EENG 2910 Project III - Digital System Design

Final Project: 8-bit Simple Processor

Nathan A Ruprecht
Ayodele Ojo
Maher Alsanwi
Yang Qi

Abstract

In the Final Project of EENG 2910, an 8-bit Microprocessor (simple processor) was designed, implemented, and tested using VHDL (**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage) by our team. A series of components were implemented in VHDL and connected ny port mapping for the simple processor to run. The implemented components were products of past assignments but brought together to be tested and simulated as one, overarching CPU. This report shows our findings and lessons learned with a formal presentation on December 7th in class.

Table of Contents

Introduction

In this final project, we are designing a simple processor with 4 4-bits registers and a 16-word memory with 8-bits words that will take in instructions and process them. A processor is the logic circuitry that responds to and processes the basic instruction that drive a computer. For the final project, we were required to answer theory questions before jumping into the implementation of our processor. We were to read the provided documents “SimpleProcessor” and “SimpleProcessorInstSet.”

“SimpleProcessor” explained a bit more of what was expected of our final project. Our simp processor (SP) needed to be able to display the outputs of the operations performed by the ALU through LEDs or the Seven Segment display. It needed to perform basic operations - add, subtract, shift and compare. Along with arithmetic functions, our SP needed to have branch instructions.

“SimpleProcessorInstSet” gave an overview of how an instruction set can be structured. In this case, instructions are 1 word (or 8 bits) that have all the needed information coded into those specific combinations of bit values. The instructions we used and combination of bit values was left open ended for teams to interpret and implement to their own desire.

Most modern processors are too complex to be used as an introductory design example. Many digital design courses and texts use hardware description language models of processors, but they are often ad hoc. What is needed is a basic processor with sufficient complexity that can be modified, programmed, and tested. Its speed and programmability are the main characteristics determining its performances. As mentioned above, a simple processor is an instructional processor, a logic-based circuit that responds to and processes the basic instructions that drive a computer.

In this Digital Systems design, it involves use of many design projects that have been done in class including a program counter (PC), register files, an arithmetic logic unit (ALU), and memory (RAM or ROM). The design of a computer processor combines these components into an integrated digital system. A simple processor has been developed for use as an integrated design in many devices. The architecture is separated into the data path and a sequential controller. The data path contains the memory, registers, ALU, and interconnecting busses. The controller implements the fetch, decode, and execute sequences, using basic state machine design techniques. The entire system is modeled in VHDL and can be simulated to demonstrate operation of the processor. A Field Programmable Gate Array (FPGA) implementation also provides a functional hardware version of the processor.

Function of a Processor - Instruction Set

A Simple Processor is just that - simple. It does exactly what it is told, when told to do it. Nothing more and nothing less. With that in mind, the first thing our team had to decide is what the processor would do, what precise instructions it would follow. We decided to simply put a number into memory (in our case, 5), then loop through the code 4 times and continuously adding 5 to a new register. Once that was done, it would go back to the beginning of the list of instructions, and start again by first clearing (zeroing out) all the memory it would use. Below is a rough (and almost Assembly looking) representation of what our instructions would look like.

```
begin   sub R1 - R1 -> R1           -zero out register 1 (R1) by subtracting it from itself
        sub R2 - R2 -> R2           - zero out register 2 (R2) by subtracting it from itself
        add 5 + R1 -> R1            - put the number 5 into R1
        shift R1 left -> R2         - shift the value of R1 (multiply by 2) and store in R2
        sub R3 - R3 -> R3           - zero out register 3 (R3) by subtracting it from itself
        sub R4 - R4 -> R4           - zero out register 4 (R4) by subtracting it from itself
loop    add R2 + R3 -> R3           - add R2 and R3 and store the answer as the new R3
        add 1 + R4 -> R4            - add 1 to R4 to use as a counter
        compare R1 > R4             - see if we've looped through 5 times
                true jmp to loop    - jump to "loop" if we have not completed 4 loops
                false jmp to begin  - we looped 4 times so jump to begin to do it all again
```

After we decided what our processor would do, we had to choose a decoding scheme to turn our Assembly style instruction set into binary. Looking at the above instructions, we know that at some point, it would add a constant to a register, it would add a register to another register, subtract registers, subtract a constant, compare a constant to a register, shift (left or right), and jump to an address. With 8 possible instructions, we use 3 bits to represent that specific operation code (Op Code) designated to that specific instruction. The next bit values determine what the constant (CONST) is or the source (SRC) register where the data will come from. Then finally the destination (DST) register where the answer will be stored. In the case of jumping, we just need the address (ADDR) of the instruction to jump to. This is done simply by assigning the PC to point at that address the next time it changes. Below is a key that breaks down how all our 8-bit instructions are read - U is unknown since it does not matter and X is the bit value:

Function	Op Code		
add constant	000	XXX (CONST)	XX (DST)
add register	001	UXX (SRC)	XX (DST)
subtract register	010	UXX(SRC)	XX (DST)
subtract constant	011	XXX (CONST)	XX (DST)
compare constant	100	XXX (CONST)	XX (DST)
shift left	101	UXX (SRC)	XX (DST)
shift right	110	UXX (SRC)	XX (DST)
jump to address	111	U	XXXX (ADDR)

R1 is 00

R2 is 01

R3 is 10

R4 is 11

Component - ROM

All of our instructions are stored in the ROM which, in this case, is the memory that will not change. The PC will point at an address in ROM and take the data (the instruction) that is stored there. On the rising edge of a clock cycle (I_clk), the program checks that the address the PC is pointing at is outside the range of how many instructions we have. If it's within range, the PC reads the data that is stored at that address. As stated before, the data stored at each address is the combination of bit values that represent our instructions. Using the above scheme, our instruction set becomes:

01000000	--instr 0	begin	clear R1
01000101	--instr 1		clear R2
00010100	--instr 2		add 5 + R1 -> R1
10100001	--instr 3		shift R1 left -> R2
01001010	--instr 4		clear R3
01001111	--instr 5		clear R4
00100110	--instr 6	loop	add R2 + R3 -> R3
00000111	--instr 7		add 1 + R4 -> R4
10010011	--instr 8		compare 4 > R4
11101010	--instr 9		true jmp to loop
11100000	--instr 10		false jmp to begin

Component - PC

A PC does not count, think of it as a pointer. The PC is always pointing at some location in memory and seeing what is there. But we can choose where it points (assign - 10), if it stays pointing at that address (no operation - 00), moves on to the next address (increment - 01), or if it goes back and points at the first address (reset - 11) using Op Code - in our code it's called I_nPCop.

Again on the rising edge of the clock (still I_clk), the code looks at what the op code is telling the PC to do. No op does nothing for this cycle. Increment will make the PC point at the next instruction. If it's pointing at the last one, the PC is moved back to the first instruction. Assign will tell the PC to point at a certain address (I_nPC). Reset will make the PC point back at the first instruction. In all cases, the PC look at the data of the address it is pointing at and tell the Control Unit / Simple Processor what is there (O_data or O_OP). We were able to recycle our code from when we first made a program counter with memory during Lab5&6.

Component - RF

The register file is our temporary memory. In other words, considering our CPU as an entire system, the register file is the platform that will show all the datas that have been processed and the data will be used in processing. These data in these register are almost constantly changing depending on what the ALU is outputting. This is where R1, R2, R3, and R4 are. So the PC fetches an instruction from the ROM, depending on the last 4 bits of the instructions is what registers are going to be manipulated. The RF will take in a "in data" variable that is a selector, and output the data of the register that was selected. The instruction gives a 2 bit value that points to a certain register, the RF takes that value and gives the data at that address. Sounds familiar because it is similar to what the PC does with the ROM.

Now in this case, the RF holds memory to be manipulated and the pointer is always being assigned (it never increments or stays still). Since the data in the RF can be changed, while interacting with the ALU, it takes the input data I_we from ALU, and output a feedback signal we_done back to tell ALU when the writing is done, so that ALU knows when to turn the I_we signal off, and we can read data from the register file again. We were able to recycle the code again from ROM from when we did it in Lab5&6 with the PC.

Component - ALU

The ALU take in 2 inputs, a selector line, a writing variable, and outputs an answer and I_we signal. All of which depend on the instruction from the PC. As we talked about, our 8-bit instruction is a combination of bit values that tell the ALU what to do and on which registers. The first 3 bits of an instruction is the op code. It's telling what function the ALU will perform. Inside the ALU, there is a case statement (a multiplexer or MUX) that looks at the op code and lets that function through to the output. Each of those functions has a component coded in Xilinx that in turn takes in input and tell the ALU the answer that in turn tells the CU the answer. Those components are the add_const, add_reg, sub_const, sub_reg, shift_left, shift_right, and compare.

The add_const and add_reg are identical. In both cases, the ALU is feeding 2, 8-bit numbers (whether a constant of our choosing or the data from the register) and using a 8-bit full adder to add the 2 binary numbers together. In our case, the 8-bit full adder has its own component being the 1-bit full adder. So 8, 1-bit full adders in series that feed in the carry bit to the next adder will produce our 8-bit full adder.

The sub_const and sub_reg are also identical in their own way. Like the adder components, these two components will add 2, 8-bit values fed by the ALU. To do binary subtraction, we do the 2s complement of the second input value (B), then add them together. Subtracting two numbers is like adding number A to the negative of number B: A + -B = A - B.

The shift components (left and right) do a bit of parsing. It grabs seven of the eight bits, moves them in the desired direction, then brings back that bit that got wrapped around. These two components only take in one input since its job is to shift and not do mathematical calculations.

The comparator is simple enough in that it just compares the 2 inputs and outputs a certain value to show how they compare to each other. If A is greater than B, output B. Otherwise, output “00010000”. Depending on those outputs fed all the way back to the CPU will determine where the PC goes next. This comparator is what determines if the CPU exits the loop or stays in it.

The bits of the instruction that correspond to a register go through its own case statement in the CPU to see which register the bit values match with. The data of that register is what is needed for the ALU. Again, a pointer versus the actual data. The bit values in the instruction is pointing to the register, the ALU needs to be able to manipulate the data of that pointer.
In order to manipulate that data, we need a variable that toggles between reading data from the RF and writing (really overwriting) data. The we_done tells the ALU if it was reading or writing, then the I_we toggles to make the ALU do the opposite the next time it goes through. We were able to recycle code used for the ALU from when we made it during Lab 3.

CPU - Architectural Diagram

CPU - State Diagram

A state diagram of the operation inside the CPU can be seen in Figure# .

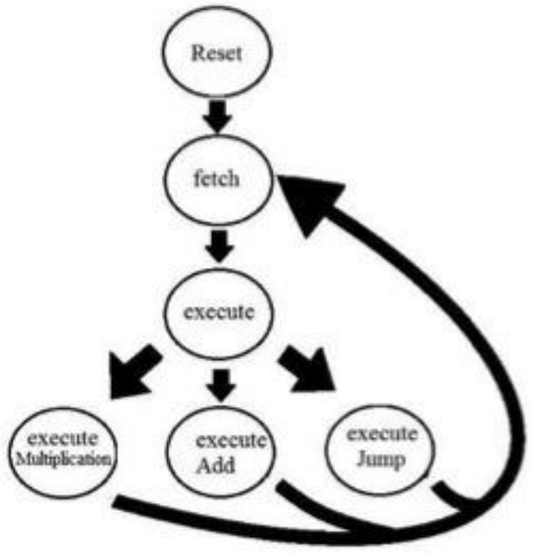

Figure # shows the state diagram of the operation that takes place inside CU. When the reset button is triggered, the fetch is activated. The instruction is then executed to one of the three operations (LDA, Add, and jump). Once the instruction processes, the data goes back to fetch again.

Results

A simple processor (SP) was designed, tested, and implemented using VHDL by our team. Using VHDL module the codes for the components were rewritten separately and then tested using Test bench. A simulation waveform was created from the testing.

Simulation

The VHDL module that we have built for simulation basically only have two input ports which are the clock signal I_clk and the reset signal rst. As the clock signal changes, the state moves sequentially, so that the OP code changes which will be decoded into sel_a, sel_b, and sig_sel. Then, the Register file and ALU reads the input selectors and process the data, finally, the state will assign the Program Counter into next instruction address, and the state will also move to the next. By observing the current state and the data stored in the Register File, we are able to understand how the CPU works.

The overall simulation result in 1000ns (100 clock period) for the design is shown in the Figure#

On Board Implementations

7 segment LED: A 7 Segment is an LED or Light Emitting Diode, is a solid state optical PN-junction diode which emits light energy in the form of “photons” when it is forward biased by a voltage allowing current to flow across its junction. A segment decoder was created that will

decode a 4-bits BCD into the 7 segment. Also a 4x1 MUX is needed to selected one of the input data and push it into segment decoder.

From BCD code to 7 segment code: BCD stands for binary coded decimal. A BCD code is a 4 bit number that can represent the numbers 0-9 (0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001). A BCD to 7 segment decoder maps each of these 1 0 codes to 7 bit codes (one bit for each segment) that will control the display.

The chart for both common anode and cathode is:

Decimal	BCD	Common anode	Common cathode
		gfedcba	gfedcba
0	0000	1000000	0111111
1	0001	1111001	0000110
2	0010	0100100	1011011
3	0011	0110000	1001111
4	0100	0011001	1100110
5	0101	0010010	1101101
6	0110	0000010	1111101
7	0111	1111000	0000111
8	1000	0000000	1111111
9	1001	0010000	1101111

The 7 segment decoder: The decoder will have 4 inputs of type std_logic or one input of type std_logic_vector(3 downto 0). We are using std_logic_vector. The output is a std_logic_vector(6 downto 0). The decoding was done with the selected signal assignment statement. For implementation we used 4 slide switches as inputs and the 7 data elements as outputs. The common element usually activates itself when not declared. If not then you must declare one more output of type std_logic and give it the value 0 or 1 depending on the element we have (anode or cathode).

Register LED: The implementation of Register LED basically uses two switches on the board to construct a 2-bit input as selector pointing to the Register File using a encoder. Then, the Register File will output a 8-bit data which is stored in that address. The 8-bit data will further be decoded into 8 1-bit data which matches 8 LEDs on the board. In this way, we are able to select one register and observe the processing that is going on the Register File.

The simulation waveform for the PC, CU, ALU, RAM, IR, and SP can be seen in Figure Stuff, Figure Stuff, Figure Stuff, Figure Stuff, Figure Stuff, and Figure Stuff.

Conclusion

please, see if you want to edit it
The goal for this project was for the students to execute an eight bit simple processor using a state machine approach with VHDL. A simple processor (SP) was successfully designed, tested, and implemented by our team. The SP contains all, or most of, the central processing unit (CPU) functions**.** The SP sends signals to control the other parts of the computer. This SP has four primary functions: fetch**,** decode**,** execute**,** and write back. There were several important components (PC, CU, MUX, DMUX, ALU, RAM, IR, and shift register) that were port mapped together for the SP to work properly.

The test bench in VHDL was used to test whether the correct inputs were passing through for the outputs received. The results were simulation waveforms of the assigned inputs that were tested. Therefore, the code was confirmed to be correct and working properly. However, some of the components failed to work together during port mapping and the final SP simulation waveform could not be created. Through the help of PowerPoint and notes posted in blackboard, and a huge help from our professor, an 8 bit simple processor was created and tested in time with less difficulties.

Appendix - Source Code