CS/ECE 3710 — Computer Architecture Lab
Checkpoint #2 — Datapath and Memory Infrastructure

Overview

In order to complete the datapath for your *insert-name-here* machine (I assume you're going to want to name your machine...), the register file and ALU that you designed in checkpoint 1 needs a little support. In particular, the extra registers that were hinted at in checkpoint 1 need to be fleshed out and the initial memory interface needs to be completed. In order to have a complete datapath, you need to make the program counter complete, the MAR (memory address register) and MDR (memory data register) need to be specified (they may be inside the M10k block RAM, or they may not), the memory access process needs to be figured out for the Block RAM on the FPGA, and other registers like the instruction register and immediate register need to be instantiated, as well as sign extenders, etc. Once this is done, the remaining tasks are the instruction decoding, the control state machine, and (the biggie) figuring out what support you need for I/O for your application.

Program Counter

The program counter is a dedicated special register in the machine that holds the address of the next instruction to execute. It needs to be capable of being updated in every way the PC needs to be updated. For your machine, this means that the PC needs to be incremented by one word (the normal case), added to a sign-extended displacement (for branches) or loaded from a register (for jumps). Your datapath needs to be able to perform all of these operations. If your PC is already set up to feed into your ALU, then you could do the branch displacement calculation by setting some input MUXes so that the PC and the immediate go to the ALU and the ALU function is set to add. You might load from a register by setting the ALU such that the appropriate register source makes it through the ALU without modification. This value can then be latched into the PC. For the increment case, you could either put the PC through one side of the ALU, and select a constant 1 for the other argument (put a constant value on one of the input MUXes or something similar to that approach), or you could build your PC as a loadable counter. If you use the counter approach you can load the counter for the update and displacement functions, and count the counter for the increment-the-pc function. The choice is yours. The advantage of the counter is that you may not have to use the complete datapath for each PC increment, the advantage of the increment-through-the-alu approach is that every pc-update function goes through the same process, but with different MUX settings. Remember that if you use the ALU to update the PC, you should *not* update the condition codes! Finally, remember that for a JAL instruction, the PC needs to have a path into the register file so that PC+1 (i.e. the address of the next instruction following the JAL) can be stored in the link register. Your datapath must allow this operation.

Another issue with the PC has to do with signed and unsigned arithmetic. Recall that the signed arithmetic is all done with two's complement numbers. This means that the range of numbers in a 16-bit word is -32,768 to 32,767. On the other hand, if you use those 16 bits to encode an unsigned number, you can represent 0 to 65,534 (64k). Since addresses are usually considered unsigned numbers, we need to consider what it means to have an unsigned PC that is operated on by a two's complement ALU, especially in the face of signed offsets that might require subtraction! Consider what happens if you try to use two's complement signed arithmetic to take a large unsigned number (large enough so the high-order-bit is 1) and add a negative signed number to it to try to subtract something (so the high-order bit is also one here since it is a negative signed number). Does it work? Are there constraints on when it works? I recommend trying this out on some smaller numbers to get a feel for what's going on. Trying this on 4- or 5-bit numbers is a good way to test things out to make sure you understand what's happening.

Note that our PC is addressing 16-bit words and not bytes! I think the best way to think about it is that with 16 bits of PC you can directly address 64k locations. Each of those locations is a 16-bit word. If you really want to think about it as bytes then the address space (without playing any tricks with segment registers or things like that to increase the address space) is 128k bytes, and the PC 15:0 is the word-portion of that 17-bit byte-address-space. But that's probably the wrong way to think about it. 64k words in the address space with each word being 16 bits is easier, I think.

Instruction Register

This is a 16-bit register that holds the current instruction that you are executing. From this register you can decode all the information needed to execute the instruction. This information will consist of register addresses (both sources and destinations), function code information for ALU and shifter, MUX settings for the various MUXes in the control paths, and other information about which instruction is being executed for so the control state machine knows what to do. The main issue with the instruction register is where it gets its data from. See the MDR discussion for more details...

Sign Extension

Various instructions in our machine make use of sign-extended immediate values. Recall from reading the 3710-ISA handout that immediate values in arithmetic operations are sign-extended from the 8-bits that are in the instruction encoding. Logical immediate operations are zero-extended instead of sign-extended. Check the 3710-ISA handout for details. Sign extension can be easily done using Verilog concatenations as inputs to various circuit components like MUXes.

Memory and Memory-Mapped I/O System

Load and Store instructions on our version of the machine also point to word locations just like the PC (load and store addresses are word-addresses). There's no way to load a single byte on our machine. Thus, the Load and Store instructions also use word addresses. I strongly recommend that you design your I/O as memory-mapped which means that access to I/O devices is by loading and storing to special locations in the memory space. Each of those locations will be a 16-bit location because all loads and stores deal with 16-bit data in our machine. That being said, there may be some limitations given that our FPGA board has only 10 switches. So, if you want to be able to load values from the switches by reading from I/O space, you may need to limit that to 8 bits (8 switches) and write code to load and shift to get all 16 bits from the switches. There may be other instances of slight mismatches between the resources on the FPGA board and what you would like as ideal I/O.

The memory map for one example of our baseline machine is shown in the slides on the canvas page (last slide in the CR-16 intro slides). In that case the memory was separated into four equal sized chunks. These chunks are either 16k words if you're thinking of 16-bit words as the basic unit, or 32k bytes if you're thinking of bytes as the basic unit (I think words are easier in this case). Looking at the top two bits of the address can tell you which quadrant of memory you're in. If you're in the top quadrant of memory (word address C000 to FFFF) then you're accessing I/O space in this example instead of code/data space. I/O devices would be mapped into specific locations, or ranges of locations, in that space. Remember, the larger you can make the range of addresses that correspond to an I/O device, the fewer bits you need to check to see if you're accessing that device.

Let's be a little more specific with a couple of required I/O devices: the LEDs and the switches on the FPGA board. The LEDs could be mapped to memory as the space defined by C0xx. This means that whenever you do a STOR to an address in the range of C000 to C0FF that value will get written to the LEDs instead of to the memory. This means a couple things from your point of view. First you need to have an LED register that is written only when addresses in that range are being written to, and the outputs of that register should be connected (through pin assignment) to the LEDs on the FPGA board. Note that there are only 10 LEDs above the switches on the board. That means that your LED-output I/O will either only show the low order 10 bits of the 16 bit register being written, or that you may have to use, say, 6 of the LEDs in one of the hex digits for the upper 6 bits of data.

Because the LEDs are a write-only I/O device, you could re-use that range of addresses for the switches. Or you could define a different memory range for reading the value from the switches. I think it probably makes more sense to use the same range. Either way the switches are also mapped into the I/O space. When you read from, say, C0xx, you should get the value on the slide switches into the register. I would probably map the low-order 8 slide switches to the switch I/O so you can only read 8 bits at a time from the switches. That way you could read in an 8 bit value in one READ, or use some simple shifting and masking to read 16 bits in two pieces from the switches.

Whatever you decide to do with those bits, you will need to build an address decoder for your memory system. In its simplest form it looks like the mini-MIPS lab and needs to make sure that memory loads and stores in the range of 0000 to BFFF go to the memory that you're using for code/data, and loads and stores outside that range do something else. Address decoding at its simplest looks at the address on the address bus and uses combinational logic to control the enable signals of memory elements based on which address is being accessed.

FPGA Block RAM interface

The most convenient memory to use for your processor is the Block RAM that lives on the Cyclone V FPGA chip. This RAM is configurable in several different ways based on how you describe the interface in Verilog. The simplest is a single-port Static RAM (SRAM) that has one address that it uses for both reading and writing. But, it can also be configured to be a dual-port memory with two completely separate address/readport/writeport interfaces. This can be tremendously useful if your CPU wants to access the memory AND some other device (e.g., a VGA controller) also wants simultaneous access to that same block of memory. But, let's not get ahead of ourselves.

A nice generic interface to memory from your CPU might look similar to Figure 2. In this interface the program counter (PC) of your CPU sends its value to the memory as the address to read from (to the Memory Address Register or MAR). The data at that address is read back through the Memory Data Register (MDR) and goes both to the Instruction Register (IR) and to other places in the CPU such as the ALU.

About Cyclone V Block RAM

First of all, please go through the Cyclone V device overview (cv_51001_device_overview.pdf), device handbook (cyclone5_handbook.pdf), and go through the **M10K** Block-RAM resources, and the embedded memory user guide (ug_ram_rom.pdf), all of which can be found on the class Canvas page. You will notice that our FPGA, Cyclone V SE A5, has **397 blocks of RAM**, and each block is a configurable block of **10K-bits** memory cells. Since we are working on 16-bit data words, we can configure each block as 16-bit wide, and 512 words. Actually, internally, the block-RAM may even be configured as 20-bit words, where the four MSBs are unused in our case.

This means that with 397 blocks, we have a total of 203,264 words, each 16-bit wide. This should be enough memory to play with for most applications that you may have in mind for the project, especially if some of the address space of your machine is dedicated to I/O. If you need more memory, we can address those issues later.

Note especially that the **M10k** memory blocks are *synchronous* memory blocks. That is,

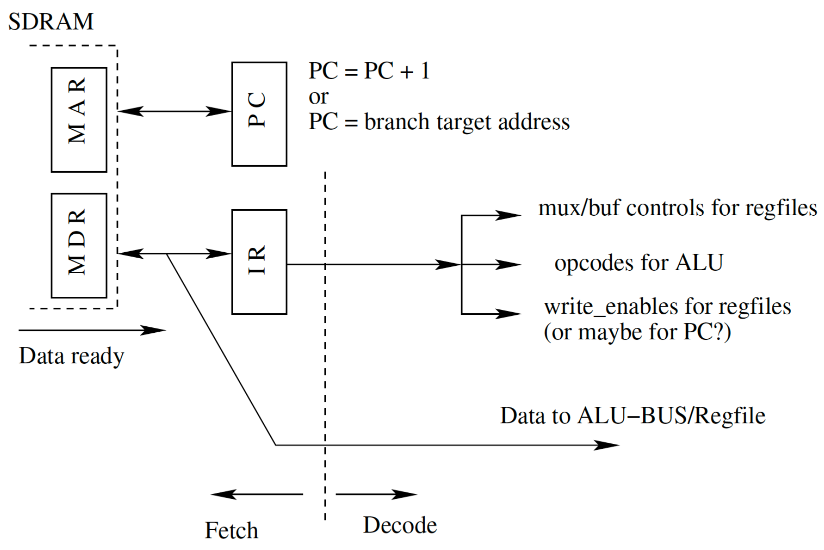

Figure 1: A generic interface between a CPU and a static RAM

there is a clock signal that goes into the Block RAM, and the activities, both read and write, happen after a clock edge triggers the RAM. If the activity is a read, then the read happens after the next clock cycle after the read address is seen by the memory. If the activity is a write (the WE is asserted), then that write also happens after the clock edge. Of course, you get a read value on every clock. Only when the WE is asserted is a value written to the RAM too.

Which brings up an important question - when you are performing a write to the **M10k** Block RAM (you have the write address on the address lines, the data to be written on the WData line (data from the CPU or RF to the RAM), and you have the WE asserted), what data is read back on that cycle? Is it the data that *used to be in the RAM* before the write? Or is it the data *you just wrote to the RAM?* It turns out that the answer (as is most often the case) is "It depends!" The **M10k** Block RAM can be configured to be either way - either *write first, then read* or *read first, then write.* Check out the Block RAM documents for more details.

For this assignment, you probably don't have enough of your CPU done to use the CPU to test the memory interface. So, you'll instantiate a Block RAM and write a simple state machine (in Verilog) to test your memory.

What to Do

Use the information in this handout, and your own knowledge of the processor design, to finish off the datapath of the machine. That is, after this lab you should be able to demonstrate (in simulation at least) that every datapath manipulation that will be required by the processor is possible. This includes memory access, memory controller, and memory state machine. You should have a list of every control point into your data path - every signal that will control some aspect of the datapath such as a MUX select, register enable, function code, etc. The decoder will then take the instruction word and produce control bits, and the finite state machine controller sequences through other control points (like register enables) and makes the instruction happen.

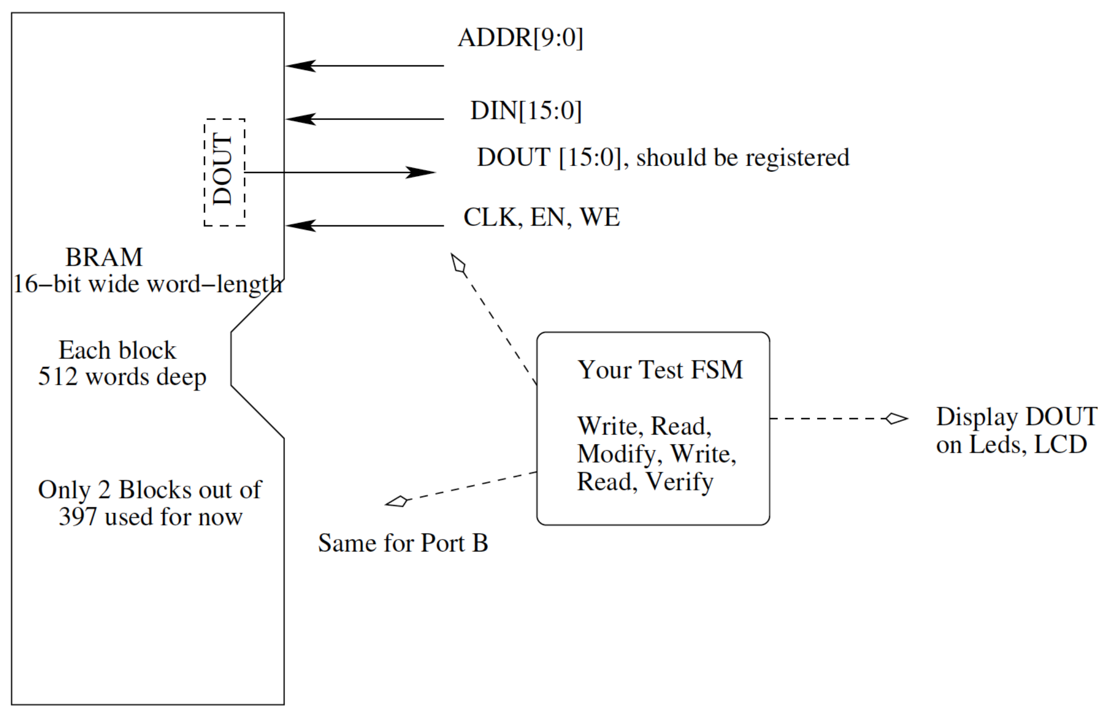

Figure 2: A sample interface between the synthesize Block RAM and your test FSM

(note that the decoder and control state machine are for another checkpoint, but you should be thinking about them!)

For your memory, do the following:

- Design (in Verilog) a synchronous true dual-port memory. This will likely come in handy later in the project when some other part of your system (such as the VGA controller) needs to have access to your system memory. See the bram.v example on the Canvas page for inspiration.
- For now, synthesize a 1k x 16 memory. Because each of the **M10k** blocks has 10k bits, or 512 16-bit words, this should synthesize using two of the blocks.
- 1k words requires 10bits of address - remember that our processor is word-addressed, not byte-addressed. Note that for the full project if you use 16 bits for your address, you can access 2^{16} words, or 64k words of memory (128k bytes). That *should* be enough for your project, but if it's not, there is more Block RAM on the FPGA that you can use. The total on the FPGA is 397Kbytes of RAM (or 198.5K words). so, not quite 2^{18} words of memory. If you really need more memory than 64k words, you could move to a 17 or 18-bit address. But, that complicates things so don't worry about that now. For this lab, 10 bit addresses and 1k word memory is what we're after.
- Using `$readmemh()` or `$readmemb()` Verilog system functions, initialize the memory with known data at various locations. Initialize some known data at addresses 0, 1, 2, and then some data at memory addresses 510, 511, 512, 513 to see how the data is stored correctly in the two different blocks of RAM.

- Simulate and validate using ModelSim
- Design a simple state machine to walk through some of the memory addresses and perform simple modifications to the data, write that data back, then re-read that location to make sure that you can write to the memory. Remember that all memory operations will be synchronized to the clock.
- For the on-board demo - put the address on the LEDs, the read data on one pair of the 7-Segment displays, and the write-data on the other pair of 7-segment displays. Use one of the pushbuttons to advance the state machine so that we can see each state change
  - Remember to use "strictly synchronous" design! The clock signal to your memory and to your FSM should be the system 50MHz clock. Use the pushbutton as input to your FSM so that it only advances to the next state when the button is pressed. Think about how to make the FSM advance only one state for each press and release of the button. And remember that the push-buttons are active-low: the signal is high until the button is pressed and goes low when you press.

Deliverables

- Turn in a photo or scan of your datapath showing how your ALU, RF, and other system registers are connected through MUXes or other structures.
- Turn in a list of all control points in your datapath that your decoder and control FSM will eventually be connected to.
- Turn in Verilog code for your 1k x 16 memory, and the FSM that demonstrates reading and writing of this memory on the DE1-SoC board.
- Demonstrate your memory on the DE1-SoC board to the TA or instructor.