CS/EE 3710 — Computer Design Lab
Mini-MIPS processor: The Semester in a Nutshell!
Controller modification, memory mapping, assembly code

Due Monday, September 19, 2022

Assignment Objectives

The objectives of this lab are to understand and extend a very very simple processor, read and write to block memory on the Altera Cyclone V FPGA, use memory-mapped IO with your processor, use the switches and LEDs on the DE1-SoC board, and write a program to run on your processor. This will give you a start on seeing the bigger picture for your semester project by essentially doing a (much) simpler version of most of the basic tasks. To accomplish this, do the following:

1. Read and understand the code for the tiny MIPS processor. This processor is from the CMOS VLSI Design book by Weste and Harris. The Verilog code from the appendix of that book is on the class web site in mipscpu.v. (slightly modified from the original)
2. Read and understand the exmem.v code, also on the class web site, that implements a 256 byte memory module used by the mipscpu.v code. This will be mapped to a block RAM by Quartus on the Altera FPGA.
3. Read and understand the fibn.asm code that runs on the combined mips.v/exmem.v circuit. This code computes the 8th Fibonacci number and stores the answer (0D hex) in memory location 255.
4. I'll also put an excerpt from the book that describes the processor on the Canvas page that you should read and understand. This describes the instructions, the instruction encoding, the initial Fibonacci code, the block diagram of the processor, and the control finite state machine.
5. Modify the mipscpu.v controller to include an ADDI (add-immediate) instruction to the instruction set. This instruction (listed in Table 1.7 in the MIPS handout) is an "I" format instruction that takes two register arguments \$1 and \$2, and an 8-bit immediate value. The function is \$1 $\leftarrow$ \$2 + imm.
6. Modify the mipscpu/exmem system with very simple memory-mapped I/O. The memory map will use the top two address bits to define regions of memory. If the top two bits are 11, then you should read and write to and from I/O space. Reading from I/O space will get the value from the switches on the DE1-SoC board. Writing to I/O space will write to a register whose output is connected to the LEDs on the DE1-SoC board.
7. Write new Verilog code (mips.v) to assemble the mini-mips processor by instantiating a copy of the mipscpu and the exmem and connecting them together, along with whatever support you need for your memory-mapped I/O.
8. Write a new Fibonacci assembly program that computes all of the first 14 Fibonacci numbers (numbered 0 through 13) and stores them in consecutive memory locations starting at location 128. After computing and storing the numbers your program should enter a loop where it reads the values on the switches, and uses that value as an address offset. Then read from memory using that offset from location 128, and write that value (e.g., the nth Fibonacci number) to the LEDs.

9. Demonstrate your mini-MIPS processor running your Fibonacci code on your DE1-SoC board. The demo will involve loading your processor onto the board, and your Fibonacci code into a block RAM (using $readmemb or $readmemh). Then your processor will run that code. Once the main code has finished running you should be able to use the switches to address the memory and have the n-th Fibonacci number show up on the LEDs.

Details and Discussion

This may seem like a daunting lab, but break it into pieces and it's not so bad! First you should read the handout from the book and make sure you understand the processor's instruction set, and the given Fibonacci code. If you understand what each instruction does, then you should be able to "execute" the program in your head or on paper to see what's happening. You don't really need to step through each instruction on paper, but do make sure you understand what's going on.

Adding the ADDI instruction

I would first take a look at Fig 1.54 and understand the control state machine that makes the processor work. Note that this is a strange little processor - it has 8-bit data paths (thus 8-bit registers), but a 32-bit instruction format. The memory is a 256 word, 8-bit memory. So, each instruction takes 4 bytes, and thus four locations in memory. This means that the instruction fetch portion of the state machine takes four cycles. After those four memory reads, the 32-bit instruction is in the Instruction Register and you can proceed to decode and execute that instruction.

If you walk thought the state machine you should be able to understand how each of the instructions in Table 1.7 are executed. This will let you decide how to add the ADDI instruction to the processor. You may need to add states to the state machine. You may need to add other support in the Verilog code. As a hint, if you're adding a LOT of stuff to the Verilog code you're probably making it more complex than it needs to be. It's a fairly simple change to the code.

Once you have the ADDI instruction added to the code, you should be able to execute the Fibonacci code in Fib 1.51 in the Quartus/ModelSim simulator and make sure that your ADDI is working properly. The code is working properly if it correctly computes the 8th Fibonacci number (starting at 0 the 8th Fibonacci number is 00001101 or 0D hex) and stores it in memory location 255.

See the exmem.v code on the web site to see both how to implement memory in the Block RAM, and how to initialize that memory with a program. You basically use a data file that Verilog loads into that memory. This data file is also used by Quartus when it's building the FPGA programming information so that the Block RAM on the FPGA also gets loaded with that data.

Memory Mapped I/O

Now that you have a working mini-MIPS processor with an ADDI instruction, you should look at modifying the memory system so that you can read a value from the switches and write to the LEDs. The technique used here is to map I/O devices into the memory space of your mini-MIPS. What this means is that for a certain range of memory addresses, reading and writing to that range of addresses has the side effect of reading or writing to the I/O device instead of actually writing to memory.

You should divide your memory space into four equal sections. The memory for this processor is extremely limited - it's only 256 memory locations because the address is only 8-bits. For this lab the memory will be divided into four regions of 64 locations each. The first three will be "regular" memory, and the upper section will be I/O. You can decide what region of memory you're in by looking at the top two address bits. If those top address bits are 00, 01, or 10, you are in "regular" memory. If those top two address bits are 11, then you are in the "I/O" region of memory.

If you read from a memory address with 11 in the top two bits you should get the value that's on the switches. If you write to the an address with 11 in the top two bits you should cause the value to be written to the LEDs.

To achieve this memory mapping you will design a circuit that looks at the top two bits of the address to know what to do on each memory operation. If you are in "regular" space, then the block RAM should provide data to the cpu, and nothing should update on the LEDs on a write. If you're in "I/O" space, then a read operation will return the value on the switches to the cpu, and a write will update the value on the LEDs. For reading from the switches this likely involves a MUX that decides whether the data that comes back from your processor is coming from the memory or from the switches. For the LEDs, you'll need (at least) a separate register to hold that value (so that things don't change between writes).

New Fibonacci Code

Now that you have a modified mini-MIPS system that has an ADDI instruction and simple memory mapped I/O, you can write your new Fibonacci code. This code should compute the first 14 Fibonacci numbers (starting at 0). It should store those numbers into memory locations 128 through 141 (decimal). In other words, the ouptut starts at location 128 and the index beyond that starting location is the nth Fibonacci number. Location 128+3 is Fibonacci(3).

Now that your program has computed and stored those numbers, you should be able to use the switches on the DE1-SoC board to access the memory. Using memory-mapped IO your program should enter a loop where it reads the value from the switches, and uses that value as an index. It then looks up the contents of memory location 128+index and stores that value to the LEDs. The effect when this program is running is that when you change the switches on the board, the Fibonacci number corresponding to that index shows up on the LEDs.

Write this Fibonacci code in the MIPS assembly code from Table 1.7. You'll have to assemble it by hand (or write your own simple assembler, although in this case that's probably too much effort) into machine code. Then you can make a fib-new.dat data file and load it into the exmem using $readmemb. You should be able to simulate and demonstrate this code on the DE1-SoC board.

Notes

There are a few things you will need to keep in mind when you're doing all this.

- Pay careful attention to the instruction encoding, especially where the register address go. For example, an ADD instruction is

  ```
  ADD $4, $5, $6  : the effect is that $4 gets the value $5 + $6
  ```

  Now if you look in Figure 1.49 you can see how to encode this. An ADD is an R type instruction. For this example, Rd (the destination register) is $4, Ra (source register a) is $5, and Rb (source register b) is $6. The encoding is not in the same order as in the symbolic assembly code. The encoding is:

```
   opcode  ra    rb    rd    0     function
-------------------------------------------
  000000  00101 00110 00100 00000  100000
```

- When you put an instruction into the .dat file that gets loaded into the exmem RAM (either for simulation or for mapping to the FPGA), the byte ordering is little ending. So, that previous 32 bit machine encoding for the ADD $4, $5, $6 instruction would look like the following in the .dat file (note that the addresses of the bytes in the file start at 0 and increase as you go down the page so these are bytes 0, 1, 2, and 3 as they go down the page):

```
00100000
00100000
10100110
00000000
```

- In order for the .dat file to work for mapping the FPGA, you MUST have exactly 256 lines in the file so it matches up exactly with the 256 byte capacity of the exmem RAM that's defined in exmem.v. You need to pad the file with 0-bytes after the program portion. See the fibn.asm file on the web site for an example.
- Note that the machine code in Fig 1.51 has a couple typos in the book version.
  - The beq instruction needs to have an offset of 4 instead of the 5 that's in the book. This is because the PC has already been incremented by the time the offset is added for a branch, so even though the branch-to location is 5 instructions further on than the beq instruction, the PC is already at "beq + 1" so the offset should be 4. Another way to look at this is if you want to branch back to the very same instruction, you'd need an offset of -1.

    Note here also that the displacements for the branches are in terms of instruction words, not bytes. So, a displacement of 2 means to branch to the third instruction following the branch, not the third byte following the branch. Likewise, all read and write addresses are word-adresses not byte addresses.
  - The opcode for the sb instruction on the last line is incorrect. It should be 101000.
- Note that the register file described in mipscpu.v has only 8 registers (each 8-bits wide). So, even though the MIPS instruction encoding has 5 bits for a register address, this mini-MIPS has only 8 registers. Don't use more than 8 registers in your new Fibonacci code.

- Just to make sure we're all looking for the same answers, here are the Fibonacci numbers that I expect to be computed by your code:

Index		**Fibonacci Number**	
Dec	Bin	Dec	Bin
0	0000	0	00000000
1	0001	1	00000001
2	0010	1	00000001
3	0011	2	00000010
4	0100	3	00000011
5	0101	5	00000101
6	0110	8	00001000
7	0111	13	00001101
8	1000	21	00010101
9	1001	34	00100010
10	1010	55	00110111
11	1011	89	01011001
12	1100	144	10010000
13	1101	233	11101001

What to Turn In

1. A README file that describes your results, problems you encountered, and what files you're including in the rest of the documentation.
2. Your modified mipscpu.v code with comments around each line that you changed when you added the ADDI instruction.
3. The Verilog code for the whole mips system including the mipscpu, exmem, and memory mapping support code.
4. A simulation that shows your mipscpu/exmem system executing the fibn.asm code from the book. This code, and the fibn.dat file, can be found on the Canvas page. Your simulation (a waveform is sufficient) should show the 8'h0D value being written to memory location 255.
5. The symbolic assembly code for your new Fibonacci program
6. A simulation that shows your processor, including the memory mapping circuits, executing your new Fibonacci program at least far enough to demonstrate computing some of the Fibonacci numbers. A waveform is sufficient, but please annotate it with information about what we're looking at.
7. Demonstrate your working circuit to the TA or instructor.