# CS/EE 3710 — Computer Architecture Lab
## Checkpoint #2 — Datapath Infrastructure

### Overview

In order to complete the datapath for your *insert-name-here* 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, the memory access process needs to be figured out (at least the initial version that uses 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 when 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. In this machine I/O is 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.

The memory map for one example of our baseline machine is shown in the slides on the web 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 (or the low half of the 16-bit value since there are only 8 LEDs) 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 the UCF file) to the LEDs on the Spartan3e board.

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 slider switches into the register. I would probably map the sliders to the lowest nibble of the register, and the four pushbuttons around the rotary switch to the next nibble. That way you can read, under program control, the value of 8 different switches and buttons on the Spartan3e board.

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 lab2 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. In particular, anything in the range C000 to C0FF will go to the LED register and come from the switches. 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.

**Block RAM interface**

The most convenient memory to use for your processor is the Block RAM that lives on the Spartan3e chip. We have a Spartan3e500 FPGA which has 20 Block RAM blocks on the chip. Each block is 18k bits. The RAM is configurable in a large number of ways (see the Mini-MIPS slides on the class web site, slide 23 or so). The most convenient for our 16 bit processor is probably the 1k x 16 version. You can actually make each block 1k x 18 if you have any need for those extra two bits. See the slides for details about the read-first/write-first issues, and remember that the block RAMs are clocked on both reads and writes.

You can instantiate Block RAM in your project in (at least) two ways:

1. Inference with behavioral Verilog: This is the technique we used in exmem.v in lab2 with mips. The exmem.v code was written according to a specific Verilog template that ISE understands. Using that template will result in Block RAM being used on the chip. You can get that Verilog template from the ISE tools. Go to Edit→LangaugeTemplates in the ISE WEBpack tool. Then select Verilog→Synthesis Options→Coding Examples→RAM→Block RAM→Single Port to see all the different Verilog code templates. You'll see that there are Verilog behavioral templates for all the different types of Block RAM configurations (single/dual port, no-change/read-first/write-first, examples with enables). You can initialize the behavioral Block RAM using the $readmemh or $readmemb command as documented in the Verilog code template.

2. Direct instantiation of a Block RAM: In this technique you directly instantiate the Block RAM module in your Verilog code by name. To see examples of these start with the Edit→LangaugeTemplates in the ISE WEBpack tool. Then open Verilog→Device Primitive Instantiation→FPGA→RAM / ROM→Block Ram→VirtexII/II-PRO Spartan-3/3E. From there you can see different examples for dual port and single port varieties. In the single-port category, for example, you can see an example of instantiating a 1kx16+2 RAMB16_S18 component in Verilog. You'll see that the mechanism for initializing the directly instantiated Block RAM is different than for the behavioral Block RAM. For direct instantiation you use the .INIT statements to specify the initial contents of the RAM.

As a side note, there are also nice Verilog templates for making smaller RAMs with "Distributed RAM" which can make nice register files. Distributed RAM is made using the Look Up Tables (LUTs) inside the logic cells on the Xilinx part.

Back to the main point, you need to pay attention to the total amount of Block RAM available on the Spartan part. Our part has 20 Block RAMs, each one is 18k bits. So, if you configure each of them as 1k x 16, then you can get a max of 20 of those 1k x 16 blocks. That doesn't fill up the available memory space of your processor. That's 20k addresses and your 16 bit address space allows 64k addresses. You have two alternatives: live within 20k words of code/data space, or use other, external, types of memory. There are good and bad points for each of these options.

**Living in 20k words with VGA:** This is probably fine as long as you don't plan on large, colorful VGA frame buffers. It will be hard to write applications that actually consume more than 20k of code/data space. But, if you use a VGA frame buffer mapped into your processor's RAM space, that can chew up space in a hurry. Consider 640x480 (standard VGA, low res by today's standards) is 307,200 pixels. If you use the VGA connector on the Spartan3e board, you get only one bit each for R G and

B. So, at 3 bits per pixel, and even packing six pixels in 18 bits (by using the parity bits of the Block RAMs and making an 18 bit word at each location), that's 50k (where k is 1024) locations to store the entire 640x481 VGA frame buffer. Oops. We only have 20k words total! Not only does 50k not fit in our available block RAM, even if it did, we wouldn't have any room left over for code!

**Reduced Resolution VGA:** What about 320x240 resolution? In that case you would map each "pixel" to a 2x2 location on the screen. That is, you'd see big chunky pixels on the screen. In that case you need 76,800 pixels, which is 12.5k 18-bit words. That's much more doable! That way you could devote a little more than half of your address space to the VGA buffer and still have 7.5k words left for code/data which would probably be fine.

Of course, you can play with even chunkier pixels to reduce the frame buffer size even more, but you see the issue.

**Character/Glyph maps:** Another option is to have each pixel be an address into a character/glyph map. In this technique you'd store a number of these character/glyphs in a separate piece of memory, then use the pixel to reference into those. In this mode you might have, say, 8x8 character/glyph maps, and then have 80 characters per line, and 60 lines on the screen. In that case you would only need 80 x 60 = 4800 locations needed for your 640x480 VGA frame buffer, and each location would look up an 8x8 character/glyph in memory. If you had, say, 256 characters/glyphs to work with, you'd need 8 bits for each screen location which would pack into two locations per word and you'd need 2400 memory addresses for the frame buffer. You'd also need to store the character/glyph table in memory and that would take some space, 8 x 8 x 256 bits to be exact (16k bits which is 1k words). This is a great option if you can live with each screen location only showing one of 256 possible characters/glyphs.

**Using external SDRAM memory:** Your Spartan3e board has a large SDRAM chip on the board. This SDRAM (Synchronous Dynamic RAM) is 512 MBit organized as 32M x 16 bits. This is huge memory space compared to our relatively puny 64k word address space. The considerable downside is that SDRAM is horrendously complex to use. It's not designed to just put one address on the RAM and get one word back. It's optimized for setting up burst accesses and then getting pipelined bursts of data back for doing things like cache line refills. There are SDRAM controllers that can make SDRAM look like simple static RAM, but even they are complex. We're currently working to get one working but it's not completely ready yet. Even when that does start to work you need to be prepared to deal with the SDRAM in bursts of at least 32 bits (i.e. two 16-bit words it the smallest unit you can deal with because they are DDR (Double Data Rate) SDRAMs that pass data on both edges of the clock). You also need to be ready to deal with slow accesses. Because of the internal construction of dynamic RAMs, reading single random words take a lot longer than streamed burst accesses. Reading a single work might have a latency of 2-20 clock cycles at 50MHz. You will also be interrupted for periodic refreshes of the RAM at approximately $7\mu s$ intervals. Even with a nice SDRAM controller, you will still need a small state machine of some sort to deal with the memory. At the very least you'll need to assert control signals like r/w and then wait for a return acknowledge to let you know that the data is ready because it won't always be ready in the same number of cycles.

**Using external SRAM memory:** We have some 256kbit (32k x 8) static RAMs (SRAM) available in the lab that you could wire up on the external connector board. You could configure your circuit on the FPGA to talk to the SRAM through the 100-pin connector that goes from the FPGA board to the external board. SRAM is much simpler to deal with than SDRAM. It looks a lot like the Block RAM, and is even simpler in the sense that the reads are not clocked. The big downside here is that the SRAMs are 5v parts and the Xilinx FPGA is a 3.3v part. You can probably drive the inputs of

> the 5v part with the 3.3v outputs from the Xilinx part, but you CANNOT drive 5v signals from the 5v part back into the Xilinx part! If you put a 5v signal on the input pad of the Xilinx part you'll destroy it! If you want to drive a 5v signal back in to the Xilinx part you must use a 300Ω resistor in series with that signal to limit the current that is being delivered to the Xilinx part. This will limit the input current to a safe level for the internal protection diodes in the Xilinx part. However, the external SRAMs use bidirectional signals on the data lines so this is a problem! We're still looking for general solutions to this problem.

The bottom line is that you will certainly use Block RAM for part of the memory space for your processor, and perhaps you can live completely within the 20k words that you have on the chip. So, you should at least make your processor work with Block RAM as a first step, and think hard about what your application will require in terms of additional 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. I recommend putting all of the datapath into a symbol and bring out all the necessary control points to pins of the symbol. This way, when you build the decoder and control state machine, you have a well-defined interface between the control and the datapath. The decoder just takes the instruction word and produces control bits, and the finite state machine controller just sequences through other control points (like register enables) and makes the instruction happen. (note that the decoder and control state machine are for another checkpoint, but you should be thinking about them!)

I'd like to do a check-off with each group during our meetings. For your checkoff be prepared to show schematics of your completed datapath, talk about each control point, and describe how each of the instructions will be executed on your control path. In addition to the schematics, make a block diagram of your datapath, and have a separate sheet of documentation on the control points of your datapath. That is, the control point document will have a list of all the control points of the datapath, what they do, and when they are set. This should be considered part of the design documentation and should be done neatly (i.e. not scribbled on the back of a napkin). The TA and I will quiz you about random instructions and you should be able to describe how they will be implemented using your datapath. We'll also ask specific details of your memory interface. Bring schematics, and test logs. Your design does not need to be complete during your checkoff. I just want to make sure that everyone is on the right track!

You should also have a good memory plan in place. You should be able to talk to Block RAM at a minimum, and then make an argument that you will be able to live with 20k total Block RAM space, or argue how you will use external or other memory sources.