memory address space of eight bits. This means that the processor can address a full 256 memory locations for instructions. The instruction memory will be implemented using an EPROM. The address pins for the EPROM should be wired to pins in the Xilinx chip which are providing the PC register’s contents. The data pins should be wired to pins in the Xilinx chip which connect to the instruction bus.

Your processor also has a generous data memory address space of eight bits. This means that the processor can address a full 256 memory locations for data. The data memory will be implemented using an RAM. The address pins for the RAM should be wired to pins in the Xilinx chip which are providing the data address bus contents. The data pins should be wired to pins in the Xilinx chip which connect to the data memory bus.

7 Clock

Rather than build a free-running clock for this project, the system clock will again be a debounced pushbutton.

8 Procedure

Your job is to implement the processor using parts from your lab kits including a Xilinx board for each team. You should implement the datapath using the macros in the xc3000 library. To control the processor, you can design the state machine controller using VHDL synthesis. The controller will be connected to the data path to make the processor execute instructions.

The peripheral circuitry needed includes the following:

- A debounced pushbutton for the system clock.
- A debounced pushbutton for reset. When this signal is asserted, the whole system is reset, and the state machine is reset to its initial state.
- A bank of eight lights to show the current value PC.
- A bank of eight lights to show the current value on the instruction bus.
- A bank of eight lights to show the current value on the data address bus.
- A bank of eight lights to show the current value on the data memory bus.
- An EPROM to store the instructions for your program.
- A RAM to store data for your program.

Note that you may want to include extra lights for debugging purposes. When you actually build this project, you will not be able to probe signals that are inside the Xilinx chip! So, if you want to know the values of any internal signals inside the Xilinx part, you will need to bring those signals to pins, and perhaps put lights on them. For example, you might want to know what state your control state machine is in for debugging purposes. This means that you might want to put the state register of your state machine on some lights.

9 Xilinx Pin Assignments

Once you’ve designed your peripheral circuitry, you need to specify which pins of the Xilinx part will be used for your signals. This is programmable like the rest of the Xilinx configuration. The Xilinx chip that is available in the lab is packaged in an 84pin PLCC package. Some of those pins are used for specific purposes, and others can be used for I/O for your design. The only addition
to this rule is that the Xilinx part has a dedicated clock line that is handy to use for your clock signal. This signal should be connected to pin 13 of the Xilinx part. In lab 7, you learned how to force a pad to a particular pin.

You should force the signals that you are interested in to specific pins. You can do this during synthesis (see lab 7). The pins that you can use for your parts are as follows: the special clock input pin is pin 13, and the general I/O pins are pins 7, 14-20, 23-30, 35, 37-40, 44-53, 57, 59, 63, 69, and 79-80. There are 38 usable pins here which should be enough for this project. You are free to assign your I/O signals to any of these pins. Note that you will need to keep careful track of how you did the assignment so that you know how to wire up your peripheral board’s connection to its bus connector (see the next section).

10 Partitioning Across Multiple Boards

On your Xilinx board, you will notice that the upper half of the board is taken up by the Xilinx part and the programming EPROM. There is a small area at the bottom of the board that you can use to place components, but there won’t be enough space here to put all the switches and lights that you’ll need for your processor. Therefore, you should put your instruction memory EPROM, data RAM, all the switches, LEDs, LED drivers, and switch debouncers on another board, and connect to the Xilinx board through the edge connectors.

Two connect two boards together, you need to insert them into the motherboard using the edge connectors (see Figure 1). Unfortunately, the wire wrap boards are symmetrical, and can be placed into the edge connector with the component side to the right or the left. If you are ever in a position to define a board’s geometry, never design it this way, since this leaves the possibility that a clumsy technician will insert the board backwards, often destroying much or all of the circuitry on that board or even other boards. ALWAYS ORIENT THE BOARD SO THAT THE COMPONENT SIDE IS TO THE RIGHT WHEN THE CASE IS OPEN IN FRONT OF YOU. The edge connector pins on the left side of the connector (see Figure 1) will correspond to numbered lines on the backplane, and those on the right to lettered lines. As you look into the top of the motherboard, pin 1 is the “top left pin” of the edge connector (the one furthest from you as you look into the case). Pin 2 and pin 27 carry ground. Pin 28, the bottom left pin of the connector, carries 5 volts, as does the pin directly across from it (bottom right corner of the connector). These numbers should match those near the wire wrap sockets soldered into the board (on the wiring side) when the board is inserted into the edge connector with the component side to the right.

![Diagram of motherboard](image.png)

Figure 1: Diagram of motherboard.

Now let us examine the bottom of the motherboard. Lift up the board and attach it to the magnets on the inside of the top of the attache case. The large bands of silver foil are connected to ground. From this perspective, pin 1 is the bottom (closest to the hinges of the case) left hand pin of each edge connector, and the 5 volt pins are at the top (furthest from the hinges). Note the positions of the ground pins (there are 4 on each edge connector). Don’t worry if some of these pins do not appear to be soldered to the silver ground foil, although at least one pin of each edge
<table>
<thead>
<tr>
<th>Wire Wrap Side</th>
<th>Component Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Pin</td>
<td>Xilinx Pin</td>
</tr>
<tr>
<td>Pin 1</td>
<td>unusable</td>
</tr>
<tr>
<td>Pin 2</td>
<td>GND</td>
</tr>
<tr>
<td>Pin 3</td>
<td>7</td>
</tr>
<tr>
<td>Pin 4</td>
<td>13</td>
</tr>
<tr>
<td>Pin 5</td>
<td>14</td>
</tr>
<tr>
<td>Pin 6</td>
<td>15</td>
</tr>
<tr>
<td>Pin 7</td>
<td>16</td>
</tr>
<tr>
<td>Pin 8</td>
<td>17</td>
</tr>
<tr>
<td>Pin 9</td>
<td>18</td>
</tr>
<tr>
<td>Pin 10</td>
<td>19</td>
</tr>
<tr>
<td>Pin 11</td>
<td>20</td>
</tr>
<tr>
<td>Pin 12</td>
<td>23</td>
</tr>
<tr>
<td>Pin 13</td>
<td>24</td>
</tr>
<tr>
<td>Pin 14</td>
<td>25</td>
</tr>
<tr>
<td>Pin 15</td>
<td>26</td>
</tr>
<tr>
<td>Pin 16</td>
<td>27</td>
</tr>
<tr>
<td>Pin 17</td>
<td>28</td>
</tr>
<tr>
<td>Pin 18</td>
<td>29</td>
</tr>
<tr>
<td>Pin 19</td>
<td>30</td>
</tr>
<tr>
<td>Pin 20</td>
<td>35</td>
</tr>
<tr>
<td>Pin 21</td>
<td>37</td>
</tr>
<tr>
<td>Pin 22</td>
<td>38</td>
</tr>
<tr>
<td>Pin 23</td>
<td>39</td>
</tr>
<tr>
<td>Pin 24</td>
<td>40</td>
</tr>
<tr>
<td>Pin 25</td>
<td>44</td>
</tr>
<tr>
<td>Pin 26</td>
<td>45</td>
</tr>
<tr>
<td>Pin 27</td>
<td>GND</td>
</tr>
<tr>
<td>Pin 28</td>
<td>unusable</td>
</tr>
</tbody>
</table>

Table 2: Lab Kit Bus.

connector should be connected to ground. Note that most of the pins on the edge connector are floating, i.e. are not wired to anything. If these pins are not already connected between two edge connectors, you will have to wirewrap these pins to each other. If they are, you should check that they are wired properly.

To connect two boards together through the edge connector, you will need to first define which signals are being passed between the boards, and on which edge connectors those signals will be passed. The edge connectors are accessed from your wirewrap board by wiring to the pins of the solder-on sockets on the bottom of the wire-wrap board. Note that the edge connector tabs are numbered. The definition of the bus connector pinouts in shown in Table 2. How you assigned your Xilinx pins will dictate how you wire your peripherals to the bus connector.

11 Some Additional Hints

Whew! This is beginning to sound like a LOT of work. Don’t worry, there’s a lot of text here, but once you understand the problem it’s not that hard. The first thing you should do is make sure
that you completely understand the way the processor executes instructions. Do each instruction by hand pretending you are the state machine just to make sure you understand the sequence of things that has to happen. Once you understand this, it’s just a matter of state machine design (see lab 7 for details on synthesis and mapping to a Xilinx FPGA) and you know how to do that (plus hooking up a few other components like counters and registers, but you know how to do that too). The nice thing is that once you have the circuit working in PowerView, you don’t have to do much wiring. Most of the wiring is contained in the Xilinx FPGA. Of course, there is a little wiring left to do, mostly in switches and lights.

You should also make sure to fix the external signals of the Xilinx part to specific pins and keep them there. Once you wire everything up, you may discover problems and have to fix something and try again. If you’ve fixed the pins, this requires only reprogramming an EPROM. If you haven’t, it will require re-wiring!

Since there are a few extra Xilinx pins and extra bus wires anything else you want to include is up to you. What are some things that you might want to consider? You might want to include lights on the state bits for the state machine or displaying the internal registers for example. Any other signals that you think might be interesting to see, you might want to bring out to pins of the Xilinx part so you can see them.

12 Checkpoint

By April 19th, you should submit schematics and simulation files for your datapath. By April 29th, you should submit your VHDL for your state machine and simulation files for it. These two checkpoints are for your benefit. It is vital to make sure you are on the right track with this one! There will not be much time to back up and fix things later! Also, if you make the checkpoints and for some reason your project does not work quite right, more credit will be given for the work you have done.

13 Final Documentation and Demo

Start early and check with us to make sure that you understand how things are working! The documentation you should be prepared to hand in before May 6th includes the same stuff you’ve been doing all along:

1. Schematics for your datapath.
2. VHDL for your state machine.
3. Simulation results.

Remember that if you don’t document your design, and you don’t comment the documentation so that the TA can understand it, you won’t be getting full credit even if everything works! If everything doesn’t work quite right it’s even more vital to have clearly documented what you did!