ELEC 326 Labs 3 and 4

16- bit Processor

Objectives

Work in Xilinx Vivado and design a simplified processor in Verilog. The full architecture to be implemented is described at the end of this document. The steps for the lab will guide you through the design of the instruction decode, ALU, register file, branch comparator, and program counter blocks. Several test programs will be provided.

Inputs/Outputs

The individual inputs and outputs of the top processor module are already connected and are described below. These are the same inputs and outputs as the previous labs, though they are used differently; besides the clock, all of the inputs and outputs are part of the data memory map for the processor and are accessible with load and store operations. For more details, see the Appendix: Memory Map below.

Inputs to the Processor

- CLK – the onboard 100Mhz clock
  - As a non-pipelined processor, your code may not run at 100MHz. A clock enable signal is provided to each of the sequential logic blocks to reduce the system clock frequency.
  - The **clk_en** signal divides the system clock frequency down to about 24 kHz – this is fast enough for the lab and all test programs provided.
- SW[7:0] – the eight board switches
  - The switches are accessible as a memory mapped register, by reading from data address 0x8000
- BTN[4:0] – the five board push buttons
- CPU_RESETN – CPU Reset N (Active Low)
  - Resets all CPU state (registers, PC, etc.) and begins code execution again from program address 0x0

Outputs of the Processor

- LED[7:0] – the eight board LEDs
  - The LED bits are accessible as a memory mapped register, exercised by writing to data address 0x8000.
    - Yes, this is the same data address as the switches. As one is accessible when reading from 0x8000 and one when writing to 0x8000, there is no conflict.
- SEG[6:0], DP, AN[3:0] – the seven segment display
  - Accessible as a memory mapped register by writing to 0x9000. The seven-segment display formatting is already implemented, as in Lab 2.

Provided files

Download **processor.v, data_flow.v, instruction_mem.v, data_mem.v**, and **Nexys4DDR_Master.xdc** from the course webpage.

In this lab, the top-level **processor.v** is already complete. Please read and become familiar with the modules.

Additionally, **data_mem.v** is already complete. The **instruction_mem.v** file has several example programs written into it that are commented out – these programs can be used at different steps in the lab below to test that your code is working

The **data_flow.v** file is where the bulk of the work in this lab will be written. All of these modules are instantiated and connected in the **processor.v** file; your task will be completing the five data flow modules. Each of these modules, except for the **instruction_decode** module (Step 1), defaults to driving useful values on its outputs so that the system may be tested at the end of each step – *e.g.*, the register file module, before Step 2, drives a constant 0x9000 as the read value for all registers, allowing the seven segment display to be written after implementing only the **instruction_decode** submodule.

Besides the provided Verilog and constraint files, an assembler is also provided. It is java based, so should run on your own computers or the lab machines, and three test programs are provided, corresponding to the three test programs coded into the **instruction_mem.v** file. The assembler is provided if you wish to write your own test programs.

Step 1: Instruction Decode

The **instruction_decode** submodule takes one input (the 16 bit instruction) and drives twenty-eight outputs. All logic in the instruction decode block should be combinational; there is no clock input to the module. The full description of the instructions is at the end of the document in the "**Appendix: Instruction Set**", but here is a brief description of some of the output signals:

- **alu_func**: the FUNC bits of the 1-input and 2-input arithmetic operations
- **destination_reg**: the Rd bits for all operations. This should be the same set of bits regardless of op-code; all operations that write to the register file use this as the destination register, and the STORE command uses these bits as the data register (so both Load and Store use Rd as the register to load from or store to the data memory)
- **source_reg1/2**: These bits are consistent across most op-codes. The only exceptions are the branch op-codes – make sure that when assigning the bits for the two **source_reg** outputs that you pay attention to the alignment in the arithmetic op-codes vs the branch op-codes.
- **Immediate**: These bits represent an integer to be used directly rather than a reference to a register (*e.g.* ADD sums the values in two registers, ADDI sums the value in a register with a 6-bit immediate). They are always the least significant bits of any instruction. Sign extension should not be performed in the **instruction_decode** module, but should be left to other blocks For instance, the branch/jump immediate values are sign extended, but it is expected that this will occur in the program counter block. In fact, the ONLY instructions for which the immediate is sign extended are branches and jumps.

All other outputs should be self-explanatory, and are Boolean values based upon a single op-code or op-code+function. Refer to the instruction set Appendix for further details. For decoding purposes, it may be useful for you to use the `*define constants* provided at the beginning of the **data_flow.v** file.

When you have finished your **instruction_decode** block, you should be able to execute a simple program that does not require the ALU, register file, or any branches. This is the first program initially provided to you in the **instruction_mem** file. After programming your board, "CAFE" should be displayed on the seven-segment display.

At this stage you will see quite a few warnings about unused signals or logic removed during synthesis. This is expected, as much of the logic for the processor is not being exercised yet.

Step 2: Register File

The register file module, **reg_file**, is a block of eight 16-bit registers.

A description of its inputs and outputs follows:

- **clk** & **clk_en**: the 100 MHz clock and a clock enable signal. The data written into a register should only occur on the clock edge when the **clk_en** signal is active
  - Effectively, this is providing a slower clock to this and the other sequential blocks in the processor. This is a method of dividing down a clock to run logic at a slower frequency.
- **reset**: Active high. When this input is a 1, reset the registers to all zeros.
- **source_reg1**: A 3-bit identifier for a register.
- **reg1_data**: The data corresponding to the register # provided by the **source_reg1** input. This should NOT be registered as an output, and should be combinational based on the input **source_reg1** value.
- **source_reg2/reg2_data**: Analogous to **source_reg1** / **reg1_data**
- **destination_reg**: A 3-bit identifier for a register
- **wr_destination_reg**: When true, an instruction is writing to a register. ONLY when this signal is true should the values in any of the registers change.
- **dest_result_data**: The 16-bit result of an instruction (ALU output or **data_mem** output) that is to be written to the register specified by **destination_reg** if **wr_destination_reg** is true.
- **regD_data**: Analogous to **reg1_data** or **reg2_data**, but for the **destination_reg** input. This is the value to be written by a STORE instruction.
- **Immediate**: An 8-bit immediate value. Only used for a MOVIL/MOVIH instruction.
- **movi_lower**: The MOVIL instruction. Copies the 8-bit immediate input into the lower byte of
- the destination register.
- **movi_higher**: The MOVIH instruction. Copies the 8-bit immediate input into the upper byte of the destination register.

It is advisable for this module to have two sections of code – combinational logic that reads the values for the data outputs (**reg1_data/reg2_data/regD_data**), and sequential logic that handles the register write-back logic. A write-back to a register should only occur if **clk_en** is true and **wr_destination_reg** is TRUE. The **movi_lower** and **movi_higher** commands take precedence over **dest_result_data** (if **movi_lower** or **movi_higher** is true, the **dest_result_data** is not valid).

When you have finished your **reg_file** block, you should be able to execute the first simple test program again. After programming your board, "BEEF" should be displayed on the seven-segment display. It is the same test program as Step 1, but with the additional functionality of the register file, it will no longer display the previous tie-off value of "CAFÉ", but the values copied into the registers with the MOVIH/L instructions.

Step 3: ALU

The arithmetic logic unit (ALU) is primarily combinational logic. There are two registered logic values, however – the borrow bit and carry bit.

The carry and borrow bits are used in 16-bit systems like the one you are designing to allow for 32-bit logic; successive ADDC commands allow the carry bit from one addition to be propagated to the next one. The carry bit can be set by any addition operation (ADD, ADDC, ADDI), and the borrow bit can be set by any subtraction operation (SUB, SUBB, SUBI), but the bits are only used during ADDC or SUBB instructions.

A description of the inputs and outputs:

- **clk/clk_en/reset**: The same function they served in the **reg_file** block. The **clk** and **clk_en** should be used when registering the carry and borrow bits, and the **reset** input should be used to set both of those state bits to zero.
- **alu_result**: The main output of the ALU. This 16-bit value should be the computational result for the current instruction.
- **arith_1op/arith_2op/addi/subi**: Each of these 1-bit signals is TRUE if and only if it holds for the current instruction op-code
- **load_or_store**: True if the current instruction op-code is either LOAD or STORE. In this case, the **alu_result** should be the same as *addi*, except that the carry bit should NOT be modified. The ALU is used for load/store instructions to calculate the address in the data memory; these instructions allow an immediate offset from a register so that data can be accessed by specifying an offset from a fixed base register.
- **reg1_data**: The first operand for a 2op instruction, or the only operand for a 1op instruction, add/subtract immediate, or LOAD/STORE.
- **reg2_data**: The second operand for a 2op instruction.
- **Immediate**: The six bit immediate for add/subtract immediate, or for the LOAD/STORE address offset.
- **stc_cmd/stb_cmd**: Command instruction to set carry bit or borrow bit, respectively.
- **alu_carry_bit**: The carry bit of the ALU, output so it may be used in the branch comparator.

Most of the code for this block should be combinational. You should have a separate sequential block for the carry/borrow bit logic. Once you have finished the ALU implementation, comment out the first test program in **instruction_mem.v**, and uncomment the test program labeled as "Test Program for Step 3". The display output for the step 3 test program should read 0xDFF8 on the seven-segment display.

Step 4: Branch Comparator and Program Counter

The final step to complete the basic processor involves two modules: the branch comparator and the program counter. The two cannot be tested separately, as you are implementing two things that are interrelated: determining whether a branch is taken, and updating the PC so that a branch or jump can have effect.

The first part is fairly simple – the branch comparator has four inputs for the four branch op-codes, input data to determine branch outcome, and a single output (**branch_taken**) that should be true if a branch op-code is active and its conditions are true.

The four branch instructions are:

- branch_eq: Branch if the two **reg_data** inputs are equal
- branch_ge: Branch if **reg1** is greater than or equal to **reg2**
- branch_le: Branch if reg1 input is less than or equal to reg2
- branch_carry: Branch if the ALU *carry* bit is set. This is particularly useful for overflow checks.

There is no way to directly test that this code is working until you have also updated the *program counter module*, as the **branch_taken** flag is an input to the **program_counter** module that you will need to handle before changes to the branch comparator module have any effect on the system.

The sequential logic for the **program_counter** module is already present in the module, and in the code provided to you, **pc_next** is always set to the current pc + 2 (2, because the instructions are 2 bytes wide).

You will need to do two things to finish the **program_counter** module. First, you will need to sign extend the branch and jump immediate signals to 16-bits. Then you will need to modify the code that determines **pc_next** based upon the **jump_taken** and **branch_taken** inputs.

IMPORTANT NOTE: even when a branch or jump is taken, the pc still increments by 2. This is fairly standard across architectures, usually to aid in pipelining. It is expected here as well, and the final test code will not work correctly if you do not do this. Ensure that no matter what you add to the pc to get **pc_next**, you also add 2. (e.g. "BEQ $1 $1 0" should advance to the next command, but "BEQ $1 $1 -2" should be an infinite loop)

There is a final test program you can execute at this point to demonstrate your working processor in the lab. Comment out the test code for the earlier lab steps and uncomment the code below the comment "Final Test Code."

When running the final test program, you should observe the following:

- The switch values should be displayed on the LEDs - there is a loop that reads the switches and writes the values back to the LEDs.

- The 7-segment display should output a Fibonacci sequence that increments every second.
- When the Fibbonacci sequence exceeds what can be shown on the 7-segment display, the CPU halts (so the switches will no longer have any effect on the LEDs).
    - The value when the CPU halts should be 0xB520
    - Immediately before halting, the program sets the 'blink' flag at data address 0x9002. Once the 7-segment display starts blinking 0xB520, the switches should have no effect.
    - Once halted (or any point, really), CPU_RESET can be pressed to reset the system to start all over again.

Step 5: Processor Additions and Assembly

Choose one of the options below. In your report, describe the tradeoffs of implementing in Assembly on this custom processor instead of a pure Verilog based implementation for your chosen option.

1. Assembly. Implement multiplication as a software program in assembly on your new processor. Read values from the sliding switches SW[7:0]. Then perform a 4-bit by 4-bit multiplication using operands from the sliding switches values as a simple software addition loop. Show the result on the right set of seven-segment displays. You should allow for a new set of inputs to be set by changing the switches and re-running your multiply program.

2. Buttons. Add support to enable reading the values of the push buttons. Write a short assembly program to (i) perform addition and subtraction for two 4-bit operands read from the sliding switches SW[7:0] (memory map 0x9000) and (ii) read the system clock (memory map 0xF000). Use buttons BTN[2:0] to select the operation +,-, or show system clock. Simply enter a continuous loop, reading the button and switch values and showing the results of the computation on the on the right set of seven-segment displays. To read values from the buttons, alter **data_mem.v** so that processor memory address 0x9006 reads from the BTN values.

3. Extended I/O. Add support to the enable the set of the left four seven-segment displays, left 8 sliders and LEDs. Write an assembly program that demonstrates them working such as reading switches values and displaying them on the seven segment displays while looping over the LEDs or showing the NOT of the switches on the LEDs. To read/write additional I/O, alter **data_mem.v** as outlined in the Appendix Memory Map, Extended I/O. This will require altering the **data_mem** module to map the additional addresses and extend the data bus, as well as modifying the processor.v to also extend the data. You must also uncomment the associated lines in the design constraint file (Nexys4DDR_Master.xdc) to enable the additional switches and LEDs.

4. Colors, Tri-Color LED, Bonus +5. The Nexys 4 board contains two tri-color LED's. Read sections 10.2 and 16.1 in the Nexys 4 DDR Manual. Construct a new Verilog module from scratch that receives CLK and 8-bit Red, Green, and Blue inputs, and outputs three values to one of the tri-

color LED's pins using pulse width modulation (PWM). Create a simple Assembly program that rotates through various color combinations or changes colors based on the sliding switches.

Do not simply supply constant power to the RGB LED in your custom module. The module should take each component color value and pulse each output corresponding to the eight-bit value - make the value 255 pulse 50% of the time to temper LED brightness, so a value of 128 should pulse the component color high for 25% of the time. Enable the RGB LED in the design constraint file (LED16_R, LED16_G, LED16_B). Alter the **data_mem.v** and **processor.v** files to pass component colors as arguments to the **data_mem** module. Create a new Verilog module in **processor.v** that receives these component colors on a memory map write to addresses 0x9010, 0x9012 or 0x9014, and saves the color components in a register. Use the system CLK to count from 0 to 512, for each component color, enable the component color on only if the value is greater than the current count.

5. <u>Bonus +5.</u> Combine Assembly and Buttons (options 1 and 2) to implement a variant of lab 1 in software assembly code. Use buttons BTN[3:0].

6. <u>Bonus +10.</u> Combine 1, 2, and 3 into an 8-bit calculator. Implement the Extended I/O (option 3). Then add Button (option 2) support and write an 8-bit calculator in Assembly (option 1) as above except using 16 bit values from the switches for the two eight bit numbers. The LEDs should show the values of the switches, the left seven-segment displays should show the two eight-bit numbers and the right seven-segment display should show the result after pushing one of the four buttons BTN[3:0].

Lab Report

Follow the report outline for previous labs. Only include code from your Step 4 and your custom assembly program from Step 5. A lab grader should sign off on Step 4 **<u>and</u>** Step 5.

In your report, describe your Step 5 and the tradeoffs of implementing in Assembly on this custom processor instead of a pure Verilog based implementation for your chosen option.

Document the role/contribution of each lab partner.

For this lab capture 4 Screenshots and Implementation Utilization.

Screenshot 1. RTL Analysis - Schematic. Identify the major system components. Relate to the in-class discussion processor components.

Screenshot 2. Synthesis – Synthesize – Schematic. Make sure you close previous schematics and are viewing the complete design. Capture a screenshot and comment on the complexity.

Screenshot 3. Synthesis – Synthesize – Schematic – Expand ALU. Comment on the ALU complexity compared to the calculator in Lab 1.

Screenshot 4. Device Implemented Design. Make sure to close previous schematics and refresh Implementation and select the top-most module to get the latest design or your changes will not show up. Compare complexity to other labs (light blue areas utilized).

Utilization. Report the total usage of components on the FPGA based on your design. (Implementation – Implemented Design – Report Utilization) Look at the Slice Logic in the Utilization window.

How many Slice Lookup Tables (LUT) and Slice Registers are used?

What is the total percentage of utilization?

Could you put 15 more processors on your FPGA?

How could you communicate with the other processors or a co-processor / specialized processor?

Appendix: Instruction Set

Op-codes

The four most significant bits of the instruction code represent the op-code for the instruction. These op-codes are defined as follows:

Op-code	Mnemonic	Instruction Format	Details
0000	NOP	NOP	
0001	ARITH_2OP	FUNC Rs1, Rs2, Rd	Valid FUNC values: ADD, ADDC, SUB, SUBB, AND, OR, XOR, XNOR
0010	ARITH_1OP	FUNC Rs1, Rd	Valid FUNC values: SHIFTL, SHIFTR, NOT, CP
0011	MOVI	FUNC Rd, #8-bit	Valid FUNC values: MOVIL, MOVIH
0100	ADDI	ADDI Rs1, Rd, #6-bit	
0101	SUBI	SUBI Rs1, Rd, #6-bit	
0110	LD	LD Rs1, Rd, #6-bit	
0111	ST	ST Rs1, Rd, #6-bit	
1000	BEQ	BEQ Rs1, Rs2, #6-bit	
1001	BGE	BGE Rs1, Rs2, #6-bit	
1010	BLE	BLE Rs1, Rs2, #6-bit	
1011	BC	BC #6-bit	
1100	J	J #12-bit	
1101	JL	JL #12-bit	
1110	INT	FUNC #4-bit	Valid FUNC values: INTE, INTD, INT_TRIG
1111	CONTROL	FUNC	Valid FUNC values: RET, STC, STB, HALT, RESET

In the instruction formats above and the Op-code Detail below, the following conventions are observed:

- Register designations (Rs1, Rs2, Rd, etc) represent Register File identifiers. They are three bits indices in all cases.
  - In the assembler, register ids are written with a dollar sign, e.g. “ADD $0 $1 $2”

- #N-bit - an N bit immediate value. May be signed (sign-extended) or unsigned (literal).
  - Example: Jump uses a sign-extended immediate, so J 0xFF8 would jump backwards, as it would be interpreted as "Jump -8". Note that the PC is still incremented, so "J 0x0" would proceed to the next instruction, but "J 0xFFE" would be an infinite loop.
  - On that note, "J 0xFFF" is not valid. The LSB of relative jump or branch commands should always be zero, as instructions are aligned to even addresses.
  - In the assembler, immediate values can be any number of formats (anything that Java's Integer.decode will parse). Examples include:
    - ADDI $0 $0 1 // Increments Reg0 by 1
    - MOVIH $0 0x80 // MOVE 0x80 into the upper byte of Reg0
- Instruction formats are consistent within a single op-code, but may vary between op-codes. The destination register location is consistent (all op-codes that store data in the register file do so with the three bits immediately following the op-code), but the source registers may be located in different bits. You will implement this in the instruction decode block.

Op-code Detail

NOP

Op-code	Reserved
0000	000000000000

No operation.

ADD

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	000

Adds Rs1 and Rs2 and stores the sum in Rd, ignoring any previous carry.

ADDC

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	001

Adds Rs1 and Rs2 and stores the sum in Rd, with previous carry.

SUB

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	010

Subtracts Rs2 from Rs1 and stores the difference in Rd, ignoring previous borrow.

SUBB

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	011

Subtracts Rs2 from Rs1 and stores the difference in Rd, with previous borrow.

AND

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	100

Performs bitwise AND of Rs1 and Rs2 and stores the result in Rd.

OR

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	101

Performs bitwise OR of Rs1 and Rs2 and stores the result in Rd.

XOR

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	110

Performs bitwise XOR of Rs1 and Rs2 and stores the result in Rd.

XNOR

Op-code	RD	RS1	RS2	Func
0001	Rd	Rs1	Rs2	111

Performs bitwise XNOR of Rs1 and Rs2 and stores the result in Rd.

NOT

Op-code	RD	RS1	Reserved	Func
0010	Rd	Rs1	000	000

Performs bitwise NOT of Rs1 and stores the result in Rd.

SHIFTL

Op-code	RD	RS1	Reserved	Func
0010	Rd	Rs1	000	001

Shifts Rs1 by one place to the left and stores the result in Rd.

SHIFTR

Op-code	RD	RS1	Reserved	Func
0010	Rd	Rs1	000	010

Shifts Rs1 by one place to the right and stores the result in Rd.

CP

Op-code	RD	RS1	Reserved	Func
0010	Rd	Rs1	000	011

Copies Rs1 to Rd.

MOVIL

Op-code	RD	FUNC	Unsigned Immediate
0011	Rd	0	#8-bit

Copies immediate value into lower byte of Rd.

MOVIH

Op-code	RD	FUNC	Unsigned Immediate
0011	Rd	1	#8-bit

Copies immediate value into higher byte of Rd.

ADDI

Op-code	RD	RS1	Unsigned Immediate
0100	Rd	Rs1	#6-bit

Adds a 6-bit unsigned value to Rs1 and stores the sum in Rd.

SUBI

Op-code	RD	RS1	Unsigned Immediate
0101	Rd	Rs1	#6-bit

Subtracts a 6-bit unsigned value from Rs1 and stores the difference in Rd.

LD

Op-code	RD	RS1	Unsigned Immediate
0110	Rd	Rs1	#6-bit

Loads Rd from the address given by [Rs1 + 6 bit unsigned value].

ST

Op-code	RD	RS1	Unsigned Immediate
0111	Rd	Rs1	#6-bit

Stores Rd at the address given by [Rs1 + 6 bit unsigned value].

In spite of the name and position, Rd acts as a source register for this command – the value in register Rd is the data to be stored in the data memory.

BEQ

Op-code	RS1	RS2	Signed Immediate
1000	Rs1	Rs2	#6-bit

Branches to the relative PC address given by the signed 6-bit immediate value if the values in Rs1 and Rs2 are equal.

BGE

Op-code	RS1	RS2	Signed Immediate
1001	Rs1	Rs2	#6-bit

Branches to the relative PC address given by the signed 6-bit immediate value if the value in Rs1 is greater than or equal to the value in Rs2.

BLE

Op-code	RS1	RS2	Signed Immediate
1010	Rs1	Rs2	#6-bit

Branches to the relative PC address given by the signed 6-bit immediate value if the value in Rs1 is less than or equal to the value in Rs2.

BC

Op-code	Reserved	Reserved	Signed Immediate
1011	000	000	#6-bit

Branches to the relative PC address given by the signed 6-bit immediate value if the carry bit in the ALU is set.

J

Op-code	Signed Immediate
1100	#12-bit

Jumps to the relative PC address given by the signed 12-bit immediate value.

STC

Op-code	Func
1111	000000000001

Set the carry bit in the ALU.

STB

Op-code	Func
1111	000000000010

Set the borrow bit in the ALU.

RESET

Op-code	Func
1111	101010101010

Reset the CPU.

HALT

Op-code	Func
1111	111111111111

Halt the CPU.

Appendix: Memory Map

The system has a fairly simple memory map – only four functional addresses and no general purpose registers. The ones currently mapped provide inputs and outputs to the cpu that are visible externally, through the LEDs, switches, and display. The **data_mem.v** file already implements this mapping, and you do not need to modify the code. This is provided as a reference if you wish to write your own assembly code for the system and wish to access any of the memory mapped data.

Address	Name	R/W	Function
0x8000	Switches & LEDs	RW	Bits[7:0] Read from SW, the board switches Write to LED, the board LEDs **Bits[15:8] return 0 unless modified to extend I/O to 16.**
0x9000	Right Seven-segment Disp	W	Write a 16-bit number to the seven-segment display, formatted as in the previous labs.
0x9002	Blink	W	Writing a 1 to bit[0] enables the seven-segment display blink functionality from Lab2
0xF000	Uptime	R	Reads the current real-time clock value in seconds. The value is approximate, and is the same seconds counter from Lab2. In a real system, this would likely be more than a 16-bit number, as the 2^16 seconds is only about 18 hours. Also in a real system, the real time clock would be significantly more accurate, and not from trial-and-error on a single Nexys 4 board. The boards vary, and your RTC counter may run faster or slower than 1 Hz.

Extending I/O

If you choose to extend the I/O of your processor in Step 5, then you must implement additional functionality in data_mem.v and processor.v (to increase data width). Try and map the following memory locations by modifying the data_mem module inputs to the following memory locations.

Address	Name	R/W	Function
0x8000	Switches & LEDs	RW	Bits[15:0] Read from SW, the board switches Write to LED, the board LEDs
0x9004	Left Seven-segment Display	W	Write a 16-bit number to the seven-segment display, formatted as in the previous labs.
0x9006	Buttons	R	Read the value of the buttons BTN[4:0]
0x9010	Tri-Color LED Red	W	Write an 8-bit component Red color to the tri-color LED module.
0x9012	Tri-Color LED Green	W	Write an 8-bit component Green color to the tri-color LED module.
0x9014	Tri-Color LED Blue	W	Write an 8-bit component Blue color to the tri-color LED

			module.

Note on non-pipelined systems

While any number of things could be added to this memory map, including any of the remaining board I/O, the non-pipelined nature of this system actually makes this more complicated. In a traditional pipelined system, any external access (say, to the board RAM) is part of the cpu pipeline, and occurs before any write back to the register file.

With a non-pipelined system, the normal write-back to the register file occurs at the end of the single processing cycle; the entire instruction is processed before the next clock edge. This is problematic when the RAM, or any other block, registers its outputs; there is a cycle delay, and so if instruction 0x4 is a read from a RAM, it is the next cycle when instruction 0x6 is being processed that the RAM data is available. Additional hooks would need to be added to the cpu to allow register write-backs on subsequent cycles.