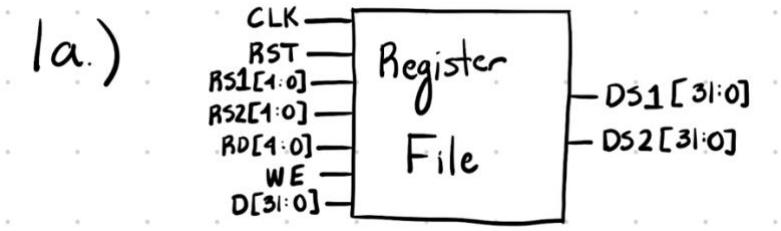
CprE 381, Computer Organization and Assembly-Level Programming

Lab 2 Report

Student Name **CONNOR J. LINK**

Submit a typeset pdf version of this on Canvas by the due date. Refer to the highlighted language in the lab document for the context of the following questions.

[Part 1 (a)] Draw the interface description (i.e., the "symbol" or high-level blackbox) for the RISCV register file. Which ports do you think are necessary, and how wide (in bits) do they need to be?



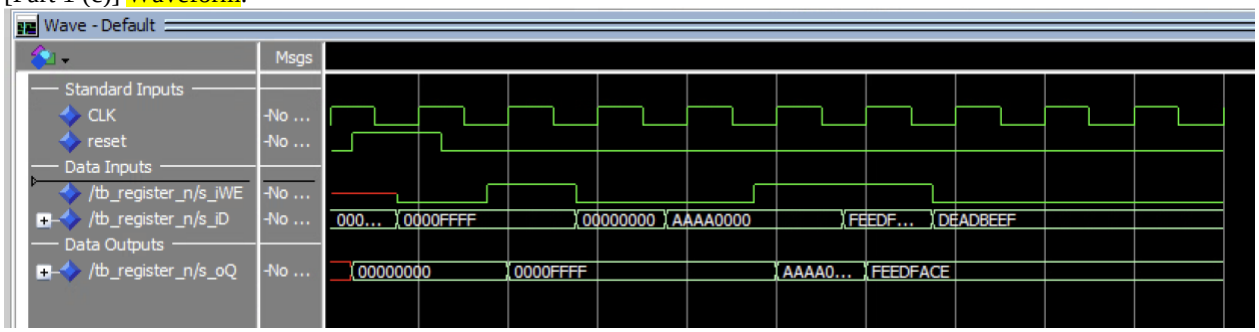
Required Ports:
Inputs:
Clock signal
Reset signal
RS1 (read signal 1): 5-bit → 32 RISC-V registers
RS2 (read signal 2): 5-bit
RD (write signal): 5-bit
Write Enable: WE
Data (D): 32-bit

Outputs:
DS1 (read data 1): 32-bit
DS2 (read data 2): 32-bit

[Part 1 (b)] Create an N-bit register using this flip-flop as your basis.

My VHDL module code, its entity testbench, and the associated TCL macro file are provided in the .zip. As expected, I used structural modeling with generate statements and the provided DFF logic to create the register design.

[Part 1 (c)] Waveform.



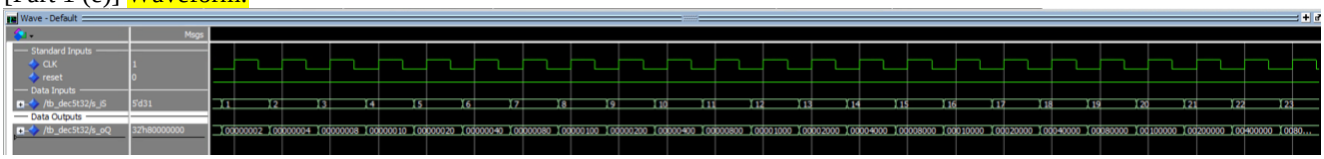
Module Under Test: register_N (RegFile/register_N.vhd)
Testbench File: tb_register_N (RegFile/tb_register_N.vhd)

The above demonstration shows my register design working correctly under a few basic test cases. As anticipated of a D-flip-flop-based design, the input data are delayed until the next clock—or in this case, positive edge—and then output only when the write enable is high. Else, the output data are retained across clock edges.

[Part 1 (d)] What type of decoder would be required by the RISCV register file and why?

It will require a 5:32 decoder as there are thirty-two registers from which to select for reading and writing (x0–x31). Since $log_2 32 = 5$, this requires five bits of input information.

[Part 1 (e)] Waveform.



Module Under Test: dec5t32 (RegFile/dec5t32.vhd)
Testbench File: tb_dec5t32 (RegFile/tb_dec5t32.vhd)

The above waveform correctly shows the decoder iterating through its consecutive, increasing one-hot-encoded outputs as the input signal goes from 0–31 through the 5-bit binary input signal.

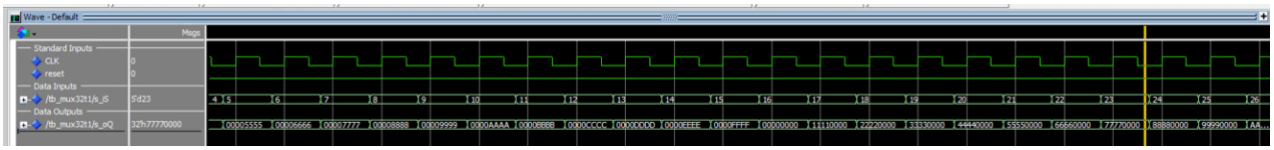
[Part 1 (f)] In your write-up, describe and defend the design you intend on implementing for the next part.

I intend on using a dataflow-based design for the multiplexer to avoid needing to write out the thirty-two cases manually. I know that our VHDL toolchain and its standard library already possess powerful synthesis machinery to infer multiplexers, so theoretically it should be possible to effectively use the five-bit select signal as an "array index."

So, instead of *o_Q <= i_D(0) when i_S = b"00000" else …*

I can do something like *o_Q <= i_D(i_S);*. **My first tries with this have given compile errors, so I'll have to do some research to get it working. But in its defense, this solution will save a lot of repeated typing and reduce the chance of error on my part.**

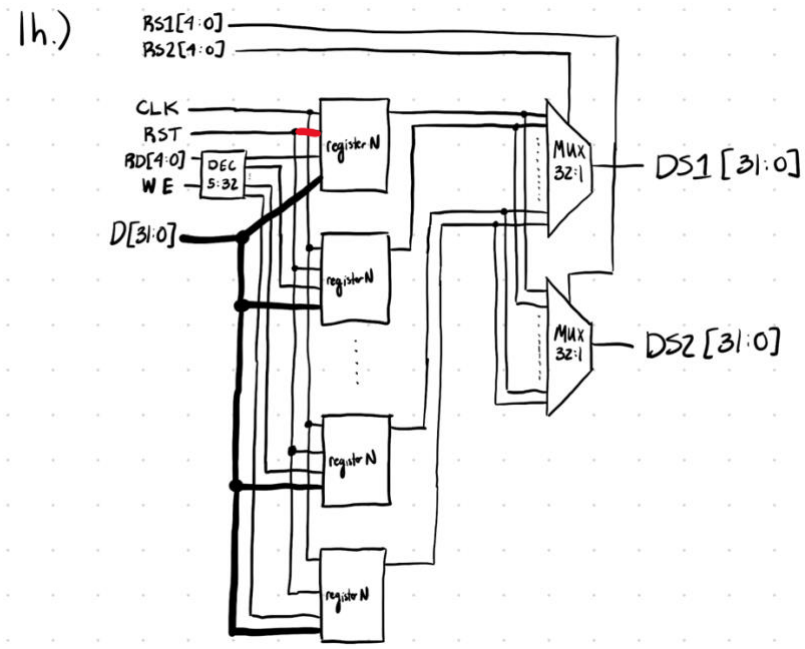
[Part 1 (g)] Waveform.



Module Under Test: mux32t1 (RegFile/mux32t1.vhd)
Testbench File: tb_mux32t1 (RegFile/tb_mux32t1.vhd)

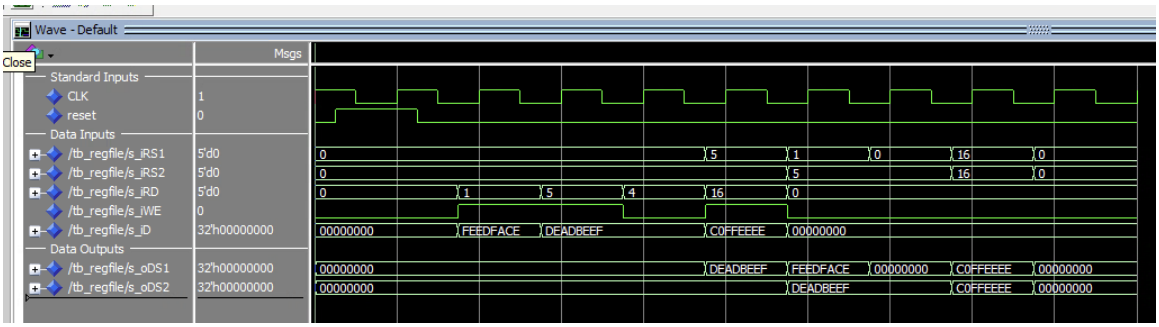
The above waveform depicts my 32-bit 32:1 multiplexer working correctly. Note: I discovered that my dataflow model requires an unsigned integer cast to compile and run correctly.

[Part 1 (h)] Draw a (simplified) schematic (i.e., components within the high-level blackbox) for the RISCV register file, using the same top-level interface ports as in your solution describe above and using only the register, decoder, and mux VHDL components you have created.



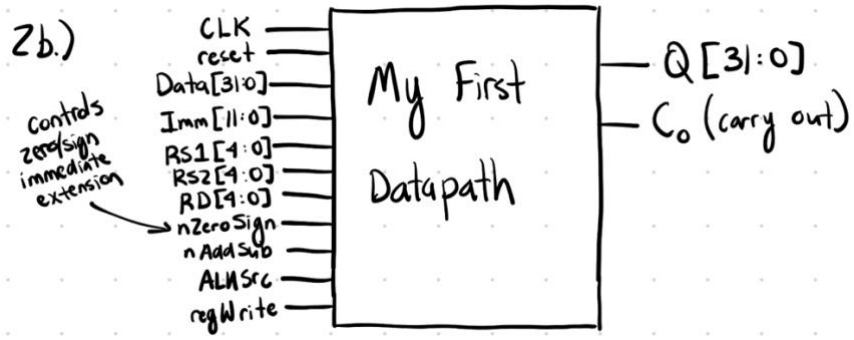
The internal design is fairly simple: the read port selectors control the output 32-bit 32:1 multiplexers (to both of which all registers are connected), and the write port selector controls the input decoder to choose which register to write to based upon the WE enable signal. The clock, reset, and input 32-bit data ports are all connected in parallel to each register. For brevity, I've elided most registers save for R_0, R_1, R_{n-2}, R_{n-1}. Note that the reset line is asserted for register x0 to ensure that it conforms to the architecture laid forth by RISC-V.

[Part 1 (i)] Waveform.



Module Under Test: regfile (RegFile/regfile.vhd)
Testbench File: tb_regfile (RegFile/tb_regfile.vhd)

[Part 2 (b)] Draw a symbol for this RISCV-like datapath.



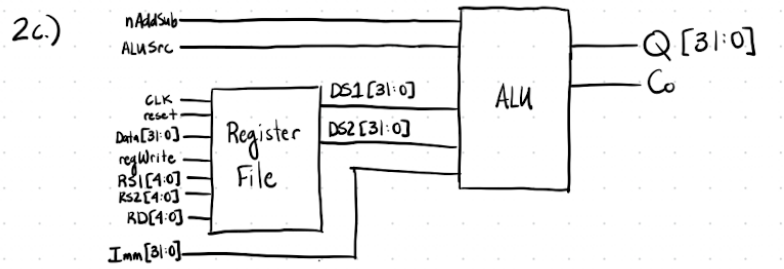
Since this first datapath is merely a loose wrapper atop the register file and "ALU," the overall block signals for this entity are broadly just a combination of these constituents.

Required Ports:
Input:
-Clock signal
-Reset line
-Data input: 32-bit
-Immediate data: 12-bit
-RS1 (Read selector 1): 5-bit → 32 RISC-V registers
-RS2 (Read selector 2): 5-bit
-RD (Write selector): 5-bit
-nZero_Sign (controls whether the extender is in zero- or sign-extend mode)
-nAdd_Sub (controls whether the "ALU" is set to add or subtract its operands)
-ALUSrc (controls whether the B operand or the extended immediate is sent to the "ALU")
-regWrite (controls the write enable signal for the embedded register file)

Output:
-Q: 32-bit
-Carry out

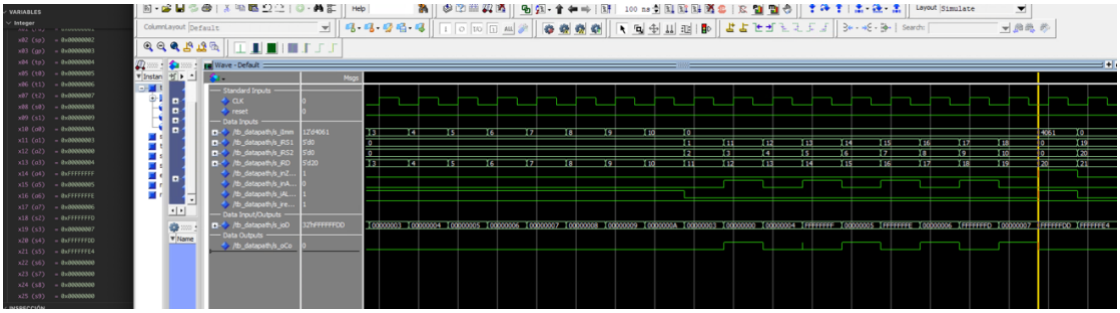
NOTE: To feed results from the ALU back to the registers, the Q signal will ultimately be connected back to this unit's data input. However, I think it makes sense here to also note it as an output signal.

[Part 2 (c)] Draw a schematic of the simplified RISCV processor datapath consisting only of the component described in part (a) and the register file from problem (1).



NOTE: To feed results from the ALU back to the registers, the Q signal will ultimately be connected back to this unit's data input. However, I think it makes sense here to also note it as an output signal.

[Part 2 (d)] Include in your report waveform screenshots that demonstrate your properly functioning design. Annotate what the final register file state should be.



Module Under Test: datapath (MyFirstRISCVDatapath/datapath.vhd)
Testbench File: tb_datapath (MyFirstRISCVDatapath/tb_datapath.vhd)

The correct register states are annotated in the ground-truth Venus window on the left.
Specifically they should be, incrementing from x0: *0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0x3,0x0, 0x4, 0xFFFFFFFF, 0x5, 0xFFFFFFFE, 0x6, 0xFFFFFFFD, 0x7, 0xFFFFFFDD, 0xFFFFFFE4, 0x0, …, 0x0.*

The penultimate trace on signal */tb_datapath/s_ioD* tracks the register output values as they are written and depicts this configuration correctly.

[Part 3 (a)] Read through the mem.vhd file, and based on your understanding of the VHDL implementation, provide a 2-3 sentence description of each of the individual ports (both generic and regular).

DATA_WIDTH: specifies—as an integer greater than zero, a `natural`—the number of bits stored for each "word" of the memory block. For our purposes, this is defaulted to 32, meaning that each memory address represents a 4-byte word. This must be considered in the memory addressing logic, for data are not single-byte-addressable under this scheme.

ADDR_WIDTH: specifies—an integer greater than zero—the bit width of the address bus used to index the memory array. Since this signal is encoded as binary, the total number of selectable addresses is calculated as 2^{ADDR_WIDTH} words. As the default is 10 bits for our case, we can address a total of 1024 individual 4-byte words. Again, we have to consider that this means 1024 addresses, and not 1024*4=4096 as the total size might indicate.

CLK: Fairly self-explanatory: this is the input clock signal. While reads are asynchronous per the continuous dataflow assignment on line 48 of *mem.vhd*, writes are synchronized to the positive edge of the clock signal.

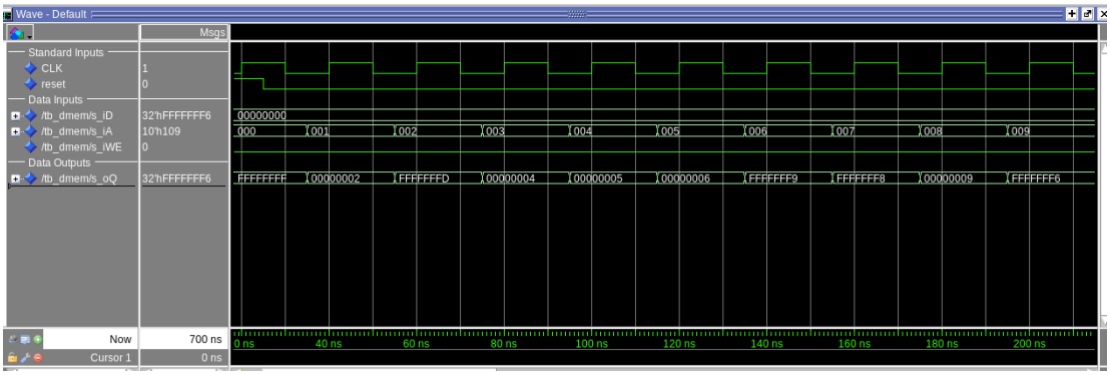
ADDR: As outlined above, this is a data bus input of *ADDR_WIDTH* bits' width. It represents the incoming address bus signal and is used to uniquely index each word of the memory array through dataflow assignment.

DATA: This is the input data bus of width *DATA_WIDTH* bits and is used as the source for writes to the memory block. Note that writes are rising-edge synchronized.

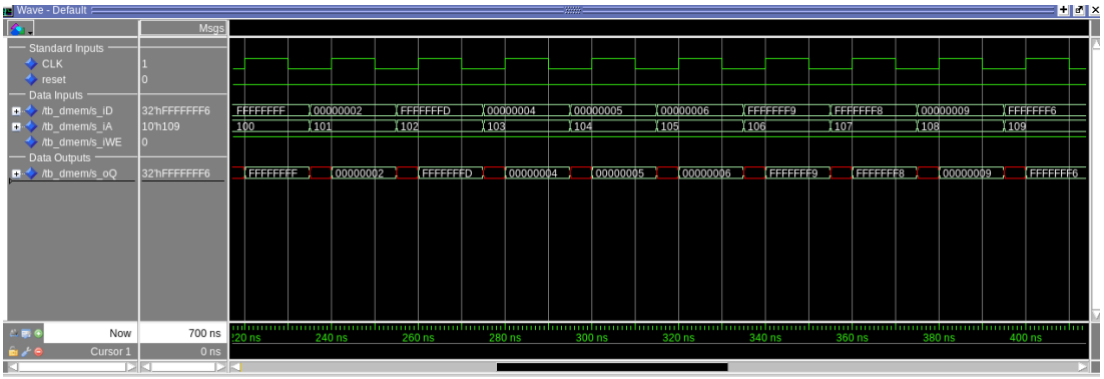
WE: This is the write enable signal for the memory. When it is low, the memory is in asynchronous read-only mode, but when it is high, the data at the currently selected address will be overwritten with the contents of the input data bus *`data`* upon the next clock rising edge.

Q: This is the output data signal of width *DATA_WIDTH*. It will always be driven to represent the contents of the data memory at the specified input address, for it is asynchronously assigned as such.

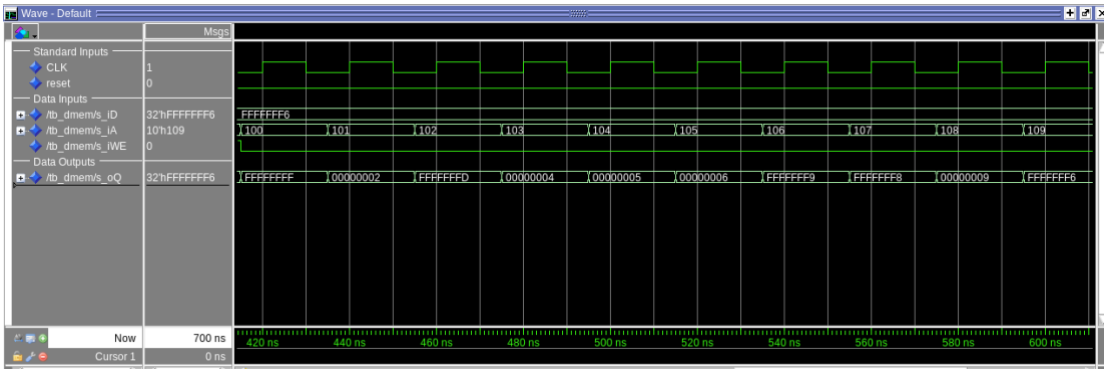
[Part 3 (c)] Waveforms.
Module Under Test: dmem (dmem.vhd)
Testbench File: tb_dmem (tb_dmem.vhd)



Test Case (ii): Read initial ten values stored in memory



Test Case (iii): Write those same values back to consecutive locations in memory starting at $100



Test Case (iv): Read those new values back to ensure they were written properly

[Part 4 (a)] What are the RISCV instructions that require some value to be sign extended? What are the RISCV instructions that require some value to be zero extended?

Any instruction that utilizes immediate data encoding may utilize either zero- or sign-extended values. Per the developer's manual at https://github.com/riscv/riscv-isa-manual/blob/main/src/rv32.adoc, the base integer instruction set uses this for:

Sign-extension:
-ADDI
-SLTI
-ANDI
-ORI
-XORI
-JAL
-JALR
-Conditional Branches (BEQ, BNE, BLT, BGE)
-LW/SW (for effective address computation)

Zero-extension:
-LHU (after read for writeback)

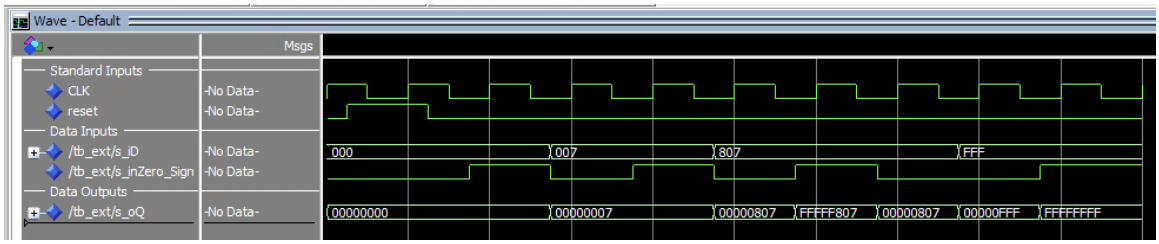
There are few examples for zero extension, for the authors specifically note that *"Immediates are sign-extended because we did not observe a benefit to using zero extension for some immediates as in the MIPS ISA and wanted to keep the ISA as simple as possible."*

[Part 4 (b)] what are the different 12-bit to 32-bit "extender" components that would be required by a RISCV processor implementation?

This isn't an exhaustive list, but an extender will be required in *at least* the following places:
-ALU (for immediate arithmetic instructions)
-PC/Control logic (for branch address calculation)
-Memory address generation unit (if we don't reuse ALU for this)

These extenders will need to be able to zero-extend their input (padding MSBs with zero) and sign-extend their input (padding MSBs with the MSB of the input) by selecting with another signal. For our purposes, *only sign-extension should be required for the base RISC-V integer instruction set,* but I believe that having the other functionality could possibly prove useful later on, so I'll implement it nonetheless.

[Part 4 (d)] Waveform.



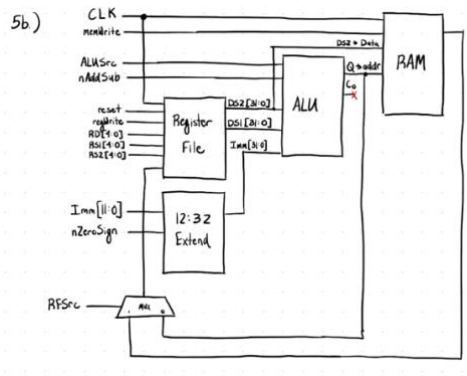
Module Under Test: ext (12:32 bit) (Extenders/ext.vhd)
Testbench File: tb_ext (Extenders/tb_ext.vhd)

[Part 5 (a)] what control signals will need to be added to the simple processor from part 2? How do these control signals correspond to the ports on the mem.vhd component analyzed in part 3?

Signals to Add:
Memory Write (memWrite): *this will attach to the `we` signal on the memory entity and will be used to overwrite the contents of a memory with the provided data at the selected address in memory.*

Register File Source (RFSrc): *this will attach to a new multiplexer to select which data to input to the register file data bus. 0 is the previous output from the ALU directly; 1 is the output from the data memory block at the calculated address from the ALU.*

[Part 5 (b)] Draw a schematic of a simplified RISCV processor consisting only of the base components used in part 2, the extender component described in part 4, and the data memory from part 3.



[Part 5 (c)] Waveform.

Code: To test the second datapath, I used the following RISC-V assembly code in RARS. I've had to use assembler directives to replicate loading the *mem.hex* file as this appears unsupported in RARS.

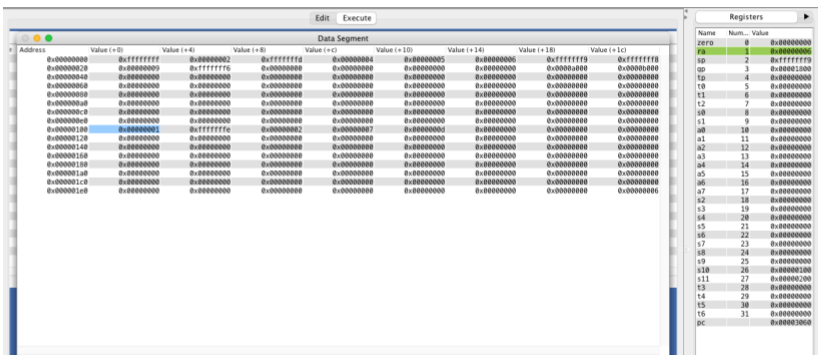
Module Under Test: datapath2 (MySecondRISCVDatapath/datapath2.vhd)
Testbench File: tb_datapath2 (MySecondRISCVDatapath /tb_datapath2.vhd)

```
.data
#.org 0
.word 0xffffffff
.word 0x00000002
.word 0xfffffffd
.word 0x00000004
.word 0x00000005
.word 0x00000006
.word 0xfffffff9
.word 0xfffffff8
.word 0x00000009
.word 0xfffffff6

.space 16
#.org 15
.word 0x0000a000
.word 0x0000b000

.text
addi x25, x0, 0   # Load &A into x25
addi x26, x0, 256 # Load &B into x26
lw x1, 0(x25)     # Load A[0] into x1
lw x2, 4(x25)     # Load A[1] into x2
add x1, x1, x2    # x1 = x1 + x2
sw x1, 0(x26)     # Store x1 into B[0]
lw x2, 8(x25)     # Load A[2] into x2
add x1, x1, x2    # x1 = x1 + x2
sw x1, 4(x26)     # Store x1 into B[1]
lw x2, 12(x25)    # Load A[3] into x2
add x1, x1, x2    # x1 = x1 + x2
sw x1, 8(x26)     # Store x1 into B[2]
lw x2, 16(x25)    # Load A[4] into x2
add x1, x1, x2    # x1 = x1 + x2
sw x1, 12(x26)    # Store x1 into B[3]
lw x2, 20(x25)    # Load A[5] into x2
add x1, x1, x2    # x1 = x1 + x2
sw x1, 16(x26)    # Store x1 into B[4]
lw x2, 24(x25)    # Load A[6] into x2
add x1, x1, x2    # x1 = x1 + x2
addi x27, x0, 512 # Load &B[64] into x27
sw x1, -4(x27)    # Store x1 into B[63]
nop
```

Golden Sample: Below is depicted the reference output from RARS for the above assembler code provided in the lab manual. Note especially the register and data page memory contents (starting at address $00000000).



QuestaSim Results: See below the memory dump from Questa after the testbench had been fully simulated through all the above instructions. Comparing to the known-good results from the emulator above, my implementation matches as expected.