CPE 233 Lab Manual

OTTER Edition

Version 1.5

Tips and Suggestions

Using Vivado

Project Organization
Create a new project for each hardware assignment. Each project should be a single complete device. While projects often have multiple design source files, those design files should be connected to form a single device. Having design source files for unconnected or unrelated modules adds processing time too. (*Vivado is slow enough on its own, it doesn't need a poorly organized project to make it worse*)

Check and Clear Warnings
There are only a select few warnings that are inconsequential, like a warning for CFGBVS or a file that cannot be deleted/removed. Most warnings will prevent the digital device from working properly, even if it appears to work in simulation. Ignoring warnings with each module as you progress in this class only adds work later. This is called incurring technical debt because at some time in the future the "debt" must be paid by fixing the issues. Completing later hardware assignments on time requires the previous modules to be designed correctly or fixed. *One of the most common warnings that students ignore initially is for inferring latches. This is always a design flaw that must be fixed to get a working device, even if though it appears to work in simulation.*

Vivado Simulator
Using the simulator effectively and efficiently is essential for properly debugging hardware designs. This includes adding internal signals from a design to the sim, changing how multibit signals are displayed (changing the radix), and using the restart / run for time controls. The simulator is heavily used in 233 to verify components and then debug issues later in the quarter. You *will* spend a lot of time with it, so spending some time early on to become familiar with how to use the simulator will save a lot of time later.

SystemVerilog Code

Hardware Description Coding
SystemVerilog coding is not writing software or a program, it is used to describe a piece of hardware. SystemVerilog uses logical constructions similar to those used in software programming to describe how the hardware functions, but it is not software that "runs" on an FPGA. Vivado is a hardware synthesizer that takes the coding description in SystemVerilog and creates a hardware layout on the FPGA. How well you describe it to the synthesizer impacts how well it understands what you are trying to accomplish and create the hardware you meant. My SystemVerilog Primer is a review of key coding concepts and practices.

SystemVerilog Style Guide
The SystemVerilog style guide can help create clearer, more organized, designs. Learning to write code in an organized manner is a vital skill. Code is ~~rarely~~ *never* written correctly the first time, even for experienced coders, so the code should be structured and organized to allow easier reading and debugging. Spending time learning to follow the style guide will save time.

Hardware Assignments

Hardware Assignment Report Format

Behavior Description **(10)**

Describe the behavior of the designed component(s). This should be a short synopsis that explains in your own words the functionality of the component(s).

Structural Design **(5)**

Include an image from Vivado of the RTL elaborated design schematic. Be sure to expand any submodules to show the hardware each is composed of. Use this image to show that the design is built effectively. Efficient designs should not include long chains of gates due to timing and propagation delay. If the design shows any latches, it will need to be fixed or redone. For proper operation of the OTTER MCU, no latches can exist.

Latches are a major cause of problems when assembling the OTTER MCU from "working" components. The simulation may show the component functions properly with latches, but they will cause timing issues that may not appear until trying to combine the components into a fully functional OTTER MCU on the FPGA hardware.

Synthesis Warnings Listing **(5)**

Include a screenshot of the Messages tab in Vivado showing a listing of all Synthesis warnings and errors.

Verification **(50)**

Provide sufficient simulation evidence to show the component functions completely and correctly. For most components, it will not be possible to show every possible input combination and test case. Test cases should be chosen intelligently to show proper functionality. The verification process is fundamentally about trying to "break" the component or cause it to fail. The more extensive the attempt to "break" the component, the more probable it is to function properly. Large simulation sets can be broken into multiple images for readability. This section should not just be a set of simulation images. Explain what is in the simulation results including what test cases were simulated and why those select cases are sufficient to verify complete functionality. Include a table of what test cases were performed to match the simulation waveform.

Failing to properly and fully verify the functionality of each component individually is the other major cause of problems when assembling the OTTER MCU on the FPGA hardware. Do not skimp on this aspect of each component or be prepared for tedious and laborious debugging after building the completed OTTER MCU on the FPGA.

SystemVerilog Source Code **(30)**

Provide the SystemVerilog source code for the component(s). The source code must be readable with proper spacing and tabbing. Use good variable and signal names. The source code should also contain comments for understanding and readability. To make the code readable in the report, use a fixed-width font (Courier or Monospace), single line spacing, and avoid line wrapping (Suggest page margins of 0.2 to 0.5). Code also needs to be highlighted which can be done with an online tool like https://pinetools.com/syntax-highlighter (Select Verilog language and the Xcode style suggested). If using Google Docs, you can also use a tool like Code Blocks (Language Verilog, Theme Xcode suggested). *Never use screenshots of code.*

Hardware Assignment 1 - Program ROM and Assembly Programming

(and a dash of Reverse Engineering)

Learning Objectives

- To understand the makeup and organization of the program memory
- To understand the basics of assembly language programming.
- To understand the architecture required to support simple assembly language operations.
- To understand how an assembly language program is assembled and converted into a binary format (machine code) which the processor can execute.
- To understand how to use an assembly language simulator to analyze an assembly language program.

General Notes

This lab demonstrates the concept of reverse engineering as part of the presentation of the architecture of the OTTER processor and basic assembly language programming. Reverse engineering is a common strategy used to evaluate products, especially in the security and military sectors. This history of computers is filled with escapades of reverse engineering, one of the most famous being Compaq's reverse engineering of IBM's PC BIOS to create the rise of the IBM clones and the personal computer market that remains today.

Program Memory (ROM)

The Program ROM (ProgROM) is a memory device that contains the program that will be executed by a microcontroller (MCU). The MCU only reads from the ProgROM, hence it is treated as read-only memory (ROM). Typically the ProgROM is loaded with the compiled and assembled machine code from an external programmer. For the OTTER MCU, the ProgROM will be preloaded with the machine code when synthesized rather than being programmed by an external loader afterwards.

All memory devices share a common interface of address and data. The ProgROM is organized as 16,384 words (32-bits of data) where each word corresponds to a single instruction (PROG_IR). The memory in the OTTER is byte addressed rather than word addressed. Because each instruction (word) is comprised of 4 bytes, a new instruction is located at every 4 address locations. If the first instruction is at address 0x0000_0000, the next instruction will be at address 0x0000_0004.

The makeup of the ProgROM is divided between two files, the ProgROM.sv and otter_memory.mem. The ProgROM.sv is a SystemVeriog source file that creates a generic memory device of size 16384x32. Once created, this source file never needs to be changed because the ProgROM is always the same memory device. What changes in the ProgROM is the data that is stored in it. This data is separated into the otter_memory.mem file which is used to initialize the ProgROM. The code for the ProgROM.sv file is given below.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:   Cal Poly
// Engineer:  Paul Hummel
//
// Create Date: 01/04/2020 01:00:34 AM
// Module Name: ProgRom
// Target Devices: OTTER MCU on Basys3
// Description:  Generic 16384x32 ROM device
//
// Dependencies: prog_rom.mem file is a raw listing of 16384 32-bit hex values
//                prog_rom.mem file is automatically created by the OTTER
//                assembler / simulator from an assembly code program.
//
// Revision:
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////

module ProgRom(
      input PROG_CLK,
      input [31:0] PROG_ADDR,
      output logic [31:0] INSTRUCT
      );

      logic [13:0] wordAddr;

      // convert byte address to word address
      assign wordAddr = PROG_ADDR[15:2];

      (* rom_style="{distributed | block}" *)
      (* ram_decomp = "power" *) logic [31:0] rom [0:16383];

      // initialize the ROM with the otter_memory.mem file
      initial begin
         $readmemh("otter_memory.mem", rom, 0, 16383);
      end

      always_ff @(posedge PROG_CLK) begin
         INSTRUCT <= rom[wordAddr];
      end

endmodule
```

Code Segment 1.1: ProgROM.sv SystemVerilog Code Listing

otter_memory.mem

The otter_memory.mem file contains the machine code that should be loaded into the ProgROM. This file is a machine code listing of 32-bit instructions in hexadecimal format. This listing can be created by the OTTER assembler RARS after it successfully assembles the assembly code program. The contents will need to be

updated for every program that needs to be loaded into the OTTER MCU. The otter_memory.mem file can be added to a Vivado project by choosing to add a design source and then selecting the .mem file

Example of Reverse Engineering ProgROM

Consider the following OTTER assembly language program

```
main:   addi x5, x0, 0x05
        addi x6, x0, 0x64
        add  x6, x5, x6
        slli x5, x5, 0x02
        or   x6, x6, x5
        jal  x0, main
```

Code Segment 1.2: Example Program

After assembling the program in RARS, the following hexadecimal machine code listing can be generated by saving a memory dump of the code (.text) portion in hexadecimal text format. This saved file can then be used as the otter_memory.mem file in the ProgROM in Vivado.

```
0x0000_0000: 00500293
0x0000_0004: 06400313
0x0000_0008: 00628333
0x0000_000C: 00229293
0x0000_0010: 00536333
0x0000_0014: fedff06f
```

Hex Listing 1.1: Hex Machine Code of Example Program

The assembly instruction can be recreated from the 32-bit machine code by comparing the various function and opcode values of the instruction to the format outlined in the OTTER Assembler Manual. This is easier to visualize and match in binary form rather than hexadecimal. The results are shown in Table 1.1 below.

ProgROM Address	Machine Code in binary	Assembly Instruction
0000	0000 0000 0101 0000 0000 0010 1001 0011	addi x5, x0, 0x05
0004	0000 0110 0100 0000 0000 0011 0001 0011	addi x6, x0, 0x64
0008	0000 0000 0110 0010 1000 0011 0011 0011	add x6, x5, x6
000C	0000 0000 0010 0010 1001 0010 1001 0011	slli x5, x5, 0x02
0010	0000 0000 0101 0011 0110 0011 0011 0011	or x6, x6, x5
0014	1111 1110 1101 1111 1111 0000 0110 1111	jal x0, -20

Table 1.1: Assembly Program Construction from Machine Code

Assignment

Reverse engineer the excerpts from the prog_rom.mem file shown below to determine the assembly instructions implemented by this file. This can be done by first creating a table similar to Table 1 from the Example Program. Any other lines from this file can be assumed to be all zeros and disregarded.

```
0x0000_0000: 11000537
0x0000_0004: 00a00f13
0x0000_0008: 00051783
0x0000_000C: 41e7da33
0x0000_0010: 00fa4633
0x0000_0014: 04c52023
0x0000_0018: ff1ff06f
```

Hex Listing 1.2: otter_memory.mem Segment for Reverse Engineering

Specific Deliverables (*This assignment will not follow the normal Hardware Assignment Report Format)*

1. Completed disassembly table for constructing the reverse engineered otter_memory.mem segment similar to Table 1.1
2. Typed assembly code for the reverse engineered otter_memory.mem segment.
3. Simulation of the ProgROM.sv using the otter_memory.mem from the reverse engineered assembly program. Create clock and address signals and increment the input address from 0 until the entire program listing has been output.

Hardware Assignment 2 - Program Counter

Learning Objectives

- To understand how to modify a generic counter module to serve more specific purposes
- To understand how to use an n-bit register and MUX to construct a counter, which can implement the features typically required of a “program counter”.

The Big Picture

While there are many different computer architectures that include a variety of modules, one common module in most all architectures is the Program Counter (PC). The basic definition of a computer is a machine that executes instructions stored in memory to produce a result. For the OTTER microcontroller, those instructions are saved in the Program ROM (ProgROM) that was studied in Assignment 1. The PC is a register that saves the instruction (program) address for the ProgROM.

General Notes

A counter is a special type of register that performs operations associated with counting. Counters generally operate synchronously, changing outputs in sync with the edge of a clock signal. Designing counters in SystemVerilog is relatively simple as an *n*-bit register with an incrementer.

This assignment also includes extra digital circuitry for loading the PC from multiple external sources. Normally the PC operates by incrementing the current address, or count, to access the next instruction from memory (ProgROM). The PC must also be able to load values for executing instructions that are not next in sequence. This will be achieved with the use of a multiplexer (MUX).

Vector Addition

Incrementers can be designed without specifying the low-level implementation such as ripple-carry adders. SystemVerilog allows adders and incrementers to be designed behaviorally, allowing the synthesizer to optimize the implementation. This is not only simpler for the designer, but typically results in a better performing design. The code segment below gives an example of adding two 4-bit signals

```
logic [3:0] a, b, sum;

sum <= a + b;     // simple addition
```

Code Segment 2.1: Example SystemVerilog Addition

Notes on SystemVerilog Structure and Assignments

1. Wires, Regs, and Logic can be used globally throughout a body of SystemVerilog code. This means that a signal can be declared at the top of an architecture, assigned within an always block, and then used anywhere in the module. This includes outside of the always block in which the signal was assigned.

2. Be mindful of blocking (=) vs non-blocking (<=) assignments in always blocks. Combinational logic should use blocking (=) while sequential logic should use non-blocking (<=). When a signal is assigned within an always block using non-blocking (<=) assignment, the signal value is only assigned or updated when the always block is completed. This means that a signal cannot be assigned a value within an always block and then expected to utilize the updated value within the same always block sequence. Signals assigned with a blocking (=) assignment are assigned or updated immediately within the sequence of the always block, so these values can be utilized later in the same always block.

The implication of the above two rules for this project is the MUX module should use blocking assignment statements while the Program Counter register module should use non-blocking assignment statements. If making both components inside a single module, the signal connecting the MUX to the Program Counter should be an internal signal (wire, reg, or logic) and not an input or output.

Circuit Details

Figure 2.1 below shows the top-level black-box model of the Program Counter register. Table 2.1 provides an overview of the PC's signals and operation.

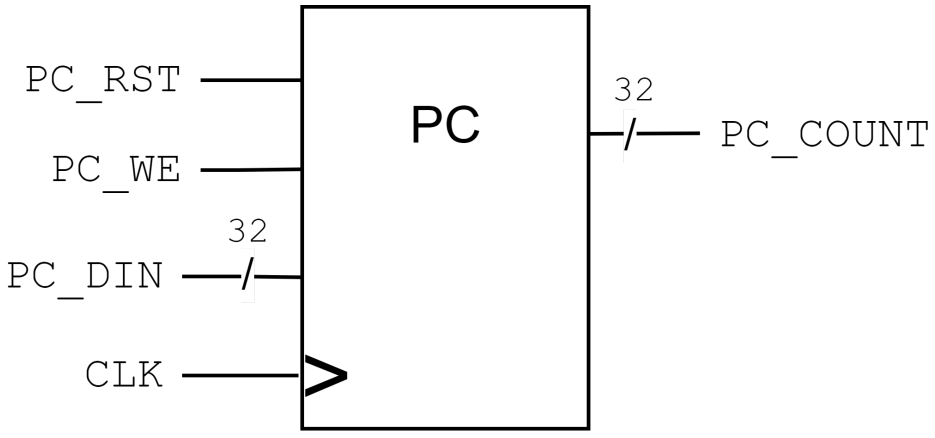


Figure 2.1: Program Counter Black Box Diagram

Signal	Comment
PC_COUNT	The current value in the PC. This value is used as an address for the ProgROM
PC_RST	Active high synchronous reset. When PC_RST = 1, the PC is reset to 0. This signal will have the highest priority.
PC_WE	Active high, enables synchronously loading the value from PC_DIN into the PC.
PC_DIN	Unsigned 32-bit value that is loaded into the PC when PC_WRITE is high
CLK	Synchronizes all PC operations. Same speed as the system clock of the OTTER MCU

Table 2.1: Overview of the Program Counter Signals

Figure 2.2 below shows how the PC is loaded from various sources with a MUX. The MUX allows the PC to change according to the current instruction being executed and the state of the OTTER MCU. The MUX inputs include:

1. PC+4: This is the most common input to the PC because it will increment to the next instruction. Remember that memory in the OTTER is byte-addressable and each instruction is a 32-bit word or 4 bytes.
2. JALR: This is the input that will cause the PC to jump to a new instruction from a `jalr` operation
3. BRANCH: This is the input that will cause the PC to jump to a new instruction from a branch (`beq`, `bne`, etc) operation
4. JAL: This is the input that will cause the PC to jump to a new instruction from a `jal` operation
5. MTEVC: This is the input that will cause the PC to jump to an interrupt service routine
6. MEPC: This is the input that will cause the PC to return from an interrupt service routine.

While some of these may seem similar and you may wonder while they would each need their own signal, the reason for the separate signals will become apparent when designing the digital hardware that creates each of these signals.

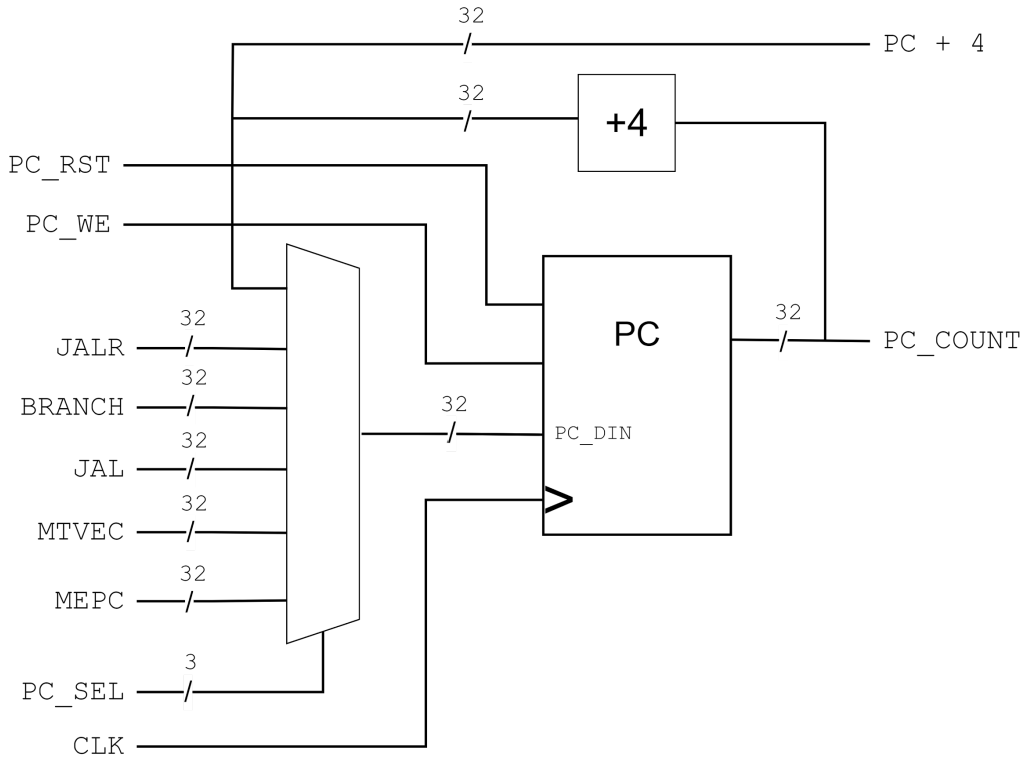
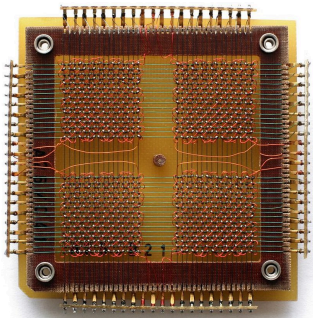


Figure 2.2: Program Counter with Input MUX and +4 Adder

Assignment

Design a PC register, input selection MUX, and +4 adder. This should be implemented in at least 2 design source files, the PC register and then a top level module that implements the MUX and +4 adder connecting to the PC register. The MUX can also be a separate design source that is connected in the top level similarly to the PC register making a total of 3 design source files in the project. The top level module connecting the MUX, PC register, and +4 adder will only be used in this assignment for testing purposes. When implementing the full OTTER in a future assignment (HW 6), the top level module will not be used. The +4 adder and MUX (if it isn't a separate module) can be re-implemented in the top level source connecting all the modules in the OTTER at that time.

Hardware Assignment 3 - Register File



Learning Objectives

- To implement the register file in the OTTER MCU architecture.
- To learn how a register file operates and how it will work when integrated with the OTTER CPU.

General Notes

A microcontroller (MCU) is generally comprised of three main components: memory, input/output, and computation. There are two basic memory types, ROM (read-only memory) and RAM (random access memory). ROM is used to store data that the MCU only needs to read when running. This is typically only used for memory that contains program instructions. The ProgROM studied before is an example of ROM. RAM memory is used for reading and writing by the MCU. Typically RAM is used to save temporary values that are loaded and operated on by the computation components.

Memory can be thought of as a table holding binary numbers. Each row can be identified by its location in the table by numbering each row starting with 0. This numbering is the address and how individual memory items can be accessed. The size of the memory unit is determined by how many rows (depth) and how many bits are saved in each row (width). The address values being binary, the number of addresses is defined as 2^N where N is the number of bits in the address. A 10-bit address can save 2^{10} = 1024 locations. Besides the address and data signals, memory components have control lines that direct operation. For building the OTTER MCU, the only control signal needed is a write enable (EN). The data entering the RAM module will only be saved when EN is active. All of the memory modules are synchronous and will only change on the rising edge of the clock signal.

OTTER Register File

The register file is a small but critical memory component in every microprocessor. Registers are the most basic memory unit that all computational operations use. The OTTER MCU has 32 registers (x0 - x31) with each register occupying a row in the register file “table”. Each register is able to store 32-bits, making the OTTER a 32-bit MCU and giving the register memory a size of 32x32. The register file is a dual-port RAM that allows reading from two locations or addresses simultaneously. The RAM memory can be read asynchronously. The register file needs two address inputs (ADR1 and ADR2) to access two registers simultaneously. The outputs, the values saved in the two registers, are sent to RS1 and RS2. ADR1 and ADR2 will have values from 0 to 31, corresponding to the register being accessed, and RS1 and RS2 will be the 32-bit values saved in each of those respective registers.

Only a single location, or register, can be saved per clock cycle. The address where data is saved is specified with the w_adr (write address) input. w_data (write data) is the 32-bit data value to be saved in the location (register) specified by w_adr. While data is read from the RAM asynchronously, data is saved in the RAM

memory synchronously on the rising edge of the clock. Reading or outputting data to RS1 and RS2 happens continuously. Saving data is controlled by the write enable signal EN. The data (from w_data) will only be saved if write enable (EN) is high on the rising clock edge.

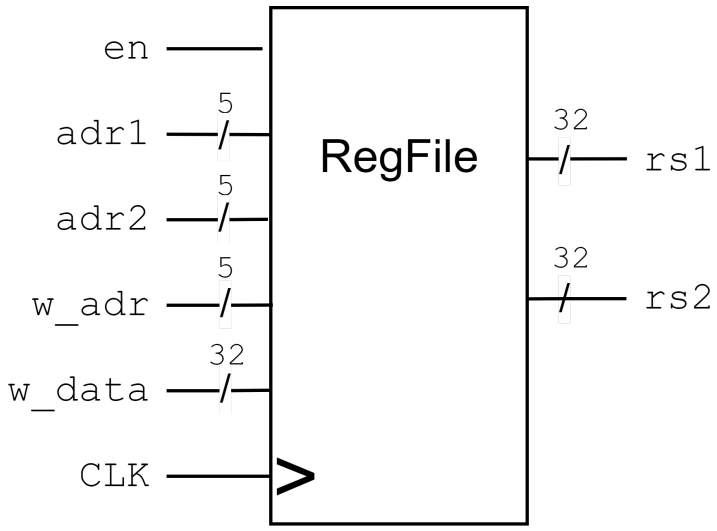


Figure 3.1: Register File Black Box Diagram

Zero Register (x0)

The RISC-V instruction set has a unique register x0 that must be implemented in the OTTER Register File. The zero register (x0) is hard wired to 0s so that it will always output a value of 0. Saving to x0 is a valid input that the Register File cannot prevent from occurring. Instead, it must be designed so that any attempts to save a different value to x0 will not change it from outputting 0s when reading x0. This can be accomplished in 2 ways. (Only 1 method should be used)

1. Initialize x0 to 0s and do not save any value over it. Add logic to prevent overwriting address 0 in memory.
2. Add logic to always output 0s when reading address 0. So new values can be saved into memory at address 0, but that value will not be output when reading from address 0.

Memory in SystemVerilog with Arrays

Memory devices are easily defined in SystemVerilog with arrays. Arrays are an indexed group of signals (vectors) of uniform size and type. Typically arrays are created as indexed groups of signals (vectors), although an array of single bit signals is possible. The data width or size of the signal (vector) is defined after the type, wire, reg, or logic. The size of the array or the number of signals grouped together is defined after the signal name. Code Segment 1 below shows an example of creating a 512x16 memory unit (512 signals indexed 0 to 511 that are each 16-bits, MSB first).

```
// Create a memory module with 16-bit width and 512 addresses
logic [15:0] ram [0:511];

// Initialize the memory to be all 0s
initial begin
  int i;
  for (i=0; i<512; i=i+1) begin
    ram[i] = 0;
  end
end

// Access the ram memory by address
ram[address]    // ram[100] = data stored in address 100
```

Code Segment 3.1: SystemVerilog for a 512 x 16 Memory Device

Assignment

Design the OTTER register memory module shown in Figure 1 and verify it functions correctly. Ensure the module can save and output 32-bit values as well as verifying the correct behavior for the zero register (x0).

Hardware Assignment 4 - Arithmetic Logic Unit and Immediate Generator

Learning Objectives

- To understand the requirements of the Arithmetic Logic Unit (ALU) based on the OTTER instructions it must support.
- To understand the design and operation of the ALU.
- To understand how immediate values are generated from the various instruction types

General Notes

The arithmetic logic unit (ALU) shown in Figure 4.1 is a central part of any microprocessor. The ALU is responsible for performing all of the arithmetic and logic operations including any bit-crunching required by all of the OTTER instructions. While the ALU is a relatively complex device, behavioral coding in SystemVerilog is able to simplify the design and use the synthesizer to handle the complexity.

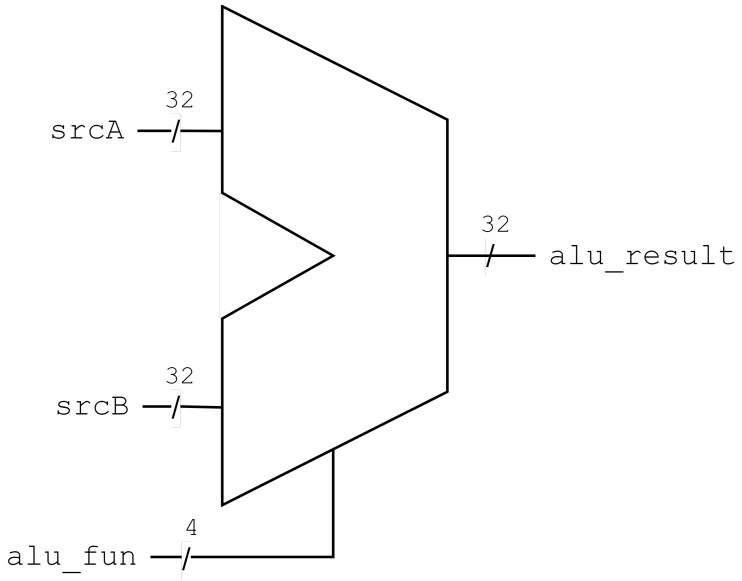


Figure 4.1: ALU Diagram

ALU

The ALU is able to perform a variety of operations on two inputs (A and B). The ALU in every microprocessor can be different to perform different operations. Specific microprocessors can be optimized for certain tasks by designing an ALU to process those functions while eliminating unneeded operations. This ALU_FUN input

will control which operation the ALU performs on the data inputs. The size of the ALU_FUN input will be determined by how many different operations are needed. The ALU in the OTTER MCU is capable of 11 operations listed in Table 4.1 below. The OTTER Assembler Manual provides an expanded explanation of each of the associated OTTER instructions. The ALU is a unique component in the OTTER MCU as one of the few modules that are completely combinational, it has no flip-flops or clock signals. This makes the ALU critical for timing requirements in the OTTER MCU. *A poorly designed ALU can cause a variety of issues that manifest in the hardware implementation but are impossible to diagnose in a behavioral simulation. Look at the elaborated design of the ALU is essential for spotting poor implementations.*

ALU_FUN	Instruction
0000	ADD
1000	SUB
0110	OR
0111	AND
0100	XOR
0101	SRL

ALU_FUN	Instruction
0001	SLL
1101	SRA
0010	SLT
0011	SLTU
1001	LUI-COPY

Table 4.1: ALU Function Table

Immediate Generator

The Immediate Generator is used to create 32-bit signed extended values from the assembly instruction immediate components. Each of the 5 types of instruction uses different bits of the assembly instruction to define the immediate value. Of the 5 types, 3 of these are routed to the ALU. The remaining 2 go to the Branch Address Generator which will be designed in a future assignment. Each immediate format is shown in Figure 4.3 below.

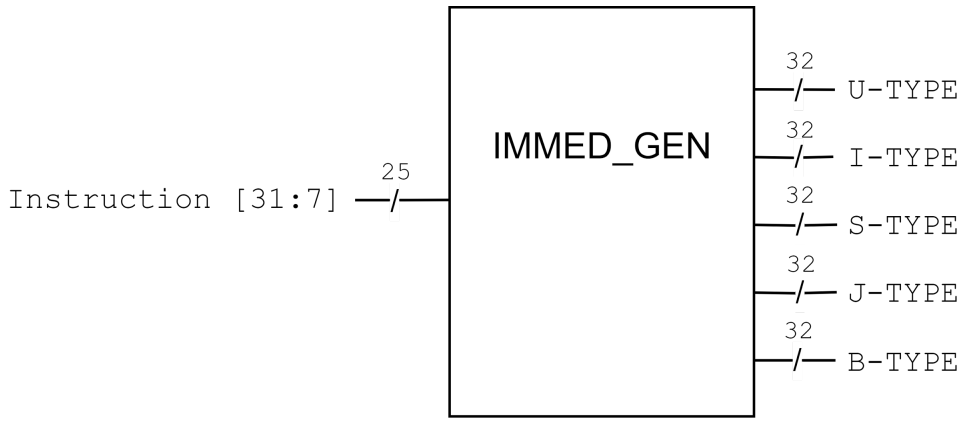


Figure 4.2: Immediate Generator Black Box Diagram

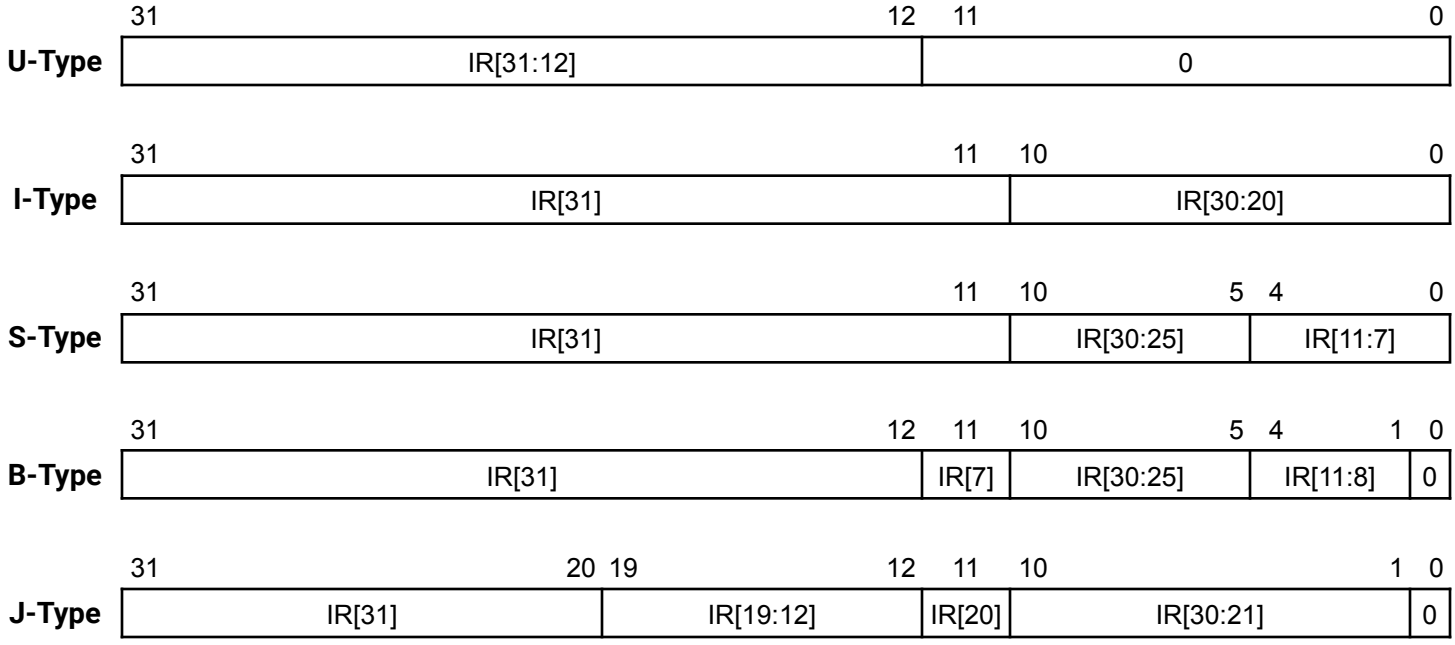


Figure 4.3: Immediate Generation Format

Assignment

Design and implement an ALU that performs the 11 OTTER arithmetic and logic functions as listed in Table 4.1 above. Design and implement an Immediate Generator that creates the sign-extended values for all 5 types of instructions.

Verification

Because of the critical nature of the ALU, a more robust verification process is warranted. Complete the test cases in Table 3 on the next page and include it in the OTTER assignment report. Note that the ALU always has a "Result" for all operations, as it is simply an output. Implement these test cases in a testbench in order to verify the proper operation of the implemented ALU. *Be sure to check the assembler manual and perform the operation by hand **before** checking the simulation.* Verify the results in the simulation match the values from Table 4.2. This means that all of the operations in Table 4.2 must be included in the testbench. For easier comparison ensure the test conditions in the simulation are in the same order they appear in Table 4.2.

Hints

- A properly designed ALU should contain 1 large MUX with a stack of logic or arithmetic gates that operate in parallel. Avoid long chains of gates or muxes in sequence.
- Ensure there are no warnings for latches
- Review bit concatenation and replication for the immediate generator.
- Use RARS to verify ALU operation results
- Use RARS to create instruction inputs for all of the immediate types

Function	ALU_FUN	Arguments		Expected Result (hex)
		A	B	
ADD		0xA50F96C3	0x5AF0693C	
		0x84105F21	0x7B105FDE	
		0xFFFFFFFF	0x00000001	
SUB		0x00000000	0x00000001	
		0xAA806355	0x550162AA	
		0x550162AA	0xAA806355	
AND		0xA55A00FF	0x5A5AFFFF	
		0xC3C3F966	0xFF669F5A	
OR		0x9A9AC300	0x65A3CC0F	
		0xC3C3F966	0xFF669F5A	
XOR		0xAA5500FF	0x5AA50FF0	
		0xA5A56C6C	0xFF00C6FF	
SRL		0x805A6CF3	0x00000010	
		0x705A6CF3	0x00000005	
		0x805A6CF3	0x00000000	
		0x805A6CF3	0x00000100	
SLL		0x805A6CF3	0x00000010	
		0x805A6CF3	0x00000005	
		0x805A6CF3	0x00000100	
SRA		0x805A6CF3	0x00000010	
		0x705A6CF3	0x00000005	
		0x805A6CF3	0x00000000	
		0x805A6CF3	0x00000100	
SLT		0x7FFFFFFF	0x80000000	
		0x80000000	0x00000001	
		0x00000000	0x00000000	
		0x55555555	0x55555555	
SLTU		0x7FFFFFFF	0x80000000	
		0x80000000	0x00000001	
		0x00000000	0x00000000	
		0x55AA55AA	0x55AA55AA	
LUI COPY		0x01234567	0x76543210	
		0xFEDCBA98	0x89ABCDEF	

Table 4.2: ALU Test Cases for Verification

Hardware Assignment 5 - Real Memory and Branch Generator

Learning Objectives

- To understand the basic functionality and implementation of the memory module
- To understand how memory can be organized and described with a memory map
- To understand the design and operation of the Branch Condition Generator and Branch Address Generator

General Notes

There are two main objectives for this assignment. The first is to understand the memory makeup and behavior in the OTTER. The Memory module is provided, but it must be thoroughly understood to utilize properly and effectively. The second is to design and implement hardware for branch operations. The branch hardware is divided into two components, a condition generator and an address generator.

Branch Condition Generator

The branch condition generator creates 3 signals that are the result of 3 comparisons on 2 input values. The 3 comparisons are Equal, Less Than (assuming inputs are unsigned), and Less Than (assuming inputs are signed). The results of those 3 comparisons are sufficient to determine every branch condition instruction in the OTTER.

Branch Address Generator

The branch address generator creates 3 signals that conditionally change the program counter (PC). The branch output is used for all branch instructions (B-Type). It is formed by adding the current PC to the B-Type immediate. The jal output is only used for the jal instruction which is the only J-Type instruction. It is formed by adding the current PC to the J-Type immediate. The jalr output is only used for jalr instructions. It is formed by adding the output of rs1 to the I-Type immediate.

OTTER Memory Module

The memory module is available on Canvas. This memory module will supersede the previously used ProgROM. This memory module can be used for reading instructions from memory, reading and writing data to and from memory, and reading and writing data to and from external IO. All reading and writing operations are synchronous. This is a requirement for utilizing a feature of the FPGA called block RAM (BRAM).

OTTER Memory Map

The OTTER uses a Von Neumann architecture because instructions and data both use the same address space. As such, organizing the regions for program code, data, and stack are completely up to the programmer. This can be overwhelming to inexperienced programmers, so the framework shown in Figure 5.1 will be used for programming and organizing memory on the OTTER. This organization provides a balance that should be adequate for any program or data storage used in this class.

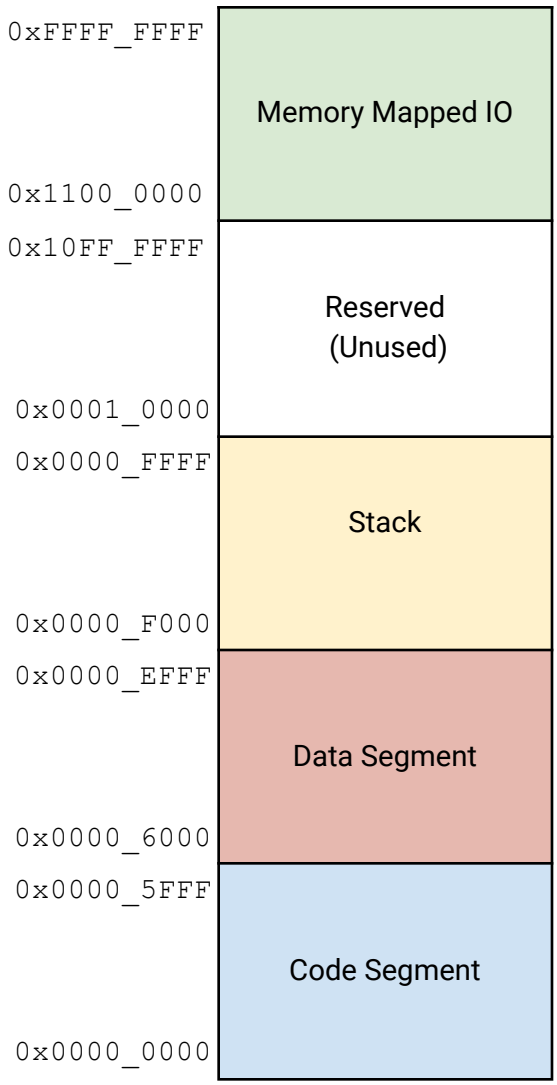


Figure 5.1: OTTER Memory Map

Signal Name	Size (Bits)	Purpose
MEM_CLK		
MEM_RDEN1		
MEM_RDEN2		
MEM_WE2		
MEM_ADDR1		
MEM_ADDR2		
MEM_DIN2		
MEM_SIZE		
MEM_SIGN		
IO_IN		
IO_WR		
MEM_DOUT1		
MEM_DOUT2		

Table 5.1: Memory Module Signal Table

Assignment

1. Fill out the signal table for the OTTER Memory module (Table 5.1 above)
2. Design and implement the Branch Condition Generator and Branch Address Generator.

Deliverables

Hardware report including the Branch Condition Generator and Branch Address Generator. Include the completed Table 5.1 above in the behavioral description section.

Hardware Assignment 6 - Control Unit

It's Alive

Learning Objectives

- To understand the basic functionality and implementation of an MCU control unit
- To understand the basics of fetch/execute instruction cycles
- To understand how to set control signals for a given instruction
- To learn how to debug and test a complex digital system

General Notes

The overall purpose of this assignment is to build the central component responsible for making a functional microcontroller (MCU), and then assembling the various OTTER modules from the previous assignments into a working MCU. This MCU will not be the complete OTTER MCU, as there are several important modules left out in order to reduce the scope of this experiment. Those modules will be added over the next couple of assignments.

While previous assignments involved assembling individual OTTER modules, this assignment requires a systems-level approach to connect several core modules into a functional unit. Understanding the OTTER at a systems level, including the interface to actual hardware, will help provide a complete understanding of the internal workings of the OTTER and computers in general. The signals in an MCU are typically divided into two groups, called the data path and control path. The data path is used to describe the path that data moves through the MCU. The data path generally consists of the signals carrying data from the program, through the registers, ALU, and to or from memory. The control path involves all of the signals that control how the data moves through the data path. The control path typically consists of select signals for muxes, function selections for the ALU, and read or write enable signals. The data path usually involves data that moves to, from, or through multiple modules. The control path typically originates from a single module, the control unit, and branches out to the other modules individually. Control signals do not typically pass through other modules.

Control Unit

The control unit acts as the brain of the OTTER MCU and is responsible for setting all of the control signals. It can be divided into two components to simplify the design and implementation process. Some control signal changes are timing critical. These control when data is read from or written to memory. Other control signals are less time dependent on when they are set or changed as long as they propagate quickly enough. The signals that require precise timing will be controlled by a finite state machine (FSM). The other signals will be controlled with a combinational instruction decoder. The architecture diagram will show which control signals are set by the FSM and which are set by the decoder.

Control Unit FSM

The control unit FSM will have 3 states, FETCH, EXEC, and WRITEBACK. The MCU will start in the INIT state on a reset to reset the PC. The MCU will then go to the FETCH state and will fetch the first instruction of the program. The MCU will then go to the EXEC state and execute the instruction. For most instructions, the MCU will then return to the FETCH state to fetch the next instruction. The exception to this behavior is loading instructions (lw, lh, lhu, lb, lbu). These require an extra cycle due to the memory module being synchronous read. The EXEC state will set all of the control signals to read from the memory, but the data will not be output until the next clock cycle. In order for this data to be properly saved in the registers, an extra cycle is needed. The FETCH state sets the memory unit to read an instruction. The EXEC state will vary depending on the instruction being executed. The WRITEBACK state sets the register file to save data and change the program counter.

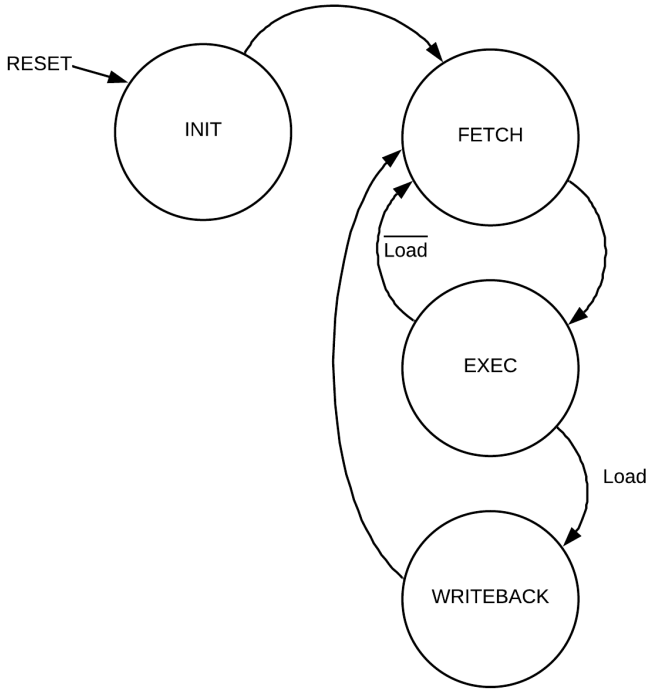


Figure 6.1: FSM State Diagram

OTTER MCU

The OTTER MCU will be implemented by connecting all of the existing modules as subcomponents. The completed MCU will consist of the entire OTTER architecture diagram, except for the CSR module. Figure 6.2 shows the black box diagram for the OTTER MCU module. This module will only include port maps of instances of modules already created and muxes for connecting signals.

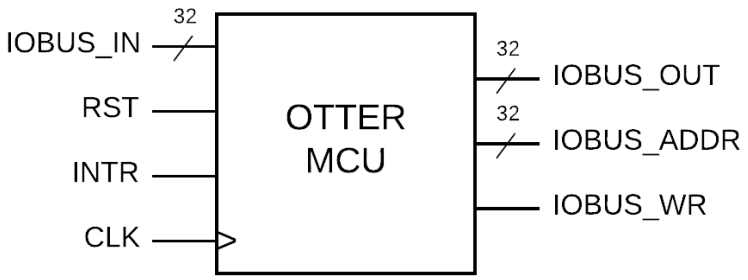


Figure 6.2: OTTER MCU Black Box Diagram

Program

Simulate the program shown in Code Segment 6.1 below with the RARS simulator until the program's functionality is understood. Use RARS to save a hexadecimal machine code dump of the assembled program to create an otter_memory.mem file for initializing the OTTER MCU.

```
main:  lui   x5,  0xAA055
       addi  x8,  x5, 0x765
       slli  x10, x8, 3
       slt   x12, x5, x8
       xor   x13, x8, x10
       beq   x0,  x0, main
```

Code Segment 6.1: Sample Program for OTTER MCU Demonstration

Assignment

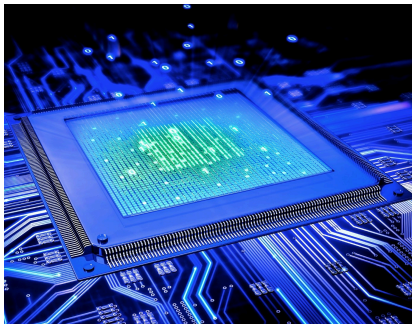
1. Implement the control unit decoder and FSM for at least all instructions used in Code Segment 6.1.
2. Implement the OTTER MCU by connecting the Program Counter, Memory, Register File, ALU, Immediate Generator, Branch Condition Generator, Branch Address Generator, and Control Unit as shown in the OTTER architecture diagram. Unused signals can be connected to 0 (CSR)
3. Verify the implemented OTTER MCU by simulating the MCU with the loaded program from Code Segment 1 above. Be sure to test the RESET input. The INTR and IOBUS_IN inputs can be kept at 0 because the hardware has not been implemented to support that functionality yet. Show any pertinent internal signals in the MCU for verification purposes (PC, PS, ALU Result, x5, x8, x10, x12, x13).

Deliverables

1. Behavioral descriptions of
    a. Control Unit Decoder
    b. Control Unit FSM
    c. OTTER MCU

2. Structural design of OTTER MCU showing internal modules as boxes
3. Synthesis warnings for OTTER MCU
4. Verification
    a. Simulation of OTTER MCU running Code Segment 6.1. (Be sure to show PC, PS, ALU Result, Registers 5, 8, 10, 12, and 13)
5. SystemVerilog code
    a. Control Unit Decoder
    b. Control Unit FSM
    c. OTTER MCU

Hardware Assignment 7 - OTTER Wrapper
Fully Functional



Learning Objectives

- To understand the more functionality and implementation of the MCU control unit
- To understand how to implement the OTTER Wrapper on the FPGA board
- To understand how to add peripherals to the OTTER Wrapper
- To learn how to debug and test a complex digital system

General Notes

The overall purpose of this assignment is to expand the functionality of the OTTER MCU and add a OTTER_WRAPPER to build a complete system that connects to the Basys3 development board. The wrapper will add the digital circuitry to connect external IO (switches, LEDs, buttons, etc) to the OTTER MCU. This will allow you to run a program on your fully functional OTTER that can utilize buttons, switches, and LEDs. To verify all of your hardware is working properly, you will run an assembly program written to specifically test every instruction. This assembly program is provided on Canvas as assembly code and a machine code mem file that you can add directly to your OTTER project.

The OTTER Wrapper

The OTTER_WRAPPER will connect the OTTER_MCU with peripheral modules that interact with the Basys3 development board and include any necessary interface functionality. Input and output MUXs will connect the IOBUS_IN and IOBUS_OUT to a variety of inputs and outputs. Every output will also have a register to keep the output signals constant until specifically changed.

Study the provided OTTER_Wrapper.sv file to understand how the MCU will interface to the Basys3 board. ***Note that the SWITCHES and LED memory location constants in the OTTER_Wrapper must match those used in the assembly code.*** Add the OTTER_Wrapper.sv file as the top module to your OTTER MCU project. Review any component names used to ensure the OTTER MCU previously built is correctly instantiated in the OTTER_WRAPPER. Write the necessary constraints file for connecting the OTTER_Wrapper to the Basys3.

Seven Segment Display:

The Test_All program will utilize the 7 segment display so a driver must be added to the OTTER WRAPPER. A 7 segment display driver (sseg dec) is provided on Canvas. This driver module will be connected to the output MUX similarly to the LEDs. Other output drivers (ADC, speaker drivers, etc) can be connected in a similar fashion to expand the functionality of the OTTER for the final project. Input drivers (keypad, keyboard, mouse, etc) would be connected to the input MUX in the same way.

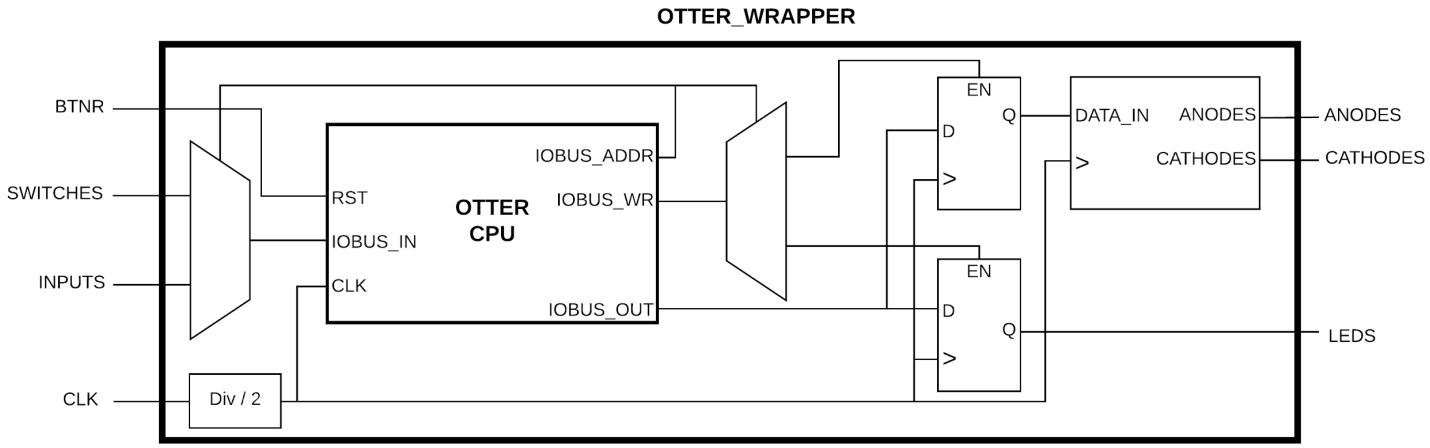


Figure7. 1: OTTER_WRAPPER Block Diagram

Test_All:

The Test_All program is an assembly program specifically written to test and verify the hardware correctly supports the RISC-V instruction set. The program will continually cycle through 37 series of tests with each series designed to thoroughly test a single instruction. Some series perform more tests than others depending upon which instruction is being verified. The program will output the current series of tests being performed on the 7 segment display in hex (0 - 24). Each series runs a sequence of tests, with each test lighting up a single LED for completion. Table 7.1 below specifies the number of tests in each series. When more than 16 tests are performed, only 16 LEDS will light up, so tests beyond 16 will have no visible indicator when completing. When a test completes but fails, the 7 segment will display -1 or 0xFFFF and the program will move onto the next series for testing.

Test_All also makes use of the switches as a method to control delays and pauses. After every series, the program will check the switches. If switch 0 is on, the program will pause, waiting for the switch to be turned off before continuing. This can ensure the results of each series of tests can be seen before moving onto the next series. Switch 1 is checked every time a test fails. If switch 1 is on, the program will pause, waiting for the switch to be turned off before continuing. When the program continues, it skips the rest of the current series and moves onto the next series.

When running on the Basys3 board, delays are needed to slow the program down to prevent the LEDs from changing faster than can be detected. However, delays add unnecessary and unwanted instructions in the simulation. To meet both needs, the program reads from the switches after every test. If switch 2 is on, the program adds a delay. Running on the board requires switch 2 to be on, but the simulation should always have switch 2 as off.

The Test_All program is provided as a mem file to use in Vivado for initializing the memory module and a debug listing showing the assembly instructions. The debug listing includes program addresses and labels to allow quick lookup and comparison when debugging.

Series	Tests	Series	Tests	Series	Test	Series	Tests	Series	Tests
0	17	8	16	10	16	18	10	20	10
1	16	9	16	11	16	19	7	21	10
2	7	A	16	12	16	1A	10	22	10
3	7	B	5	13	9	1B	2	23	10
4	7	C	5	14	9	1C	5	24	10
5	19	D	5	15	7	1D	2		
6	17	E	15	16	7	1E	4		
7	20	F	16	17	7	1F	10		

Table 7.1: Table of Series and Tests in Test_All

Assignment

It is impossible to overstate the importance of ensuring that the OTTER MCU is working properly on hardware in this assignment. When there is a failure or a series of tests fails to complete, it will be necessary to delve into the program to determine what specific instruction caused the hardware to fail the test. This must all be done in Vivado's Simulator. The Vivado Simulator will be critical for properly debugging the design.

1. Complete implementation of the control unit decoder and FSM for all instructions except CSRRx and MRET.
2. Verify the OTTER is working for all of the implemented instructions by running Test_All on your OTTER. Because it is highly likely (*assuredly*) that there are bugs in your OTTER implementation, it is best to start by running Test_All in Vivado's simulator. Only after you see that it is working as intended in the simulator should you test it on the Basys3. If any part of the program fails to work on the Basys3, go back to the simulation and trace all of the internal signals to find the error.
3. Demonstrate all of the working test cases to the instructor or TA. This demonstration will replace and account for the verification section of the report. No simulation images or tables will be needed in the report because each group would have different simulations depending upon which issues or aspects of the OTTER needed fixing.

Debugging Tips

- Add the program counter to the simulation as a reference to track where the simulation is in relation to the assembly program
- Add the state variable (PS) from the Control Unit to reference when the MCU is Executing vs Fetching

- Look at the IOBUS_WR signal because it will go high every time the LEDs or 7 Segment is changed. You can select this signal in the simulator and then use arrow keys to move to the next transition to quickly move through the simulation and verify the test cases are passing. The 7 Segment should count at the beginning of each set of tests and the LEDs should light up 1 at a time (1, 3, 7, 15, 31, 63, …) as it goes through the test cases.
- Search for the PC address of the <fail> subroutine. If the PC ever gets to this address, it means an error occurred. Back trace from this to see where the program jumped to this address and then diagnose the error that caused it.
- Ensure the entire program has run properly in the simulator by following the PC. It should start at the beginning of the main loop, go through each set of tests and then jump back to the beginning of the loop

Deliverables

1. Behavioral Descriptions - OTTER Wrapper
2. Structural Design of OTTER Wrapper showing components (Do **not** expand any of the modules)
3. Synthesis Warnings for OTTER Wrapper
4. SystemVerilog Code
    a. Control Unit FSM
    b. Control Unit Decoder
5. *No verification section is needed in the report, but you must demonstrate your working TestAll to your instructor or a lab assistant.*

Hardware Assignment 8 - Interrupts and Button Bounce

Pardon the interruption

Learning Objectives

- To understand microcontroller interrupts, including the linkage between MCU hardware and the OTTER MCU instruction set architecture
- To implement the interrupt architecture on the OTTER MCU hardware
- To understand how to write and implement interrupt service routines (ISRs) on the OTTER architecture
- To learn about switch bounce and debouncing requirements for external input devices

General Notes

Microcontrollers perform operations in programmed and defined sequences that occur at deterministic times. Not all events that a microcontroller may need to react to occur at defined or known times. Most input interactions with a user cannot be programmatically defined for when they happen. Button presses or keyboard inputs can be read by the microcontroller, but the data read by the microcontroller is only important when a button or key has actually been pressed. If the microcontroller program reads input from a set of buttons when an instruction is executed, the likelihood that the instruction to read the buttons occurs in the same few milliseconds of time that the user presses the button is low.

One technique to circumvent this issue is to repeatedly read the buttons until a button press is detected. This is a process called polling because the input is continually polled until some input is detected. While this method can function in some situations, it has several limitations. This technique does not scale well with more inputs that require polling. If polling an increasing amount of inputs in sequence, there becomes a greater chance that an input event is missed because it occurs between being polled. The more significant drawback to polling inputs is that the microprocessor cannot perform any other operations and is instead wasting power cycles effectively doing nothing. When building embedded devices, efficiency, especially with respect to power, is critical. Many embedded devices have limited power sources, often needing to stretch battery life as long as possible.

A microcontroller could be more efficient if it was possible to perform other operations or power down until an input event occurs that it needs to handle, and when the event like a button press is detected, the microcontroller could pause what it is currently doing to go read the button inputs. This would ensure no input is missed while not wasting any cycles continuously polling. Modern microcontrollers have the ability to do this with a hardware mechanism called an interrupt.

Interrupts

Interrupts do exactly what they sound like, they interrupt the microcontroller from what it is doing to signal it to do something else. This is done by creating a specific subroutine that is executed when an interrupt occurs. This subroutine is called an interrupt handler or interrupt service routine (ISR). When an interrupt occurs, this

subroutine is executed. When the subroutine is completed, the microcontroller returns to wherever it was in the program when the interrupt occurred and continues where it left off. Typically interrupts are for events that occur at random or arbitrary times. Interrupts can also be used to signal important events that require immediate attention. Consider events like the bumper sensor in a car that triggers the airbags. The airbags should never be delayed because the bumper sensor is waiting to be polled or because the car microcontroller is busy updating the dash with the current engine rpm. Most modern microcontrollers have complex interrupt handling hardware that is able to handle over 50 unique interrupts, each with its own ISR.

Many embedded systems are designed to do everything with interrupts. The main program sets up all of the peripherals and interrupts before entering a low power mode doing nothing. When an event occurs, the interrupt causes the microcontroller to wake up, process the event, and power back down to wait again. Because interrupts can occur at any time it is important to handle the event quickly so that a second interrupt event doesn't get missed while processing the first one. The golden rule to writing an ISR is to get in and get out as quickly as possible. An ISR should never have polling loops or software delays. The ISR should be written to handle a single specific task efficiently. Any time consuming calculations or data processing can be triggered to run in the main program after the ISR is complete rather than extending the time spent in the ISR.

OTTER MCU Interrupt

Rather than implementing a complex interrupt handler, the OTTER MCU can only handle a single external interrupt. The mechanism used to handle interrupts in the OTTER MCU is referred to as a vectored interrupt. Once the OTTER MCU receives an interrupt, the MCU completes the execution of the current instruction and then jumps to a predetermined address in instruction memory. This address is known as a vector address. This address will be where the ISR code begins. The code in the ISR generally implements some task to appropriately handle the interrupt. When the MCU finishes executing the ISR, the MCU returns to where it was in the program when the interrupt was detected.

When the OTTER MCU receives an interrupt, part of the automatic response of the hardware is to prevent the MCU from acknowledging any further interrupts until the current interrupt is processed (the ISR is finished). Preventing the OTTER MCU from processing later occurring interrupts is referred to as interrupt masking. Remember, preventing the OTTER MCU from acting on interrupts does not prevent the interrupt causing event from happening, and instead simply causes the OTTER MCU to ignore any occurrence of an interrupt. Interrupts can also be manually masked or enabled during any program segment.

Control Status Registers (CSR)

The control status registers are a set of special purpose registers for specific extra functionality. While the RISC-V instruction set allows up to 4096 registers in the CSR, the OTTER will only use 3 of them listed below in Table 8.1. The CSR also contains the necessary logic to save the program counter and disable interrupts when an interrupt occurs.

Name	CSR Address	Description
MSTATUS	0x300	Machine Interrupt Status Register - Bit 3 (MIE) is used for enabling / disabling interrupts and bit 7 (MPIE) is used to copy and restore MIE during the ISR to avoid the ISR from being interrupted.
MTVEC	0x305	Machine Trap Vector - The value that is loaded into the program counter when an interrupt occurs
MEPC	0x341	Machine Exception Program Counter - Saves the current program counter when an interrupt occurs to be restored back after the interrupt subroutine is completed

Table 8.1: Table of Series and Tests in Test_All

Low-Level Details of Interrupt Handling

The OTTER MCU handles the low-level mechanics of interrupt processing in a manner similar to the handling of CALL and RET with subroutines. Assuming that interrupts are currently enabled, when the OTTER MCU detects that the signal attached to the interrupt input has been asserted, execution of the current instruction is completed before processing of the interrupt begins. After executing the current instruction, the OTTER MCU will go to the interrupt state and do 3 things.

1. Copy the vector address (MTVEC) to the program counter.
2. CSR saves the current address of the Program Counter to the MEPC so that execution can continue from where the interrupt occurred after the interrupt service routine is done.
3. CSR copies the MIE bit of MSTATUS (bit 3) to the MPIE bit of MSTATUS (bit 7) and clears the MIE bit.

Processing the ISR begins and continues until the OTTER MCU encounters an MRET instruction. Execution of this instruction causes the OTTER MCU to restore the address saved in MEPC to the Program Counter. This should be the address that was saved when the interrupt was detected which was the address of the next instruction that would have been executed had there not been an interrupt. MRET will also restore the MIE bit of MSTATUS (bit 3) by copying the MPIE bit (bit 7) and then clearing the MPIE bit.

General Sequence of a OTTER MCU Interrupt

1. The OTTER MCU detects an asserted signal on the interrupt input (assume the OTTER MCU has not masked the interrupt)
2. Execution of the current instruction completes and the OTTER MCU goes into an interrupt state.
3. In the Interrupt state
    a. Copy MIE bit of MSTATUS (bit 3) to the MPIE bit of MSTATUS (bit 7)
    b. Clear the MIE bit of MSTATUS to prevent interrupts during the ISR
    c. The current program counter is stored in the MEPC register in the CSR
    d. The program counter is loaded with the value in the MTVEC register of the CSR
4. The control unit FSM goes to the fetch state, fetching the first instruction of the ISR
5. The ISR executes

6. Execution of the ISR completes with a MRET instruction which does the following:
    a. Restore the MIE bit of MSTATUS (bit 3) by copying MPIE bit of MSTATUS (bit 7)
    b. Clear the MPIE bit of MSTATUS
    c. Load the program counter with the MEPC register of the CSR. This would have been the next instruction to fetch when the interrupt occurred.
7. Execution resumes by fetching the instruction that was next when the interrupt occurred.

Modifying the OTTER Control Unit for Interrupts

Interrupts on the OTTER MCU are primarily handled by the control unit, both the FSM and decoder. To implement interrupts the control unit FSM must be modified to contain an extra state that handles the OTTER's interrupt mechanism. The proper vernacular for this modification is to include an "interrupt cycle" into the current FSM. Figure 8.1 shows the state diagram for the Control Unit including the interrupt cycle. Note that entry into the interrupt state occurs when the OTTER's INTR input is asserted in the EXECUTE or WRITEBACK states. In the INTERRUPT state, the control unit asserts the signals to implement steps 3.a-c listed above.

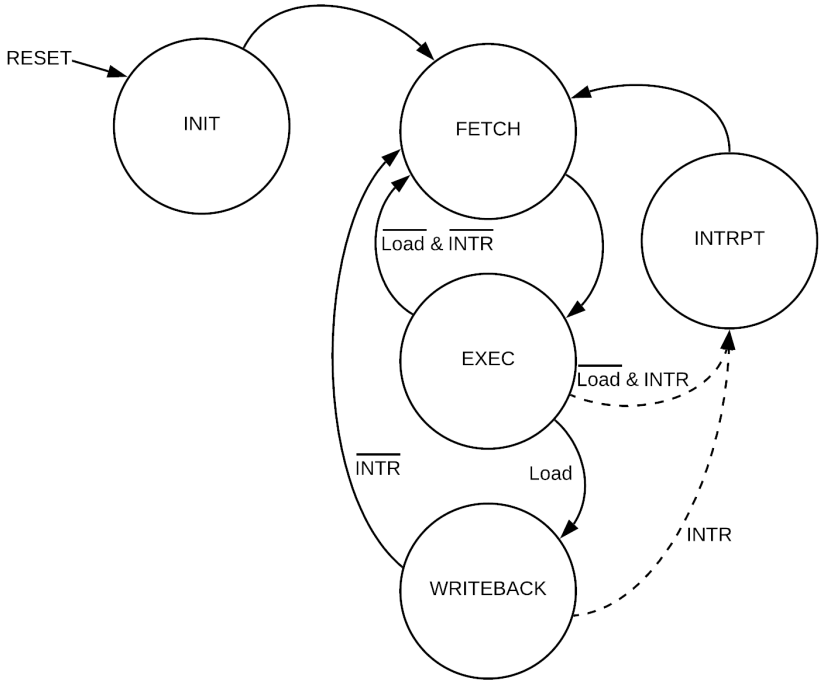


Figure 8.1: The OTTER Control Unit State Diagram with Interrupts

Interrupt Signals

When the signal on the interrupt input is asserted and the interrupt is not masked, the interrupt process listed above occurs. The interrupt signal must be a pulse of a particular length in order to be detected. Designers deal with three main issues regarding interrupt signals connected to MCUs:

1. If the interrupt pulse is too short, the MCU may not detect it and the interrupt event will effectively occur without triggering the MCU to interrupt.

2. If the interrupt signal is active for too long of a pulse the MCU may attempt to process the single event as multiple repeated interrupts. While having the MCU automatically mask the interrupts helps this problem, the MCU hardware cannot completely eliminate a poorly implemented external interrupt signal.

3. If the interrupt signal is noisy or bounces back and forth between high and low, a single interrupt event can trigger multiple interrupts. This is typically the result of using a physical button or switch as an interrupt signal because mechanical contacts create a bounce situation that is described below.

The ideal interrupt pulse length for the OTTER MCU should be four clock cycles to ensure proper acknowledgment and processing. This equates to 80 ns.

Button Bounce

Mechanical buttons or switches present design challenges when used in conjunction with “fast” devices such as MCUs. Mechanical contacts have a tendency to “bounce” when actuated. This means that when a switch is connected or turned on, the output can “bounce” back and forth between “on” and “off” before the switch eventually settles in the “on” state. The result is that one intentional physical button press may result in many unintentional button presses in the microsecond timescale. Because MCUs typically execute instructions in the nanosecond time range, hardware designers and/or embedded programmers must account for this potential of unintentional presses in order to ensure the device operates properly. Figure 8.2 shows a example of a bouncing switch (source:https://softsolder.com/2012/07/13/contact-bounce-why-capacitors-dont-fix-it/).

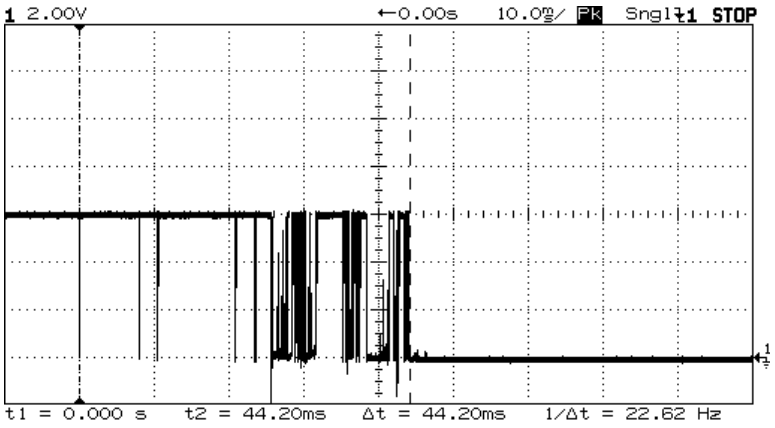


Figure 8.2: Example of Button Bounce

Approaches to handling switch bounce generally fall into two categories: hardware solutions or software solutions. The preferred solution for the OTTER MCU is to handle debounce in hardware as there is sufficient unused hardware resources on the FPGA. Moreover, the software solution unnecessarily complicates the program code, which should be avoided whenever possible. This assignment depends on multiple button presses acting as interrupts, so the button input will need to be debounced to work properly.

The hardware to do this is provided in a debouncer one-shot. This device actually serves two functions: it debounces the button input and provides an output with “one-shot” characteristics of a single pulse of 80 ns

that is ideal for an interrupt input. A button press typically lasts several milliseconds which is long enough to execute an ISR. Even if the button input was debounced, the button press would cause the long pulse issue listed above (Number 2 under Interrupt Signals). The one-shot creates a single short pulse after the button is released. The debouncer one-shot must be connected to the OTTER MCU interrupt signal and button input inside the OTTER WRAPPER.

Interrupt Test Program

The program in Code Segment 8.1 below is designed to test and verify the CSR and interrupt hardware on the OTTER similarly to the previous TestAll program. Interrupts will be enabled or disabled by the program depending on Switch 0. When interrupts are enabled, LED 0 will be turned on. Anytime after interrupts are enabled, pressing the button connected to the interrupt signal will trigger an interrupt. Every interrupt will increment a count saved in the data segment and display it on the 7 segment display. If any errors are detected, the program will go to a FAIL section and output an error code with 0xFF on the 7 segment display. Use RARS to assemble and step through the code to understand how it works. Interrupts can be simulated by setting the MIP register in RARS to 1 in RARS. Export a memory dump of the text and data sections as hexadecimal text to create a machine code mem file to add to the OTTER in Vivado. (Side note: Code Segment 8.1 is a good example of proper code formatting, commenting, and register use practices At this point, your assembly code should be structured, formatted, and commented equally well.)

Assignment

1. Modify the OTTER MCU such that it can process interrupts and execute the remaining OTTER assembly instructions (CSRRx and MRET). Create and connect the CSR module. Modify the Control Unit modules to add the interrupt state and the new instructions in the execute state.
2. Modify the OTTER Wrapper to connect the leftmost button on the development board (Basys3) to the input of the provided debouncer one-shot. The output of the debouncer one-shot connects to the interrupt input of the OTTER MCU.
3. Test the results of the previous two steps using the test program in Code Segment 8.1.
4. Demonstrate the working test program to your instructor or TA.

Deliverables

1. Behavioral Descriptions of CSR
2. Structural Design of CSR
3. Synthesis Warning of OTTER Wrapper with CSR and button debouncer.
4. SystemVerilog Code
    a. CSR
    b. Control Unit Decoder
    c. Control Unit FSM

5. Verification waveform showing (Can be multiple waveforms or a single waveform with the times specified when each of the below occur)
    a. Executing a CSRRW instruction
    b. Executing a CSRRS instruction
    c. Executing a CSRRC instruction
    d. Executing an MRET instruction
    e. Interrupt state
6. Demonstrate your working Interrupt test program to your instructor or a lab assistant.

```
################################################################################
# Main initializes registers, including CSR, clears the 7 seg display, checks
# the SWITCHES to either enable or disable interrupts (SW0), and checks a
# register (s1) acting as an interrupt flag. If the flag is set, it will update
# the 7 seg display with the updated interrupt count which is saved in the data
# segment to save register use.
#
# When changing any CSR register (MSTATUS, MTVEC) the register is read after
# writing and verified to change correctly. If a value doesn't match, an error
# code is set (s3) and the program goes to a fail loop that sets 7 Seg to
# 0xFFxx with xx being the error code (listed below).
#
# The ISR sets the interrupt flag  register (s1). The current interrupt count
# is read from the data segment, incremented, and written back. MSTATUS is also
# read to verify the MIE pending bit is set accordingly. If an error is
# detected an interrupt error flag register (s2) is set.
#
# Fail Codes (s3)
# 1 - Error setting MTVEC
# 2 - Error setting MSTATUS
# 3 - Error clearing MSTATUS
# 4 - Error with MSTATUS in the ISR
################################################################################
# Register Use Table
# s0 : MMIO
# s1 : Interrupt Flag
# s2 : Interrupt error Flag
# s3 : Fail Code
################################################################################
.eqv MMIO,   0x11000000          # MMIO address and offsets
.eqv LEDS    0x20
.eqv SEV_SEG 0x40
.eqv STACK   0x10000
.eqv INT_EN, 8                   # enable interrupts MSTATUS = 0x08

.data                            # keep interrupt count in data segment
INTR_COUNT: .half 0
```

```
.text
INIT:       li    sp, STACK          # setup sp
            li    s0, MMIO           # setup MMIO pointer
            la    t0, INTR_COUNT
            sh    x0, 0(t0)          # clear interrupt count
            la    t0, ISR
            csrrw x0, mtvec, t0      # setup ISR address
            addi  t1, x0, 0
            csrrw t1, mtvec, t0      # read mtvec
            addi  s3, x0, 1          # ERROR Code 1
            bne   t0, t1, FAIL       # check mtvec
            addi  s1, x0, 0          # clear interrupt flag
            addi  s2, x0, 0          # clear interrupt error flag
            sw    x0, SEV_SEG(s0)    # clear 7 seg

LOOP:       lw    a0, 0(s0)          # read SWITCHES
            andi  a0, a0, 0x01       # Mask SWITCH 0
            sw    a0, LEDS(s0)       # set LEDs
            call  SET_INT            # subroutine to enable or disable interrupts
            beqz  a0, NO_ERROR       # return 0 if no error
            add   s3, a0, x0         # copy error code
            j     FAIL

NO_ERROR:   beq   s1, x0, LOOP       # check for interrupt flag
            la    t0, INTR_COUNT
            lhu   t0, 0(t0)          # not zero, so read new interrupt count
            sh    t0, SEV_SEG(s0)    # update 7 seg with new count
            addi  s1, x0, 0          # clear interrupt flag
            beqz  s2, LOOP           # check interrupt error flag
            addi  s3, x0, 4          # set error code for mstatus interrupt

FAIL:       li    t0, INT_EN         # Fail State
            csrrc x0, mstatus, t0    # disable interrupts
            li    t0, 0xFF00
            or    s3, s3, t0         # ERROR Code
            sw    s3, SEV_SEG(s0)    # display ERROR on 7 Seg
STOP:       j     STOP

###############################################################################
# Set/Clear Interrupts Subroutine - Subroutine to set or clear MIE bit of
#   MSTATUS. After updating MSTATUS, register is read back and checked. If the
#   values don't match, an error code is returned
# a0: 1 - enable, 0 - disable
# return a0: error code (0 = no error)
###############################################################################
SET_INT:    li    t0, INT_EN
            beqz  a0, SET_INT_DIS    # check for enable / disable MIE
            addi  a0, x0, 0          # clear return value (error code)
            csrrs x0, mstatus, t0    # enable interrupts
```

```
            addi  t1, x0, 0
            csrrc t1, mstatus, x0     # read mstatus
            and   t1, t1, t0          # mask MIE bit
            beq   t0, t1, SET_INT_RET # check MIE bit for error
            addi  a0, x0, 2           # set error code return value
            ret
SET_INT_DIS: addi  a0, x0, 0          # clear return value (error code)
            csrrc x0, mstatus, t0     # disable interrupts
            addi  t1, x0, 0
            csrrs t1, mstatus, x0     # read mstatus
            and   t1, t1, t0          # mask MIE bit
            beqz  t1, SET_INT_RET     # check MIE bit for error
            addi  a0, x0, 3           # set error code return value
SET_INT_RET: ret

###############################################################################
# ISR - Sets interrupt flag (s1) and increments count saved in saved in data
#   segment. Also checks MPIE and MIE bit of MSTATUS and sets interrupt error
#   registers accordingly
###############################################################################
ISR:        addi  sp, sp, -8          # push t1, t2 to stack
            sw    t1, 4(sp)
            sw    t2, 0(sp)
            addi  s1, x0, 1           # set interrupt flag
            la    t1, INTR_COUNT
            lhu   t2, 0(t1)           # read current interrupt count
            addi  t2, t2, 1           # increment interrupt count
            sh    t2, 0(t1)           # save new interrupt count
            li    t2, 0x80            # MPIE and MIE bit mask
            csrrs t1, mstatus, x0     # read mstatus
            beq   t1, t2, ISR_RET
            addi  s2, x0, 1           # set interrupt error flag
ISR_RET:    lw    t2, 0(sp)           # pop t1, t2 from stack
            lw    t1, 4(sp)
            addi  sp, sp, 8
            mret
```

Code Segment 8.1: Interrupt Test Program

Software Assignments

Software Assignment Report Format

FlowChart **(30)**

A flow chart of the program. The flow chart should be detailed enough that another person could write a complete program from the flow chart that functions the same way, but it does not necessarily need to have a block for each instruction. Recommend software for creating a flow chart: Diagram.net (formerly Draw.io), Visio (free at https://azureforeducation.microsoft.com/devtools), Lucidchart, or Dia.

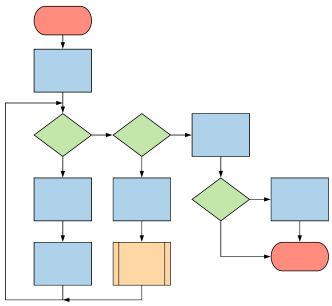
Verification by Simulation Results **(30)**

No simulation or timing diagram is visible with the RARS simulator so create a table with the test cases that were used with the RARS simulator to verify the code works as intended. Obviously, it will not be feasible to test every possible input combination, so a sufficient number of test cases should be chosen to ensure the code works for every input possible. This requires test cases to be chosen intelligently to discover any potential bug or logic issue. For the test cases used, give some explanation on why those specific cases were chosen.

Assembly Source Code **(40)**

Provide the assembly source code for the problem. The source code must be readable with proper spacing and tabbing. Use good label names. If registers are used for single variable purposes, add comments detailing their use. The source code should also contain comments for understanding and readability. To make the code readable in the report, use a fixed-width font (Courier or Monospace), single line spacing, and avoid line wrapping (Suggest page margins of 0.2 to 0.5). Code also needs to be highlighted which can be done with an online tool like https://pinetools.com/syntax-highlighter (Select mipsasm language and the Xcode style suggested). If using Google Docs, you can also use a tool like Code Blocks (Language mipsasm, Theme Xcode suggested). *Never use screenshots of code.*

Software Assignment 1 - Introduction to Assembly Language Programming



Learning Objectives:

- To understand the basics of assembly language programming.
- To understand the basics of flow charting
- To understand how to use an assembly language simulator to analyze an assembly language program.

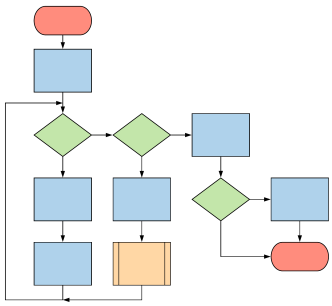
Assignment:

First, create a flowchart that will perform the following behavior. Then implement the flowchart with OTTER MCU assembly language using only the instructions in the OTTER assembly manual. Make sure your code is formatted properly including comments. Use the RARS simulator to ensure the program performs as desired.

1. Read three 16-bit unsigned values consecutively from SWITCHES (address 0x11000000), add them together, and output the result to LEDS (address 0x11000020). Assume the input values are 16-bit unsigned values. (Assume it is possible for the switches to change between each time they are read)

2. Read an input from SWITCHES (address 0x11000000). Assume the input is a 16-bit signed value in 2's complement (RC). Change the sign of the input and output the result to LEDS (address 0x11000020). The result should also be a 16-bit signed value in 2's complement (RC).

Software Assignment 2 - Conditional Statements



Learning Objectives:

- To understand how to flowchart conditional statements
- To understand how to use branching to create conditional assessments in assembly language programming.
- To gain experience using an assembly language simulator to analyze an assembly language program.

Assignment:

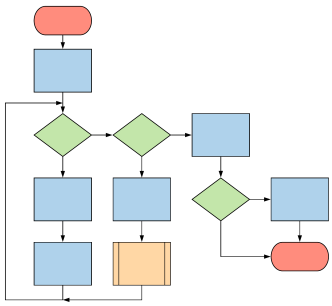
First, create a flowchart that will perform the following behavior. Then implement the flowchart with OTTER MCU assembly language using only the instructions in the OTTER assembly manual. Make sure your code is formatted properly including comments. Use the RARS simulator to ensure the program performs as desired.

1. Read a 32-bit unsigned value from SWITCHES (address 0x11000000). If the input is greater than or equal to 32,768, the value is divided by 4. You can ignore any remainder. If the value is less than 32,768, the value is multiplied by 2. The result should be written to the 7 SEGMENT (address 0x11000040).

2. Read a 32-bit unsigned value from SWITCHES (address 0x11000000). If the input value is a multiple of 4, all of the bits should be inverted, otherwise, if the input value is odd, add 4095 and divide the result by 2, otherwise subtract 1 from the value. The result should be written to the 7 SEGMENT (address 0x11000040).

Assume the BASYS3 SWITCHES are expanded to 32 individual switches and 7 SEGMENT is expanded to 8 digits.

Software Assignment 3 - Loops



Learning Objectives:

- To understand how to flowchart loops
- To understand how to use branching to create loops in assembly language programming.
- To gain experience using an assembly language simulator to analyze an assembly language program.

Assignment:

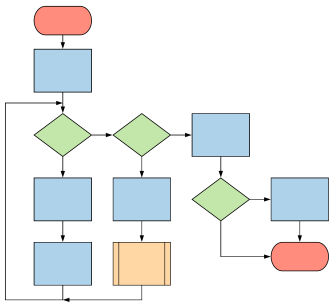
First, create a flowchart that will perform the following behavior. Then implement the flowchart with OTTER MCU assembly language using only the instructions in the OTTER assembly manual. Make sure your code is formatted properly including comments. Use the RARS simulator to ensure the program performs as desired.

1. Read a 32-bit value from SWITCHES (address 0x11000000). Assume the 32-bit value is the combination of two 16-bit unsigned values. Split the 32-bit input into two 16-bit values (the upper and lower 16-bits). Assume both of the 16-bit values are unsigned. Multiply the two 16-bit values together and output the result as a 32-bit unsigned value to 7 SEGMENT (address 0x11000040).

2. Read a 32-bit value from SWITCHES (address 0x11000000), delay for 0.5 seconds, and then output the value to 7 SEGMENT (address 0x11000040). The OTTER MCU operates at 25 MHz so each instruction takes 40 ns to run. (*Hint: Use loops.*) **There is no method to verify the run time in RARS and RARS does not execute at the same speed as the OTTER.** You will need to verify your timing by showing your calculation for the number of instructions being performed between the load (reading from the switches) and store (writing to the 7 segment display) instructions.

Assume the BASYS3 SWITCHES are expanded to 32 individual switches and 7 SEGMENT is expanded to 8 digits.

Software Assignment 4 - Arrays



Learning Objectives:

- To understand how to flowchart using arrays in assembly.
- To understand how to use arrays in assembly language programming.
- To understand how to sort an array of values.
- To gain experience using an assembly language simulator to analyze an assembly language program.

Assignment:

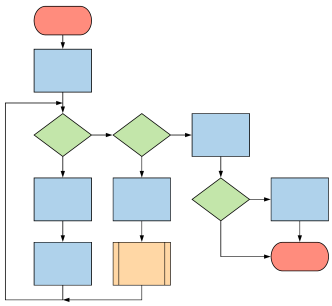
First, create a flowchart that will perform the following behavior. Then implement the flowchart with OTTER MCU assembly language using only the instructions in the OTTER assembly manual. Use register alternative names (ie s0, t0) instead of register numbers (ie x5, x10). Make sure your code is formatted properly including comments. Use the RARS simulator to ensure the program performs as desired.

1. Using `.data`, define an array in the data segment that contains the first 25 values of the Fibonacci sequence (starting with 0 and ending with 46368). Write a program that progresses through the array and calculates the difference between values that are 3 spots away from each other. For example, the first value (0) and the 4th value (2) is a difference of 2. The next calculation would be between the 2nd (1) and 5th (3) values. When no item exists 3 spots away, no difference can be calculated. Each difference should be output to LEDS (address 0x11000020).

2. Create an array of 10 32-bit values. Read 10 32-bit values consecutively from SWITCHES (address 0x11000000) saving them in the array. After the 10th value has been saved, the array should be sorted from least to greatest. The sorted array should be output to LEDS (address 0x11000020) in order from least to greatest.

Assume the BASYS3 SWITCHES and LEDS are expanded to 32 individual switches and LEDs.

Software Assignment 5 - Division / Stack



Learning Objectives:

- To understand how to flowchart binary division operations.
- To understand how to perform binary division in assembly language programming.
- To understand how to flowchart using the stack in assembly.
- To understand how to use the stack in assembly language programming.
- To gain experience using data segments in an assembly language program.
- To gain experience using an assembly language simulator to analyze an assembly language program.

Assignment:

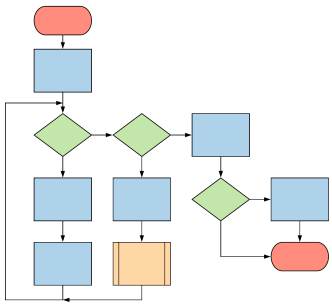
First, create a flowchart that will perform the following behavior. Then implement the flowchart with OTTER MCU assembly language using only the instructions in the OTTER assembly manual. Use register alternative names (ie s0, t0) instead of register numbers (ie x5, x10). Make sure your code is formatted properly including comments. Use the RARS simulator to ensure the program performs as desired.

1. Read two 16-bit unsigned values from an array in the data segment. Divide the first value by the 2nd and output the quotient (result) to 7 SEGMENT (address 0x11000040) and the remainder to LEDS (address 0x11000020).

2. Read 32-bit values repeatedly from SWITCHES (address 0x11000000). Each value should be added to the stack until the value 0xFFFFFFFF is read. When 0xFFFFFFFF is input the program will not add it to the stack but instead will output all of the saved values to LEDS (address 0x11000020) in the reverse order they were input. Specify the limits of your program. (Assume it is possible for the switches to change between each time they are read)

Assume the BASYS3 SWITCHES and LEDS are expanded to 32 individual switches and LEDs.

Software Assignment 6 - Subroutines



Learning Objectives:

- To understand how to flowchart subroutines.
- To understand how to use subroutines in assembly language programming.
- To gain experience using an assembly language simulator to analyze an assembly language program.

Assignment:

First, create a flowchart that will perform the following behavior. Then implement the flowchart with OTTER MCU assembly language using only the instructions in the OTTER assembly manual. Use register alternative names (ie s0, t0) and utilize registers according to standard use practices. Make sure your code is formatted properly including comments. Use the RARS simulator to ensure the program performs as desired.

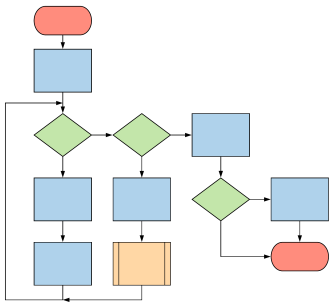
1. Create a subroutine that will divide a value by 10. The subroutine should utilize appropriate registers for passing parameters, returning results, and only affect temporaries. Write a main program that reads a 16-bit unsigned value from an array in the data segment and converts the value into a 5 digit, BCD (binary coded decimal) value using the divide by 10 subroutine. For example, if the input value is 0x3A6C (14956 decimal), it should be split into 5 nibbles (4-bits) for 1, 4, 9, 5, and 6, so the final answer would be 0x14956. The resulting 20-bit BCD can be written as a 32-bit word to another array in the data segment. The arrays can be of any length.

2. Create a subroutine that implements the following greatest common denominator function given below. The subroutine should utilize appropriate registers for passing parameters, returning results, and only affect temporaries. Write a main program that reads two 16-bit unsigned values from an array in the data segment, uses the subroutine to find the GCD, and saves the result to another array in the data segment. *Do not replace the subroutine call with a loop.* You can assume all values are unsigned and greater than 0. Specify any limits of your program.

```
int gcd(int a0, int a1){
  if (a0 > a1)
      a0 = gcd(a0-a1,a1);
  else if (a0 < a1)
      a0 = gcd(a0,a1-a0);
  return a0;
}
```

Code S6.1: GCD Subroutine Code

Software Assignment 7 - Interrupts



Learning Objectives:

- To understand how to flowchart interrupts.
- To understand how to use interrupts in assembly language programming.
- To gain experience using an assembly language simulator to analyze an assembly language program.

Assignment:

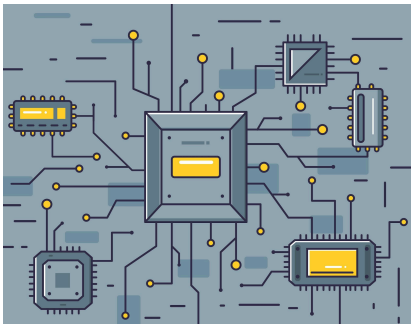
First, create a flowchart that will perform the following behavior. Then implement the flowchart with OTTER MCU assembly language using only the instructions in the OTTER assembly manual. Use register alternative names (ie s0, t0) and utilize registers according to standard use practices. Make sure your code is formatted properly including comments.

1. Write an interrupt driven program that will turn the LEDs (address 0x11000020) on or off. Every interrupt will alternately toggle certain LEDS on or off. The LEDs that change (toggle) are the ones corresponding to the value on the SWITCHES (address 0x11000000). When an interrupt occurs, if the switch for a specific bit is 1, the corresponding bit on the LEDs will toggle (change from on to off or off to on).

Example:	LEDS start as	0101 0011 1010 1100
	Interrupt occurs	
	SWITCHES are	1111 0000 0110 1001
	LEDS change to	**1010** 0011 1**10**0 **0**10**1**

2. Write an interrupt driven program that toggles the LEDs (address 0x11000020) based on the SWITCHES (address 0x11000000) when an interrupt occurs just like problem 1 above. However, if on two consecutive interrupts the switches have the same value, the program will turn all of the LEDs off until BUTTON 0 is pressed. Button 0 is connected to bit 0 of address 0x11000200. *Button 0 is not connected to any interrupts.* This means the LEDs will all stay off until button 0 is pressed. When button 0 is pressed, the program will light up the LEDs that were turned on before the duplicate switches were detected and return to the previous functionality of toggling the LEDs as before. You can assume the same switch values will never repeat on more than two consecutive interrupts. (*Hint: The data segment can be used to save a program state or pass values to / from an ISR*)

Final Design Project



Learning Objectives

- To gain experience interfacing external devices to a digital system
- To gain experience programming a complex self-contained digital device
- To gain experience debugging and testing a complex digital system

Assignment

The final project should be the culmination of everything learned in this course. It will use the fully functional OTTER as the core of a digital device. The project needs to use an assembly language program that runs on the OTTER and performs a useful function or utility. It may control devices, be a game, or some interesting technology demo. The assembly application needs to control or use at least one new peripheral, e.g. the VGA,, keyboard, speaker. The device may also utilize the buttons, switches, or LEDs of the Basys3 board. The application should operate continuously once launched, e.g. it should run until the user wants to quit. If it is a game, it should keep score in some fashion, e.g. show the score on the seven-segment displays. The final project is designed to allow you to use class time to learn something that interests you and get course credit for it. This is a choose your own adventure to try something new or reinforce a topic you are interested in.

Suggested Approach (AKA good software / system design methodology)

1. Consider the hardware and how it interfaces with the OTTER. Design how the peripheral connects to the OTTER (IOBUS IN / OUT).
2. Create a fresh Vivado design project. Import only the needed design source files for the OTTER. ***When importing make sure to select the option, copy sources into project.*** This will create a new copy of the source file in the project to ensure accidental changes to previous projects will not create problems. This will also make it easier to share the project as one of the deliverables.
3. Use the included sample test programs or write a simple one to test the peripheral with the OTTER. This should verify the hardware is working and demonstrate how to use it. This could be expanded to create basic subroutines for interacting/interfacing with the peripheral.
4. Design the main program structure. Trace out the main program loop. Consider when outputs are updated and when inputs are read. Consider timing and if/when delays may be necessary. Consider an intro that may wait for an input to start or an exit screen that waits before restarting the program.
5. Break the program down into smaller chunks that can be written and tested individually. The more modular the program design is the easier it will be to build and debug. Look for tasks that are repeated and create subroutines. The main program could be a simple loop of subroutine calls.
6. Consider register usage. Specify registers that may be used to keep MMIO addresses or reserved for interrupt uses. Write comment blocks at the top of the program listing them.

7. Write and test individual chunks of code. Start with the individual subroutines specified. Be sure to follow standard register usage with subroutines (arguments, temporaries, saved). Test each subroutine as they are written. Write the program in chunks, testing it after each new piece is added. *Do not write a big monolithic program and only test it after everything is written.*
8. Enjoy the extra hours of sleep while others have to debug problems because they didn't have the time to fully follow each of the steps above.

General Tips

- Never forget, weeks of programming can save you hours of planning!
- Comment code as it is written to help organize it as it is built. Comments are also good for identifying different sections or modules to easily find segments when debugging.
- Format code well (proper indentation and white space) as it is written to make debugging easier.
- Every time you change the mem file in Vivado, a full synthesis must be redone. Vivado does not do this by default so it must be specifically selected.
- Remember button presses are 100+ ms while instructions are executed every 40ns.
- When debugging, use the LEDs and 7 segment to display data values, indicators, or counts to show where in the program things are happening or how often some code segment is repeating.
- Add pause loops waiting for a button or switch to be pressed can give time to see what is displayed on outputs when the outputs change quickly.
- Make a backup of working code between major revisions or before each new section is added so that development can restart at a working stage if the code is lost or debugging edits break working code.
- Do not become emotionally attached to any code. It is often easier (and quicker!) to start over and rewrite code than it is to fix poorly designed or implemented code. Yes, it may have taken 20 hours to write some piece of code, but debugging and fixing it may take another 20 hours while starting over and writing it with a better plan would only take 10.

Final Report

The final report will be a very different document from the previous lab reports. The final report will be a technical owner's manual that describes the functionality and design of the digital device. The purpose of this document is to tell others what the device does, how to operate it, and how it is built. The report should include the following sections.

Introduction

This section of the documentation should introduce the device built. Explain what the device does. If it is a game, what is the basic game play? How many levels? For a generic digital device, what are the device's specifications? What is the operating range and precision of the sensor measurements? How fast is the input checked? What is the range of input and output values? Essentially what information would a potential user need to know to decide if they wanted to use this device or if it would meet their needs?

Operating Instructions

This section of the documentation should explain how to use the device. This would be like the owners manual for the device or game. If it is a game, how is it played? How do you score, win, or lose? What is displayed and where? How is the device used? Do not assume the user is familiar with the Basys3 board. (You do not have to

explain how to program the Basys3 board from Vivado, you can assume the device is “preprogrammed” when turned on) Pictures are very useful here to aid in explaining how to use your device. Refer to the example Operational Manuals on Canvas.

Peripheral Details (only if new, and not provided on Canvas)
*This section is **not** needed if using a peripheral that was provided on Canvas*. This section is only needed if this project includes a new hardware peripheral that was created for this project. Describe the behavior of the designed peripheral(s). This should be a short synopsis that explains the functionality of the peripheral(s). Include a block diagram of the designed peripheral(s). Show all inputs and outputs, specifying bus widths where applicable. Also include an elaborated design schematic of the hardware to show how it was implemented. Specify the working parameters of the design and how it operates. Specifications could include: how fast can this part operate, does it require specific timing or clocks, range and precision of output values, or memory size. If someone wanted to use this peripheral in their design, what would they need to know to connect this part and use it correctly?

External Circuit Peripheral (only if new, and not provided on Canvas)
*This section is **not** needed if using a peripheral that was provided on Canvas*. If the device used any external connections or circuitry for a peripheral include a schematic of how the circuit components are connected and the external device is connected to the Basys3 board.

Software Design
Give an explanation of the overview of how the program functions. This should give a general breakdown of the program flow and the logic behind the approach taken. *A flow chart must be included, but a flowchart by itself is typically inadequate.* The flow chart for the final report *cannot* be hand-drawn. A good approach to take is to start by explaining the main program logic. When explaining the main program logic, only describe what subroutines do when they are called from the main program. Include a flow chart for main. If the flowchart is large, break it into sections across pages and use off page connectors. After explaining the main program, explain how each subroutine works individually. Include flowcharts for each of these. Organizing the text descriptions together with a flowchart allows for easier comprehension by the reader.

Appendix
1. Full assembly code listing with comments (this is the assembly code I will use for grading the project)
2. Peripheral SystemVerilog code listing (only if new or significantly changed)

Deliverables
The final project will have 2 deliverables due at different times.

1. A copy of the working project folder with all of the sources. This is easy to create by copying the assembly code file into the root Vivado project folder and then zipping the folder. This zip file will be submitted the day demos are due. This is file is used for archival and reference use only.

2. A final project report that includes the sections listed above following the same formatting practices used for hardware and software assignments. It is important to include all new source code in the appendix of the report for grading purposes. The report will not be due with the demo.

Grading

The final project does not use a standard rubric because every project is unique. However, the grade is broken into 2 parts and assigned for each deliverable individually.

1. Project Quality - A general assessment of the project. This is not just, or even largely, based on if the project works. A mostly or partially working difficult project can receive a higher score than a working simple project. The project performance, difficulty, complexity, creativity, and code quality (HDL and assembly code) all factor in.

2. Project Report - A grade on the quality of the final report. Does the report look like effort was invested in creating it? A poor or non-working project can receive a significantly higher grade by creating a high quality report.

General Considerations on Grading Final Projects

The typical week throughout the quarter included 1 Hardware assignment and 1 Software assignment to be completed concurrently. For the final two weeks, only HW 8 and the final project are assigned. This means 1 week of time on a software assignment and 1 week of time on a hardware and software assignment is expected to be spent working on the final project. A number of class days are reserved for solely working on the final project as well. Also consider the weight of the course grade assigned to a single hardware or software assignment compared to the weight assigned for the final project. In general, if the final project source listing is the same size as a typical hardware or software assignment, an above average grade should not be expected. (*Side note: the code size is not an absolute grading measurement. Copying and pasting the same group of instructions 100 times does not make the quality of the project better. In fact, writing code in that manner instead of using loops and subroutines would receive a lesser grade.*) The size, scope, and complexity of the hardware that is interfaced with the OTTER is also factored in so that a high quality project with even a small demo assembly code program is possible. The details for these factors should be included and highlighted in the project report. The project report is used in assessing the project quality, including all source code listings, even though these do not directly factor into the project report grade.

The final project can be worked on in groups. Groups are even encouraged to develop more robust projects. Understandably a project worked in a group will have higher expectations than a project done individually. This applies both to the project quality and the project report. Working in groups can be synergistic and more efficient in designing and debugging with multiple perspectives. Creating diagrams, screenshots, and photos for the report can also be divided between multiple people. Although group sizes are not limited, I would caution working in groups larger than 2 because the effort added from more than 2 people diminishes but the expected output is not.

As a final note, do not worry if technical debt accrued through the quarter or a series of missteps results in a less than stellar final project and/or report, or even no final project at all. While the grade for the final project is larger than any previous individual assignment, it is still a small factor in the final grade of the course. Completing all the hardware and software assignments is a bigger achievement and is weighted accordingly. The goal of the course is learning, and the final project was designed to allow customized learning for everyone in the class. It was not designed to create work that earns credit. Completing a final project is not necessary to fully understand the core material in the course. For this reason, and due to grading time limitations, late project demos and report submissions are not accepted. The final project will never cause a failing grade because learning the course material is possible regardless of the final project.