CPE 233 Lab Manual

SystemVerilog RAT Edition

Version 1.0

RAT Assignments

RAT Assignment Report Format

Black Box Block Diagram **(5)**

Include a block diagram of the designed hardware component(s). Show all inputs and outputs, specifying bus widths where applicable.

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. Use this image to show 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 must be fixed or redone. For proper operation of the RAT MCU, no latches can exist.

Latches are a major cause of problems when assembling the RAT 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 RAT MCU on the FPGA hardware.

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. 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 RAT 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 RAT 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, it can be easily highlighted with https://emn178.github.io/online-tools/syntax_highlight.html Select Verilog language and the Xcode style.

RAT Assignment 1 - Program ROM and Assembly Programming

(and a dash of Reverse Engineering)

Learning Objectives

- To understand the makeup and organization of the Program ROM (ProgROM)
- 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 RAT 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 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 RAT 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 1024 chunks of 18-bits. The 1024 chunks or locations are identified by addresses 0 to 1023. Each address (PROG_ADDR) identifies an individual 18-bit logic vector that corresponds to a single instruction (PROG_IR). The ProgROM outputs a single 18-bit instruction synchronously, on each rising edge of the clock. The makeup of the ProgROM is divided between two files, the ProgROM.sv and prog_rom.mem.

The ProgROM.sv is a SystemVeriog source file that creates a generic memory device of size 1024x18. 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 prog_rom.mem file which is used to initialize the ProgROM. The code for the ProgROM.sv file is given below.

The prog_rom.mem file contains the raw data that should be loaded into the ProgROM. This file is a raw listing of 1024 18-bit instructions in hex. This file is automatically created by the RAT Simulator when it successfully assembles the assembly code program. The prog_rom.mem will need to be replaced or contents updated for every program that needs to be loaded into the RAT MCU.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Company:   Cal Poly
// Engineer:  Paul Hummel
//
// Create Date: 06/28/2018 01:00:34 AM
// Module Name: ProgRom
// Target Devices: RAT MCU on Basys3
// Description:  Generic 1024x18 ROM device
//
// Dependencies: prog_rom.mem file is a raw listing of 1024 18-bit hex values
//               prog_rom.mem file is automatically created by the RAT
//               assembler / simulator from an assembly code program.
//
// Revision:
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////


module ProgRom(
    input PROG_CLK,
    input [9:0] PROG_ADDR,
    output logic [17:0] PROG_IR
    );

    (* rom_style="{distributed | block}" *)  // force the ROM to be block memory
    logic [17:0] rom[0:1023];

    // initialize the ROM with the prog_rom.mem file
    initial begin
        $readmemh("prog_rom.mem", rom, 0, 1023);
    end

    always_ff @(posedge PROG_CLK) begin
        PROG_IR <= rom[PROG_ADDR];
    end

endmodule
```

ProgROM.sv SystemVerilog Code Listing

Example of Reverse Engineering ProgROM

Please note that this program will not assemble on the latest version of the RAT assembler as it starts at address 0x00. The assembler requires address 0x00 to remain unused. The .ORG value in this program would

need to be moved to a higher value, e.g. 0x10, for this program to assemble without error. All other aspects of this example are correct.

Consider the following RAT assembly language program

```
.EQU LED_PORT = 0x10                ; port for output
.CSEG
.ORG          0x00                  ; code starts here

main_loop:    MOV     R10,0x05
              MOV     R11,0x64
              ADD     R10,R11
              ADD     R10,0x14
              MOV     R20,R10
              OUT     R20,LED_PORT
              BRN     main_loop      ; endless loop
```

Code Segment 1: Example Program

After assembling the program in the RAT Simulator, the following prog_rom.mem file is generated.

```
0: 36A05
1: 36B64
2: 02A58
3: 28A14
4: 05451
5: 35410
6: 08000
7: 00000
```

Hex Listing 1: Hex Machine Code of Example Program

The assembly instruction can be recreated from the 18-bit machine code by comparing the opcodes and field codes of this instruction to the format outlined in the RAT Assembler Manual. The results are shown in Table 1 below.

ProgROM Address	Concatenation forming Machine Code	Assembly Instruction
0	11 0110 1010 0000 0101	MOV R10,0X05

1	11 0110 1011 0110 0100	MOV R11,0X64
2	00 0010 1010 0101 1000	ADD R10,R11
3	10 1000 1010 0001 0100	ADD R10,0X14
4	00 0101 0100 0101 0001	MOV R20,R10
5	11 0101 0100 0001 0000	OUT R20,LED_PORT
6	00 1000 0000 0000 0000	BRN main_loop

Table 1: Assembly Program Construction from Machine Code

Assignment

The assignment is broken into 2 parts.

***Part 1*:**

Code Segment 2 below shows Test Program A, an example RAT assembly program. Write and assemble this program in the RAT Simulator. (*Be sure to include a blank line after the last line of code.)*

```
.EQU LED_PORT = 0x10                ; port for output
.CSEG
.ORG          0x40                  ; code starts here

main_loop:    MOV     R10,0x05
              MOV     R11,0x64
              ADD     R10,R11
              ADD     R10,0x14
              MOV     R20,R10
              OUT     R20,LED_PORT
              BRN     main_loop      ; endless loop
```

Code Segment 2: Test Program A

Step through the program with the RAT Simulator and analyze Program A. Watch the contents of the various registers as the program executes. Complete Table 2 by including the following information after the simulator executes the listed instruction. The completed table should have one line for each instruction in the program, so add rows as needed. Include a copy of this completed table in the lab report. This table is intended to show the status of the RAT MCU as the code is executed, and is intended as a walk-through to help in understanding program operation during execution. Note that the instructions in Table 2 are not intended to correlate to Code Segment 2, but are just examples.

Address	Instruction	Destination Register	C Flag	Z Flag	OUT(port_id)

0x30	MOV R10, 0xA5	R10 = 0xA5	x	x	
0x31	ADD R10, 0x64	R10 = 0x09	1	0	
0x32	OUT R10, 0x40		1	0	0x09 => (0x40)
...					

Table 2: Program A Analysis Table

***Part 2*:**

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.

```
 0: 00000
... ...
40: 37DFC
41: 29D01
42: 37EFA
43: 37F05
44: 01EFA
45: 03EEA
46: 0820B
47: 00000
```

Hex Listing 2: prog_rom.mem Segment for Reverse Engineering

Complete Table 3 below from the reverse engineered code in the same manner as Table 2 was created from Program A.

Address	Instruction	Destination Register	C Flag	Z Flag	OUT(port_id)

Table 3: Reverse Engineered Program Analysis Table

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

1. Part 1

a. Completed program analysis Table 2 for Program A

b. Simulation of the ProgROM.sv using the prog_rom.mem module from Program A. Create a clock signal and address signal in a test bench. The address should increment for every clock pulse, starting at address 0 and incrementing until the entire program has been output.

2. Part 2

a. Completed disassembly table for constructing the reverse engineered prog_rom.mem segment similar to Table 1

b. Typed assembly code for the reverse engineered prog_rom.mem segment.

c. Completed Table 3 for the reverse engineered assembly program

d. Simulation of the ProgROM.sv using the prog_rom.mem module from the reverse engineered assembly program. Similar to Part 1, create clock and address signals and increment address from 0 until the the entire program has been output.

RAT 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 (counter) 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 RAT microcontroller those instructions are saved in the Program ROM (ProgROM) that was studied in Assignment 1. The PC is the component that creates the address signal for the ProgROM.

General Notes

A counter is a special form of register that performs operations associated with counting. Counters are generally operated synchronously, changing outputs in sync with the edge of a clock signal. Designing counters in SystemVerilog is relatively simple as an *n*-bit register of *n* flip-flops 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. HDL design language allows adders and incrementers to be designed behaviorally, allowing the synthesizer to optimize the implementation. This is not only simpler, but typically results in a better performing design. SystemVerilog allows addition of vectors (groups of bits). 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 1: Example SystemVerilog Addition

Circuit Details

Figure 1 below shows the top-level black box model of the Program Counter. Table 1 provides an overview of the PC's signals and operation.

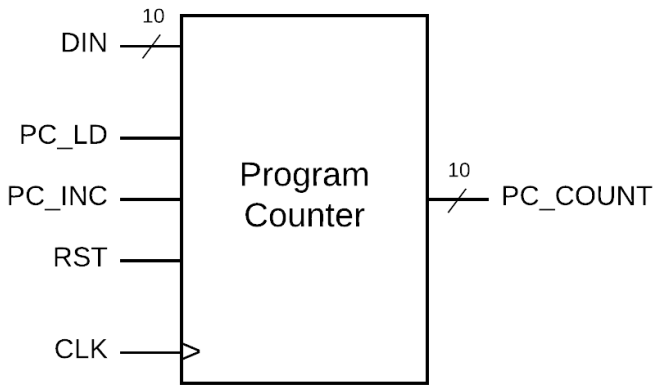


Figure 1: Program Counter Black Box Diagram

Signal	Comment
PC_COUNT	The current value in the PC. This value is used as an address for the ProgROM
RST	Active high synchronous reset. When RST = 1, the output of the PC is 0. This signal will have the highest priority.
PC_LD	Active high, synchronously load the value from DIN into the PC. This signal will have a higher precedence than PC_INC.
PC_INC	Active high, synchronously increments the value in the PC.
DIN	An unsigned 10-bit value that is loaded into the PC when PC_LD is high
CLK	Synchronizes all PC operations. Same speed as the system clock of the RAT MCU

Table 1: Overview of the Program Counter Signals

Figure 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 RAT MCU. The MUX inputs include:

1. FROM_IMMED: Branch and Call instructions cause the program to jump to a new location in the program. This new location is converted to an address that is hard coded into the instruction.
2. FROM_STACK: Some instructions cause the program to jump to a subroutine and the current location is saved to the stack. When the subroutine is completed, the stack is able to provide the address to return back to.
3. 0x3FF: Interrupts cause the PC to go to address 0x3FF, the last address of the ProgROM. This input is just a constant

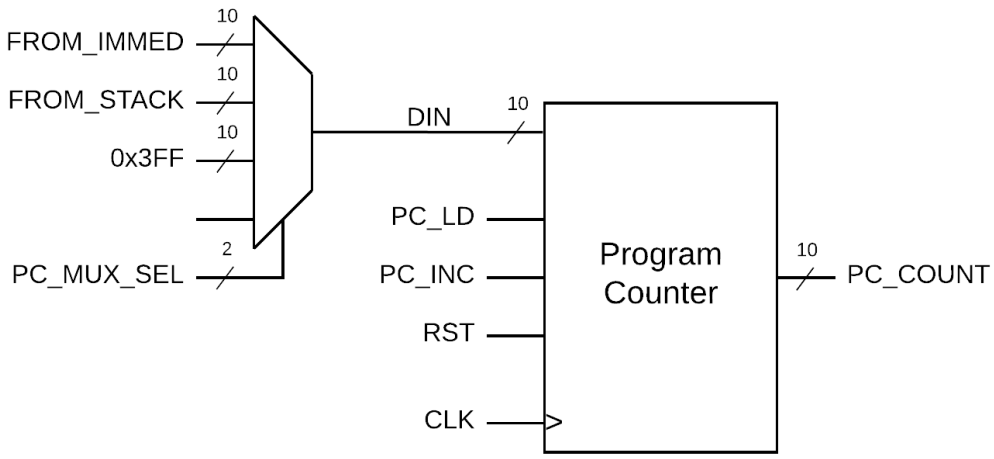
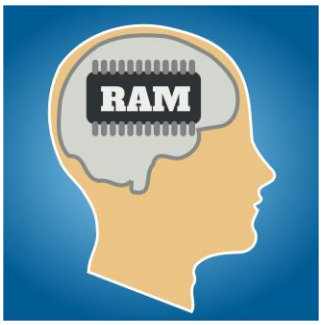


Figure 2: Program Counter with Input MUX

Assignment

Design a PC and associated input selection MUX. Use at least two components in the design, one for the PC and one for the MUX. These components can be either two separate modules (two-level design) or two always blocks in the same module (one-level design).

RAT Assignment 3 - Memory Fundamentals



Learning Objectives

- To implement the register file in the RAT MCU architecture.
- To learn how a register file operates and how it will work when integrated with the RAT CPU.
- To become familiar with the operation of the scratch memory, which the RAT CPU uses as Random Access Memory (RAM).

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 RAT MCU, the only control signal needed is a write enable (WR). The data entering the RAM module will only be saved when WR is active. All of the memory modules are synchronous and will only change on the edge of the clock signal.

RAT MCU RAM Memory

The RAT MCU has two RAM memory modules

1. 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 RAT MCU has 32 registers (R0 - R31) with each register occupying a row in the register file “table”. Each register is able to store 8-bits, making the RAT an 8-bit MCU and giving the register memory a size of 32x8. The register file is a dual port RAM which allows reading from two locations or addresses simultaneously. The RAM memory can be read asynchronously. The register file needs to address inputs (ADRX and ADRY) to access to registers. The output, the values saved in the two registers, is sent to DX_OUT and DY_OUT. ADRX and ADRY must be a value from 0 to 31, corresponding to register being accessed, and

DX_OUT and DY_OUT will be the 8-bit values saved in each of those respective locations. Only a single location, or register, can be saved per clock cycle. To save on complexity, the address used to save data to will reuse the ADRX input. The 8-bit value to be saved in the location ADRX is DATA_IN. While data is read from the RAM asynchronously, data is saved in the RAM memory synchronously on the rising edge of the clock. The data (from DIN) will only be saved if write enable (RF_WR) is high on the rising clock edge.

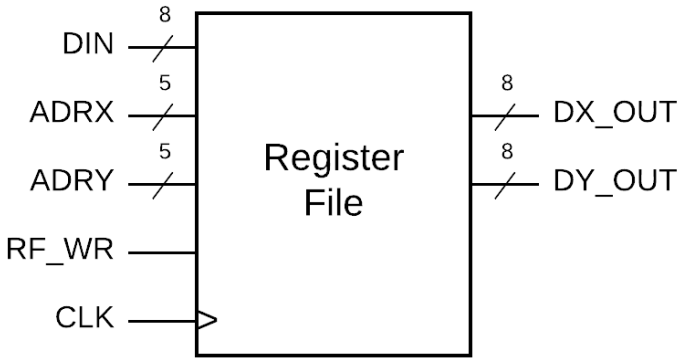


Figure 1: Register File Black Box Diagram

2. Scratch RAM: The scratch RAM in the RAT MCU serves two purposes. First, it provides a temporary scratch pad for saving data that will not fit into the register file and is not immediately necessary for operating on. Data can be directly transferred between the register file and the scratch RAM. The scratch RAM is also used for the stack memory of the RAT MCU. The stack is a memory construct that will be detailed in later assignments. The scratch RAM is a different size than the register file. It is 256x10 meaning there are 256 locations with each location storing 10-bits. When transferring data between the register file and the scratch RAM, only 8-bits are used. Those 8-bits are saved in the lower 8-bits of the 10-bit data of the scratch RAM. The scratch RAM is a single port RAM, so only a single address can be read from and written into. The same address is used for both operations (SCR_ADDR). A write enable (SCR_WE) control signal behaves the same as the register file, controlling when data (from DATA_IN) is saved. Data is read from the scratch RAM asynchronously and written to synchronously on the rising edge of the clock.

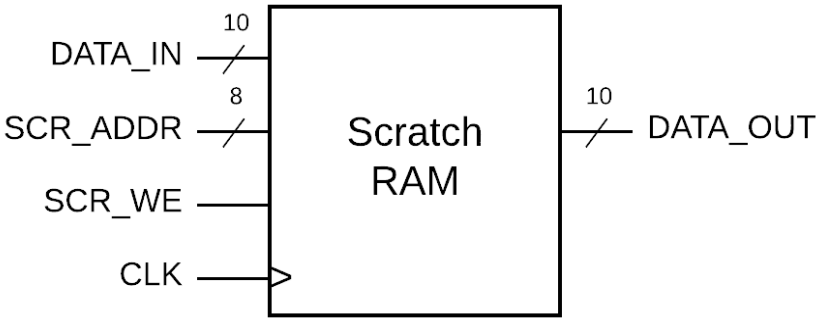


Figure 2: Scratch RAM Black Box Diagram

Memory in SystemVerilog with Arrays

Memory devices are easily defined in SystemVerilog with arrays. Arrays are an indexed group of signals of uniformed size and type. Typically arrays are created as indexed groups of vectors, although an array of single bit types is possible. The data width or size of the vectors that are grouped by index 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 creating a 512x16 memory unit.

```
// 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++) begin
    ram[i] = 0;
  end
end

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

Code Segment 1: SystemVerilog for a 512 x 16 Memory Device

Assignment

1. Using an array, design the REG_FILE memory module according to the black box diagram in Figure 1
2. Using an array, design the SCRATCH_RAM memory module according to the black box diagram in Figure 2.

RAT Assignment 4 - Arithmetic Logic Unit

Learning Objectives

- To understand the requirements of the Arithmetic Logic Unit (ALU) based on the RAT instructions it must support.
- To understand the design and operation of the ALU.

General Notes

The arithmetic logic unit (ALU) is the central part of any microprocessor and heart of the RAT MCU. The ALU is responsible for performing all of the arithmetic and logic operations including any bit-crunching required by all of the RAT 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.

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 SEL input will control which operation the ALU performs on the data inputs. The size of the SEL input will be determined by how many different operations are needed. The ALU in the RAT MCU is able to capable of 15 operations listed in Table 1 below. The RAT Assembler Manual provides an expanded explanation of each of the associated RAT instructions.

SEL	Instruction		SEL	Instruction
0000	ADD		1000	TEST
0001	ADDC		1001	LSL
0010	SUB		1010	LSR
0011	SUBC		1011	ROL
0100	CMP		1100	ROR
0101	AND		1101	ASR
0110	OR		1110	MOV
0111	EXOR		1111	unused

Table 1: ALU Selection Table

The ALU is a unique component in the RAT MCU as the only module that is completely combinational, it has no flip-flops or clock signals. This makes the ALU critical for timing requirements in the RAT MCU. A poorly

designed ALU can cause a variety of issues that manifest in the hardware implementation but are hard to diagnose in simulation. Figure 1 shows a black box diagram of the RAT ALU.

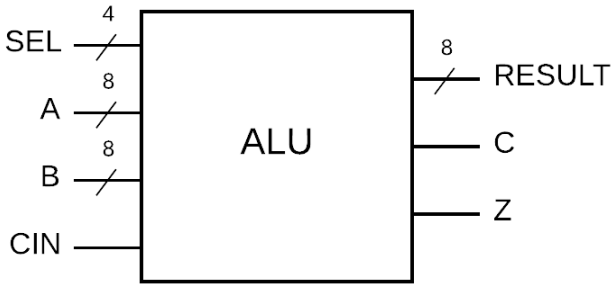


Figure 1: ALU Black Box Diagram

Assignment

Design and implement an ALU that performs the 15 RAT arithmetic and logic functions as listed in Table 1 above. A good way to design the ALU is to put the result of each operation into a nine-bit signal and then assign RESULT to the lower 8 bits of that signal and C to MSB of the nine-bit signal.

Verification

Because of the critical nature of the ALU, a more robust verification process is warranted. Complete the test cases in Table 2 on the next page and include it in the RAT assignment report. Note that the ALU always has a "RESULT", "C" and "Z" value for all operations, as they are simply outputs. If the C or Z flags aren't affected by the instruction, the Control Unit (a separate module) will take place of not loading the C and Z outputs of the ALU into the flag registers.

Implement these test cases (yes all of them) in your simulation file and ensure that the hand calculated results from Table 2 match the results from the simulation. For easier comparison ensure the test conditions in the simulation are in the same order they appear in Table 2.

Hints

- Look at the elaborated design of the ALU to ensure there are no long chains of gates and small MUXs
- A properly designed ALU should contain 1 or 2 large MUXs. If there are more than 2 large MUXs in the elaborated design it can be further optimized.
- Ensure there are no warnings for latches

Function	SEL	Arguments			Expected			Simulation
		A	B	Cin	Result (hex)	C_Flag	Z_FLag	Time (ns)
ADD		0xAA	0xAA	0				
		0x0A	0xA0	1				
		0xFF	0x01	0				
ADDC		0xC8	0x36	1				
		0xC8	0x64	1				
SUB		0xC8	0x64	0				
		0xC8	0x64	1				
		0x64	0xC8	0				
SUBC		0xC8	0x64	0				
		0xC8	0x64	1				
		0x64	0xC8	0				
		0x64	0xC8	1				
CMP		0xAA	0xFF	0				
		0xFF	0xAA	0				
		0xAA	0xAA	0				
AND		0xAA	0xAA	0				
		0x03	0xAA	0				
OR		0xAA	0xAA	0				
		0x03	0xAA	0				
EXOR		0xAA	0xAA	0				
		0x03	0xAA	0				
TEST		0xAA	0xAA	0				
		0x55	0xAA	0				
LSL		0x01	0x12	0				
LSR		0x80	0x33	0				
		0x80	0x43	1				
ROL		0x01	0xAB	1				
		0xAA	0xF2	0				
ROR		0x80	0x3C	0				
		0x80	0x98	1				
ASR		0x80	0x81	0				
		0x40	0xB2	0				

`MOV`		`0x00`	`0x30`	`0`				
		`0x43`	`0x00`	`1`				

Table 2: ALU Test Cases for Verification

RAT Assignment 5 - Control Unit / RAT MCU

It's Alive!

Learning Objectives

- To understand the basic functionality and implementation (FSM) of a MCU control unit
- To understand the basics of fetch/execute instruction cycles
- To understand how to set control signals for a given instruction

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 RAT modules from the previous assignments into a working MCU. Though this MCU will be able to execute an actual RAT assembly language program, it will only use a limited subset of the available RAT instructions. This MCU will only be a subset of the final RAT MCU, as there will be 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 RAT modules, this assignment requires a systems-level approach to connect several core modules into a functional unit. Understanding the RAT at a systems level, including the interface to actual hardware, will help provide a complete understanding of the internal workings of the RAT and computers in general.

Control Unit

The control unit is a finite state machine (FSM) that provides control signals to the other RAT modules, ensuring everything functions properly. Recall that all of the RAT modules had control inputs. These inputs are the outputs of the control unit. The control unit provides the proper sequencing of modules operations in order to generate a working MCU.

1. Complete the control signal table (RAT_MCU_ControlSignal_Table spreadsheet on polylearn) for the following. If any signals are "don't care" for a given instruction / state, enter them as 0.
    a. States: INIT, FETCH
    b. Instructions: IN, MOV, EXOR, OUT, BRN
2. Transfer the signal values from the spreadsheet to the skeleton control unit created in class. To avoid latches, make sure every output signal from the control unit gets set for every case. The easiest way to assure this is to always initialize every signal to 0 and then only change those that matter based on the instruction or opcode.

C / Z Flag Registers

The C and Z flags can be implemented as two 1-bit registers. The C flag is identical to the Z flag with the added functionality of set and clear control inputs. These flags can be implemented as individual entities or a single entity with multiple process blocks. For now the simplified version of the flag file will be used (top left corner of the RAT architecture diagram) that does not include shadow flag registers. This design will be expanded or replaced with the more complex implementation incorporating the shadow flags in a later assignment.

RAT MCU

The RAT MCU will be implemented by connecting all of the existing modules as subcomponents. The completed MCU will consist of the entire RAT architecture diagram, but for this assignment only the Program Counter, ProgROM, Register File, ALU, Control Unit, and Flags will be used. Figure 1 shows the black box diagram for the RAT MCU module. This module will only include port maps of instances of modules already created and some MUXs for connecting signals.

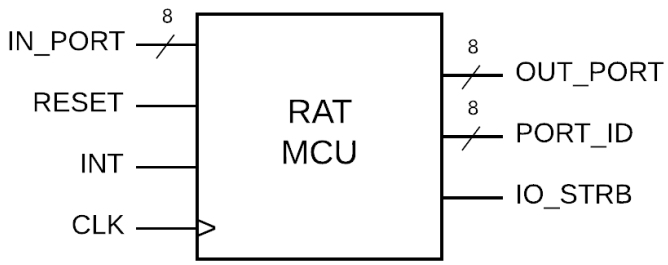


Figure 1: RAT MCU Black Box Diagram

ProgROM

Simulate the program shown in Code Segment 1 below with the RAT simulator until the program's functionality is understood. Add the assembler created prog_rom.mem file to the RAT MCU.

```
.EQU SWITCH_PORT = 0x20 ; port for switch input
.EQU LED_PORT    = 0x40 ; port for LED output

.CSEG
.ORG 0x10

main:   IN     r10,SWITCH_PORT
        MOV    r11,0xFF
        EXOR   r10,r11
        OUT    r10,LED_PORT
        BRN    main
```

Code Segment 1: Sample Program for RAT MCU Demonstration

Assignment

1. Implement and verify the C and Z flag registers
2. Implement the control unit for the specified instructions and states
3. Implement the RAT MCU by connecting the Program Counter, ProgROM, Register File, ALU, Flags, and Control Unit as shown in the RAT architecture diagram.
4. Verify the implemented RAT MCU by simulating with the given prog_rom. Be sure to test the RESET input. The INT input can be kept at 0 because the hardware has not be implemented to support that functionality yet. Show any pertinent internal signals in the MCU for verification purposes.

Deliverables

1. Black Box Diagram - RAT MCU
2. Behavioral Descriptions of
    a. Flags
    b. Control Unit
    c. RAT MCU
3. Structural Design of RAT MCU showing components
4. Verification
    a. Simulation of Flags
    b. Simulation of RAT MCU
5. SystemVerilog Code
    a. Flags
    b. Control Unit
    c. RAT MCU

RAT Assignment 6 - RAT MCU / RAT Wrapper

Rolling Now

Learning Objectives

- To understand the more functionality and implementation of the MCU control unit
- To understand how to set control signals for a given instruction
- To understand how to implement the RAT SoC (Wrapper) on the FPGA board
- 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 RAT MCU and add a RAT_WRAPPER to build a complete system that connects to the Basys3 development board. Though this MCU will be able to execute more instructions than the previous design, it will still use a subset of the RAT instruction set. This will allow a few modules to be left out to reduce the complexity of this assignment.

The Control Unit

Expand the control until to operate with all of the existing hardware implemented in the RAT MCU.

1. Complete the control signal table (RAT_MCU_ControlSignal_Table spreadsheet on polylearn) for all of the remaining instructions except those that use the Scratch RAM (blue) and the interrupt hardware (gray).
2. Transfer the signal values from the spreadsheet to the working control unit from the previous assignment.

The RAT Wrapper

The RAT_WRAPPER will connect the RAT_MCU with peripheral modules that interact with the Basys3 development board and include any necessary interface functionality. Input and output MUXs will connect the IN_PORT and OUT_PORT 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 RAT_Wrapper.sv file to understand how the MCU will interface to the Basys3 board. ***Note that the switch and LED port_id constants in the RAT wrapper must match those used in the assembly code.*** Add the RAT_Wrapper.sv file as the top module. Review any component names used to ensure the RAT MCU previous built is correctly instantiated in the RAT_WRAPPER. Write the necessary constraints file for connecting the RAT_Wrapper to the Basys3.

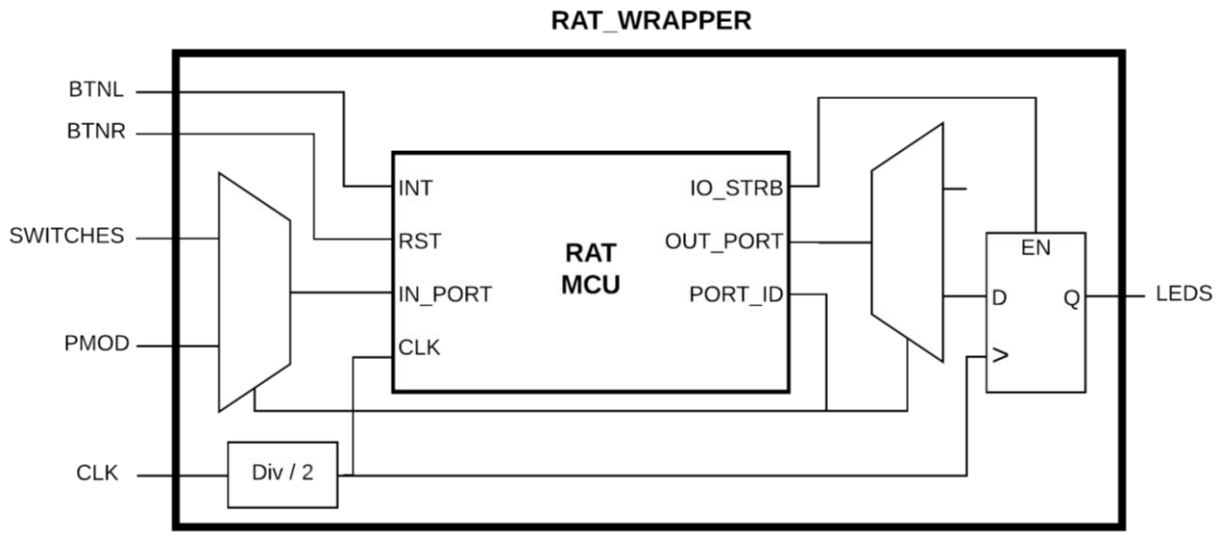


Figure 1: RAT_WRAPPER Block Diagram

Assignment

It is impossible to overstate the importance of ensuring that the RAT MCU is working properly on hardware in this assignment. To underscore this importance, there is a relatively large suite of test programs that must be used to test the RAT design. There are six test programs in phase 1. Some of these test programs are slightly different versions that test the same instructions, but each version is equally useful and important to verify a fully working hardware design.

There is brief description of each program provided in a spreadsheet on PolyLearn. These programs provide a visual test of each program. If the program passes the visual test on the Basys3, go onto the next program. Otherwise, it will be necessary to delve into the guts of the assembly language program to determine what instruction caused the hardware to fail the test. The Vivado Simulator is critical for properly debugging the design.

1. Verify the RAT is working for all of the implemented instructions by running all of the assembly programs on PolyLearn under Phase 1. If any of the programs fail to work, run the program under simulation and trace all of the internal signals to find the error.
2. Demonstrate all of the working phase 1 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 RAT needed fixing.

Debugging Tips

- Shorten the delay loop timings in the assembly programs before simulating
- 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
- Based on the LEDs that fail to light up, use the assembly program to identify where the error occurs, and then use the program code address to identify where in the simulation to begin checking with the program counter.

Deliverables

1. Black Box Diagram - RAT Wrapper
2. Behavioral Descriptions - RAT Wrapper
3. Structural Design of RAT Wrapper showing components
4. SystemVerilog Code - Updated Control Unit

RAT Assignment 7 - Stack Pointer / RAT MCU

Full Steam Ahead

Learning Objectives

- To understand how the RAT MCU uses the Control Unit to support the instructions that utilize the scratch RAM including stack-oriented RAT instructions
- To understand how stack-oriented instructions use the stack pointer to access the scratch RAM
- To understand how all non-interrupt oriented RAT instructions operate within the RAT MCU
- To understand how to add peripherals to the RAT Wrapper
- To learn how to debug and test a complex digital system

Stack

The stack is a first in, last out (FILO) memory buffer. Items are added to, or pushed onto, the top of the stack as shown in Figure 1. Items are removed from, or popped off, the top of the stack as shown in Figure 2. The stack is an ideal memory structure to save data and load it back in the reverse order.

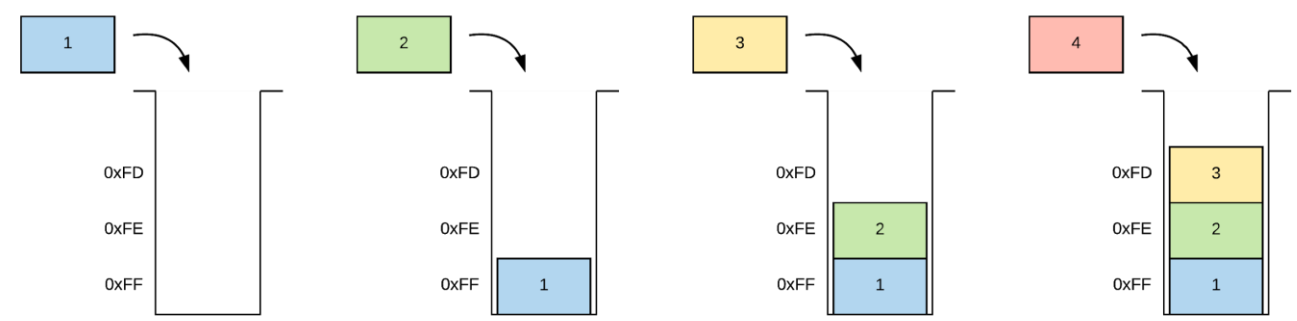


Figure 1: Example of Pushing onto the Stack

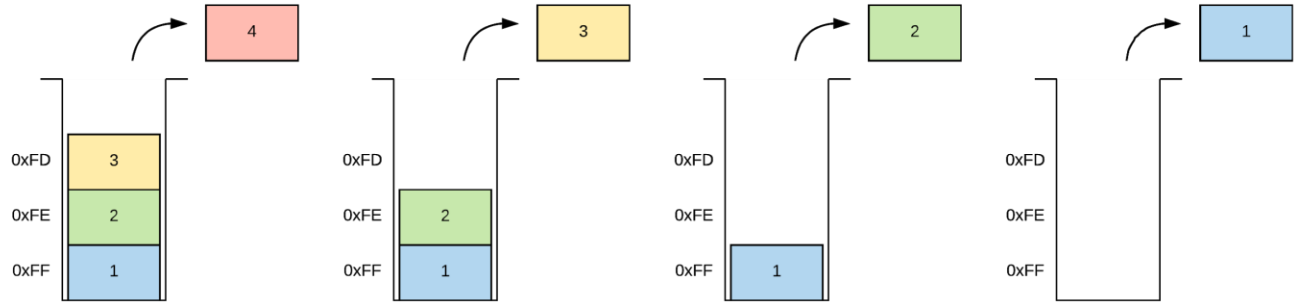


Figure 2: Example of Popping from the Stack

The advantage of using the stack from a programming perspective is the management of addresses and organizing the data in the memory modules is all handled in hardware by the MCU. While most MCU designs contain both the notion of a stack and a seperate generic memory to store data under program control, many MCUs such as the RAT MCU use the same physical memory device (Scratch RAM) to both implement stack-oriented instructions and provide generic data storage for the programmer. The stack is saved in the Scratch RAM starting at the bottom, or the last address of the Scratch, 256 (0xFF). As items are pushed onto the stack, the used data grows towards lower addresses. A completely full stack would occupy every address up to and including 0x00. Keeping track of where the stack was occupying memory in the Scratch RAM requires another hardware module, the Stack Pointer.

Stack Pointer

The Stack Pointer is a hardware module that keeps track of the address in the Scratch RAM that is being occupied by the top of the stack. The Stack Pointer is used as an address input into the Scratch RAM for pushing data onto and popping data off of the stack. It automatically keeps track of where the top of the stack is so the programmer does not have to keep track in the assembly code. As data is pushed on top of the stack, the address is decremented, and as data is popped off the top of the stack, the address is incremented. The stack pointer can also be reset or loaded with an arbitrary value. This functionality should be familiar because the Stack Pointer is a counter like the Program Counter. The Stack Pointer can be built like the Program Counter with the addition of a decrement function. All operations, including reset, of the stack pointer occur synchronously.

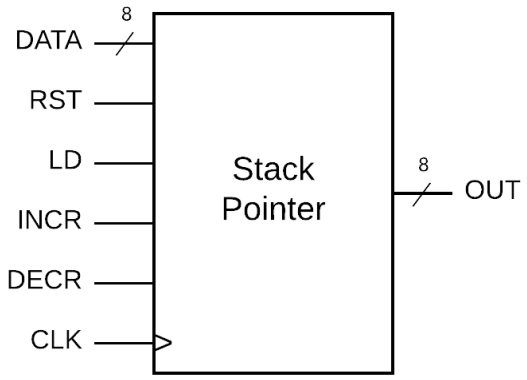


Figure 3: Black box diagram for Stack Pointer.

Scratch RAM Memory Uses

The Scratch RAM is used for the dual purpose of accessing data in arbitrary locations (ST and LD) and from the stack (PUSH and POP). While Scratch RAM saves data in 10-bit chunks, the following instructions only utilize the lower 8-bits of the data: ST, LD, PUSH, POP. These instructions read or write data directly from the instruction or a register. The stack is also used when making subroutine calls. The stack is used to save the current address in the Program Counter for a CALL instruction. The RET instruction will then load the Program Counter with the address saved from the stack. The Program Counter value is 10-bits, which necessitates the

10-bit size of the Scratch RAM. The Scratch RAM is used for the dual purpose of arbitrary data storage and the stack to optimize use without requiring unused hardware. It is the responsibility of the programmer to determine how the Scratch RAM memory is utilized and ensure no locations get overwritten.

When an instruction involving the stack is executed, the Stack Pointer provides the address to the Scratch RAM. An instruction that pushes data onto the stack, the address to save data too is 1 location above (less than) the current Stack Pointer value. The Stack Pointer is then decremented to point to the new top of the stack. If the current top of the stack is at address 0xF9, pushing data to the stack will go into address 0xF8 and the Stack Pointer will decrement to be 0xF8. Removing data from the stack reads data from the current address of the Stack Pointer. The Stack Pointer is then incremented to point to the new top of the stack at the next item. If the current top of the stack is at address 0xF5, popping data from the stack will read the data at address 0xF5. Then the Stack Pointer will be incremented to 0xF6. The difference between using the address of the Stack Pointer (pops) and the address above the Stack Pointer (pushes) is controlled with the MUX connected to the Scratch RAM address input in the RAT architecture. Notice one of the inputs includes a decrement (-1) that is used for pushing data to the stack.

Seven Segment Display:

The TestAll program will utilize the 7 segment display so a driver must be added to the RAT WRAPPER. A 7 segment display driver (sseg dec) is provided on PolyLearn. This driver module will be connected to the output MUX similarly to the LEDs. A signal must be added to connect the output MUX to the 7 segment driver. To keep consistent formatting, a new constant should be added for the 7 segment port id (0x81) similar to the LEDs. Other output drivers (ADC, speaker drivers, etc) can be connected in a similar fashion to expand the functionality of the RAT for the final project. Input drivers (keypad, keyboard, mouse, etc) would be connected to the input MUX in the same way.

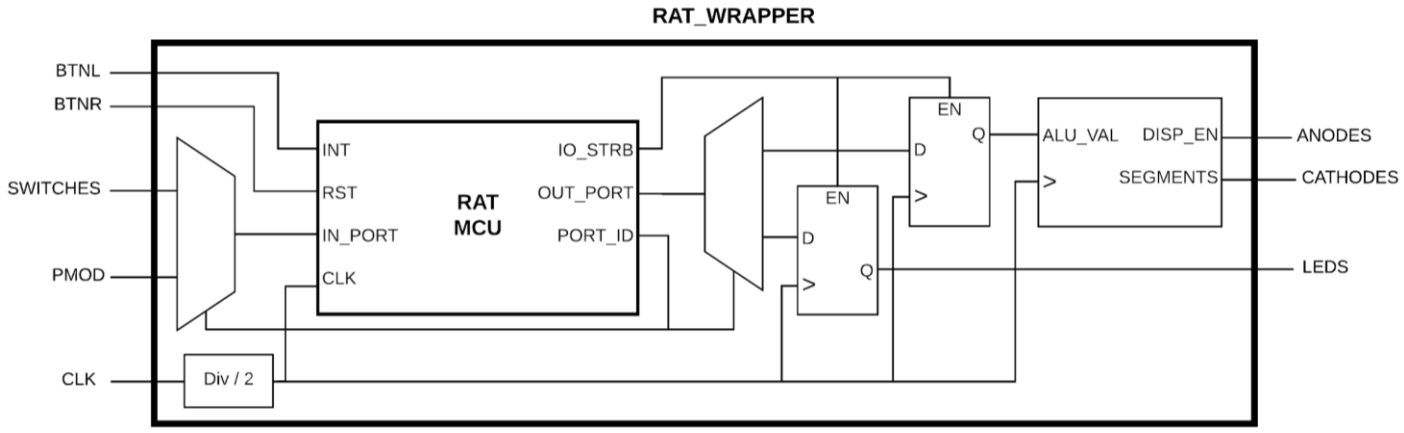


Figure 4: Connecting 7 Segment Display to the RAT MCU.

Assignment

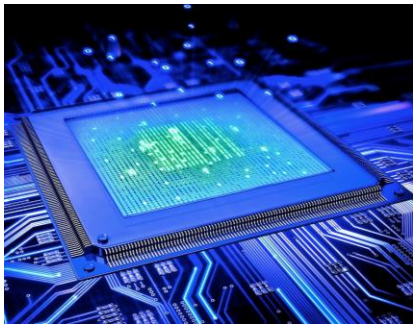
1. Implement the Stack Pointer
2. Add the stack pointer and Scratch RAM to the RAT MCU from the previous assignment. Implement any MUXes needed for fulling connecting the Stack Pointer and Scratch RAM.
3. Add to the Control Unit from the previous assignment to include all of the RAT instructions except those dealing with interrupts (SEI, CLI, RETID, & RETIE). These are highlighted in gray in the Control Unit spreadsheet from before.
4. Add the 7 segment driver to the RAT WRAPPER.
5. Verify the RAT is working for all of the implemented instructions by running all of the assembly programs on PolyLearn under Phase 2. If any of the programs fail to work, run the program under simulation, tracing all of the internal signals to find the error as before. A version of each test program from both phase 1 and phase 2 is concatenated into one assembly language program: testAll.asm. This program will run through 7 test cases, displaying which test case is currently running using the 7 segment display.
6. Demonstrate all of the phase 2 tests including testAll to an instructor or TA on your Basys3 board.

Deliverables

1. Black Box Diagram - Stack Pointer
2. Behavioral Descriptions - Stack Pointer
3. Structural Design - Stack Pointer
4. SystemVerilog Code
    a. Stack Pointer
    b. Updated Control Unit

RAT Assignment 8 - Interrupts and Button Bounce

Finish Line!



Learning Objectives

- To understand microcontroller interrupts, including the linkage between MCU hardware and the RAT MCU instruction set architecture
- To implement the interrupt architecture on the RAT MCU hardware
- To learn how to simulate interrupt operation on the RAT MCU
- To understand how to write and implement interrupt service routines (ISRs) on the RAT 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 programmatic 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 every few nanoseconds, the likelihood that the instruction to read the buttons occurs in the same few nanoseconds 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 in 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 whatever instruction it was processing when the interrupt occured 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 the 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 100 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 extended the time spent in the ISR.

RAT MCU Interrupt

Rather than implementing a complex interrupt handler, the RAT MCU can only handle a single external interrupt. The mechanism used to handle interrupts in the RAT MCU is referred to as a vectored interrupt. Once the RAT MCU receives an interrupt, program control transfers to a predetermined address in instruction memory (ProgROM) after execution of the current instruction completes. This address is known as a vector address, and should always contains a BRN instruction. The instruction branched to will be the starting address of the ISR. This allows the ISR code to be located anywhere in the ProgROM while allowing a hard wired interrupt vector address. The code in the ISR generally implements some task to appropriately handle the interrupt. When the MCU finishes executing the ISR, the MCU returns program control to fetch the instruction following the instruction that was being executed when the interrupt was detected.

When the RAT 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 executed). Preventing the RAT MCU from processing later occurring interrupts is referred to as interrupt masking. Remember, preventing the RAT MCU from acting on interrupts does not prevent the interrupt causing event from happening, and instead simply causes the RAT MCU to ignore any occurrence of an interrupt. Interrupts can also be manually masked or enabled during any program segment using the instructions SEI and CLI. SEI (set interrupt enable) allows or enables the RAT MCU to process interrupts, and CLI (clear interrupt enable) disables or masks interrupts.

Low-Level Details of Interrupt Handling

The RAT 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 not masked, when the RAT 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. The first step in the processing of the interrupt is to place the vector address in the program counter, which will cause the RAT MCU to fetch the instruction at the location of the vector address after completing the execution of the current instruction. The RAT MCU hard wires the vector address to 0x3FF. At the same time, the RAT MCU pushes the current address of the Program Counter, which is the location of next instruction that would have been executed had there not been an interrupt, onto the stack. In addition, the RAT MCU simultaneously "saves the current context" of the RAT MCU by saving the current values of the carry and zero flags into the "shadow" carry and zero flag registers. All of these saved values are necessary to return the MCU back to its current state after the ISR is completed.

Processing the ISR begins and continues until the RAT MCU encounters a RETIE or a RETID instruction. Execution of one these instructions causes the RAT MCU to pop an address off the stack and into the Program Counter. This should be the address that was pushed when the interrupt was detected which was the address of the next instruction that would have been executed had there not been an interrupt. In addition, the RAT MCU simultaneously "restores the context" of the processor at the time the interrupt was received by copying the contents of the shadow carry and zero flags back into the actual carry and zero flags.

General Sequence of a RAT MCU Interrupt

1. The RAT MCU detects an asserted signal on the interrupt input (assume the RAT MCU has not masked the interrupt)
2. Execution of the current instruction completes and the RAT MCU goes into an interrupt state
3. In the Interrupt state
    a. Mask the interrupt so the MCU will ignore further interrupts
    b. Current Program Counter (address of the next instruction, not the ISR) is stored on the stack
    c. The context of the MCU is saved by copying the zero and carry flags into shadow flags
    d. The Program Counter is loaded with 0x3FF
4. The Control Unit goes to a fetch state, fetching the instruction at 0xFF (This should be a branch instruction that directs program control to the address of the ISR
5. The instruction at location 0x3FF is executed (branching to ISR)
6. Execution of the ISR begins.
7. Execution of the ISR completes with a RETIE or RETID instruction
    a. The RAT MCU pops the saved program counter off of the stack and loads into the program counter. This would have been the next instruction to fetch when the interrupt occurred.
    b. The RAT MCU context is restored by copying the shadow carry and zero flags to the actual carry and zero flags
    c. Interrupts are masked (RETID) or unmasked (RETIE)
8. Execution resumes by fetching the instruction that was next when the interrupt occurred.

Modifying the RAT Control Unit for Interrupts

Interrupts on the RAT MCU are primary handled by the Control Unit. To implement interrupts the Control Unit must be modified to contains an extra state that handles the RAT's interrupt mechanism. The proper vernacular for this modification is to include an "interrupt cycle" into the current FSM. Figure 1 shows the state diagram for the Control Unit including the interrupt cycle. Note that entry into the interrupt state occurs when the RAT's INT input is asserted in the EXECUTE state. In the INTERRUPT state, the Control Unit asserts the signals to implement steps 3.a-d listed above. (Hint: Remember to include the INT input in the FSM sensitivity list)

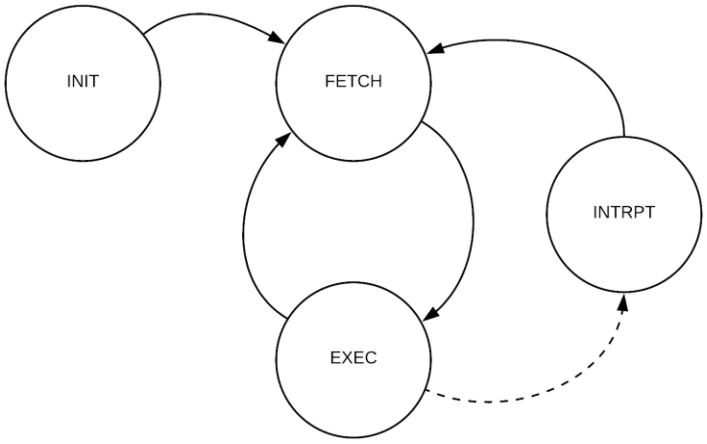


Figure 1: The RAT Control Unit State Diagram with Interrupts

Modifying the Flag Registers for Interrupts

Because interrupts can occur at any point in the running program, the context of the MCU must be saved. For example, if the interrupt occurred between a CMP and BRNE instruction, the ISR could potentially overwrite the Z flag. When the MCU returns from the ISR to continue execution at the BRNE instruction, the MCU needs to have the Z flag value that was set from the CMP function. The context of the MCU is determined by the Z and C flags. Those flags can be overwritten during an ISR, so they must be saved before the ISR is started and restored after the ISR is completed. To achieve this, shadow flags will be added to the flag file. When an interrupt occurs, the flags will be backed up to the shadow flags before starting the ISR. After the ISR completes the shadow flags will be used to reload the Z and C flags. This will ensure the context of the MCU will be the same as if the interrupt never occurred. The current implementation of the FLAGS module will be expanded as shown in Figure 2 below.

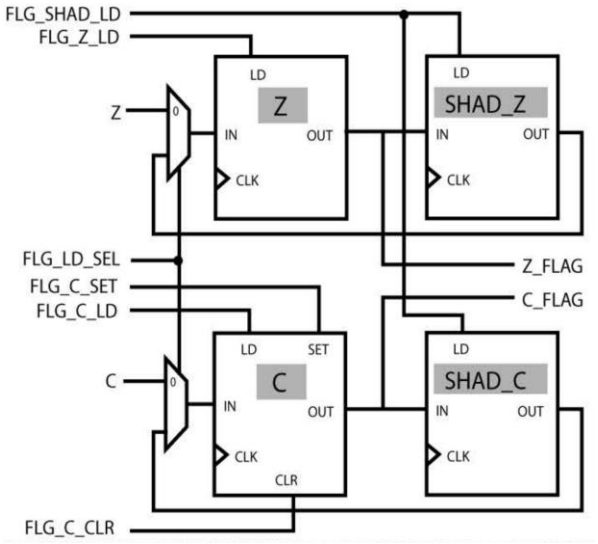


Figure 2: FLAG Module with Shadow Flags.

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 the MCU entering into an interrupt cycle.
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. Mechanical contacts create a bounce situation that is described below.

The ideal interrupt pulse length for the RAT MCU should be between two and three clock cycles to ensure proper acknowledgement and processing. This equates to 40 - 60 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 operate properly. Figure 3 shows some examples of a bouncing switch (source:https://softsolder.com/2012/07/13/contact-bounce-why-capacitors-dont-fix-it/).

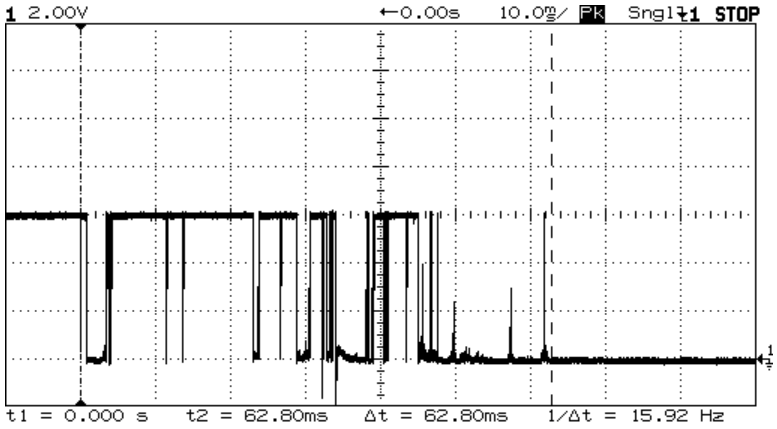


Figure 3a: Example of Button Bounce

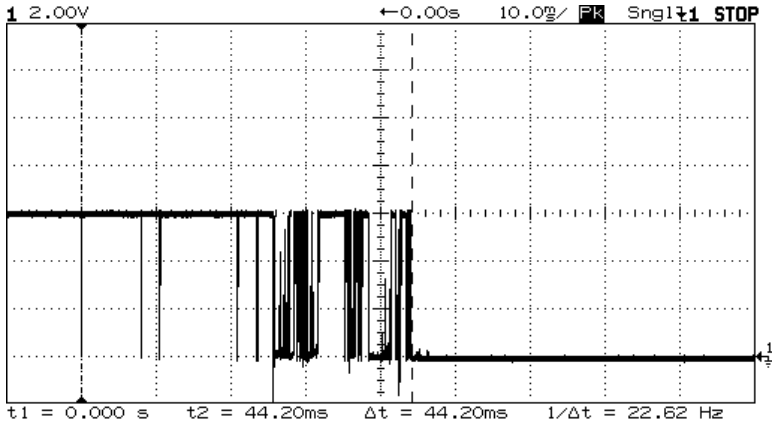


Figure 3b: Example of Button Bounce

Approaches to handling switch bounce generally fall into two categories: hardware solutions or software solutions. The preferred solution for the RAT 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 always be avoided. 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 60 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 in Interrupt Signals). The one-shot creates a single short pulse after the button is released. The debouncer one-shot must be connected to the RAT MCU interrupt signal and button input inside the RAT WRAPPER.

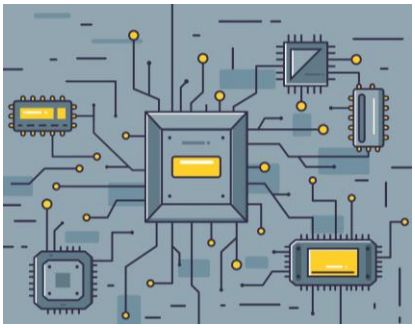
Assignment

1. Add the carry and zero shadow flags and the interrupt hardware to the RAT MCU. Consult the RAT architecture diagram and RAT Assembler Manual for further details.
2. Modify the RAT MCU such that it can process interrupts and execute the following RAT assembly instructions: RETIE, RETID, SEI, and CLI. (These are highlighted in gray in the Control Unit spreadsheet) This step includes Control Unit modifications for adding the interrupt state to the FSM as well as implementing these instructions in the Execute state.
3. Modify the RAT Wrapper to connect the left-most button on the development board to the input of the provided debouncer one-shot. The output of the debouncer one-shot connects to the interrupt input of the RAT MCU.
4. Test the results of the previous two steps using the test program in Phase 3.
5. Demonstrate the working test program to your instructor or TA.

Deliverables

1. Black Box Diagram - Updated Flag File
2. Behavioral Descriptions - Updated Flag File
3. Structural Design
    a. Updated Flag File
    b. Updated RAT MCU
    c. Updated RAT Wrapper
4. SystemVerilog Code
    a. Updated Flag File
    b. Updated RAT MCU

RAT Assignment 9 - 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 RAT as the core of a digital device. The project needs to use an assembly language program that runs on the RAT 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 peripheral, e.g. the VGA, the mouse, the keyboard. 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.

Final Report

The final report will be a very different document from the previous lab reports. The final report will be a technical document 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?

Operation Manual

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? What is displayed and where? How is the device used? Refer to the example Operational Manuals on PolyLearn.

Peripheral Details (only if new)

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. 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 this peripheral could be purchased as a physical device, what specifications would be listed in its data sheet? If someone wanted to use this peripheral, what would be needed to know to connect this part and use it correctly?

External Circuit Peripheral (only if new)

If the device used any external connections or circuitry for a peripheral include a schematic of how 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.

Appendix

1. Full assembly code listing with comments
2. Peripheral SystemVerilog code listing (only if new)

Software Assignments

Software Assignment Report Format

Behavior Description **(10)**

Describe the behavior of the designed program. This should be a short synopsis that explains in your own words how the program functions. For simple, straight forward programs, this may be just what it does. For less intuitive programs it should not be just what it does, but how it does it.

Flow Chart **(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. The flow chart can be hand drawn as long as it is legible. Any grading mistakes due to illegible or confusing drawings are solely the responsibility of the student. Recommend software for creating a flow chart: Visio (free at calpoly.onthehub.com), Lucidchart (free with Cal Poly Office365), Draw.io, or Dia.

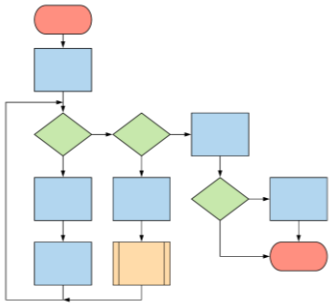
Verification by Simulation Results **(30)**

No simulation or timing diagram is visible with the RAT simulator so create a table with the test cases that were used by the RAT 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. For the test cases used, give some explanation on why those specific cases were chosen.

Assembly Source Code **(30)**

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. The source code should be formatted with a fixed width font (Courier or Monospace). Never use screen shots of code. Black text is easier to read than pixelated fonts no matter how colorful and pretty it looks.

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.

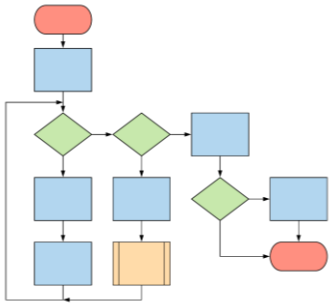
Assignment:

First create a flowchart that will perform the following behavior. Then implement the flowchart with RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Read 3 inputs from port id 0x30, add them together, and output the result to port id 0x40. Assume the input values are 8-bit unsigned values.

2. Read an input from port id 0x30. Assume the input is an 8-bit signed value in 2's complement (RC). Change the sign of the input and output the result to port id 0x40. The result should also be an 8-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.

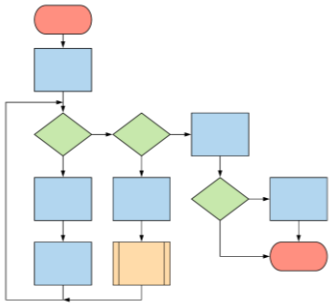
Assignment:

First create a flowchart that will perform the following behavior. Then implement the flowchart with RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Read an 8-bit unsigned value input from port id 0x30. If the input is greater than or equal to 128, the value is divided by 4. You can ignore any remainder. If the value is less than 128, the value is multiplied by 2. The result should be output to port id 0x42.

2. Read an 8-bit unsigned value input from port id 0x30. If the input value is a multiple of 4, all of the bits should be inverted, otherwise if the input value is odd, add 17 and divide the result by 2, otherwise subtract 1 from the value. The result should be output to port id 0x42

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.

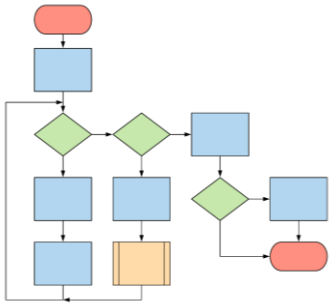
Assignment:

First create a flowchart that will perform the following behavior. Then implement the flowchart with RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Read a value from port_id 0x9A. Assume the 8-bit value is the combination of two 4-bit unsigned values. Divide the 8-bit input into two 4-bit values (the upper and lower 4-bits). Assume both of the 4-bit values are unsigned. Multiply the two 4-bit values together and output the result to port_id 0x42. Optimize your code for size.

2. Read a value from port_id 0x9A, delay for 0.5 seconds, and then output the value to port_id 0x42. The RAT MCU operates at 25 MHz so each instruction takes 40 ns to run. (*Hint: Use nested loops.*) There is no method to verify the run time in the RAT Assembler, so you will need to verify your timing by showing your calculation for the number of instructions being performed between IN and OUT.

Software Assignment 4 - Division



Learning Objectives:

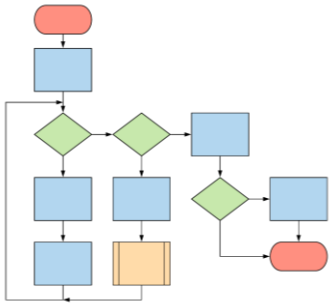
- To understand how to flowchart binary division operations.
- To understand how to perform binary division in assembly language programming.
- To gain experience using an assembly language simulator to analyze an assembly language program.

Assignment:

First create a flowchart that will perform the following behavior. Then implement the flowchart with RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Read a value from port_id 0x9A. Assume the input is an 8-bit unsigned value. Divide the input by 3 and output the quotient (result) to port_id 0x42 and the remainder to port_id 0x43.
2. Read two 8-bit unsigned values from port_id 0x9A. Divide the first value by the 2nd and output the quotient (result) to port_id 0x42 and the remainder to port_id 0x43.

Software Assignment 5 - 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.

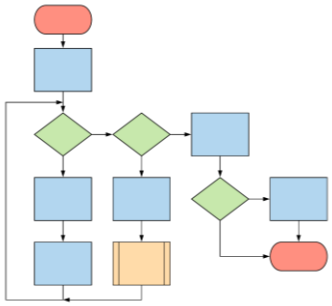
Assignment:

First create a flowchart that will perform the following behavior. Then implement the flowchart with RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Create an array in the data segment that contains the first 14 values of Fibonacci sequence (starting with 0 and ending with 233). 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 port_id 0x42.

2. Create an array of 10 values. Read 10 values from port_id 0x9A 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 port_id 0x42 in order least to greatest.

Software Assignment 6 - The Stack



Learning Objectives:

- 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 an assembly language simulator to analyze an assembly language program.

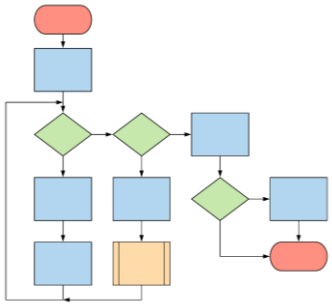
Assignment:

First create a flowchart that will perform the following behavior. Then implement the flowchart with RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Read a sequence of values from port_id 0x9A. Each value is added to the stack until the value 0xFF is read. When 0xFF is input the program will not add it to the stack, but instead will output all of the saved values to port_id 0x42 in the reverse order they were input. Specify the limits of your program.

2. Read a sequence of values from port_id 0x9A. Each value is added to the stack until the value 0xFF is read. When 0xFF is input the program will not add it to the stack, but instead will output all of the saved values to port_id 0x42 in the same order they were input. Specify the limits of your program. (Hint: You do not have to use POP to “read” from the stack)

Software Assignment 7 - 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.

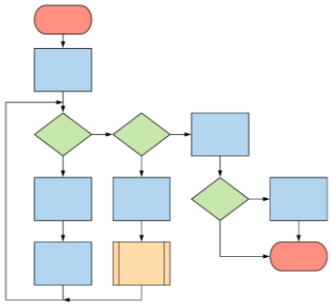
Assignment:

First create a flowchart that will perform the following behavior. Then implement the flowchart with RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Create a subroutine that will divide a value by 10. Then use the subroutine to convert an unsigned 8-bit value into a 3 digit binary coded decimal. The main program should read an 8-bit unsigned value from port_id 0x9A. The value will be converted into a BCD (binary coded decimal) of 3 values for 100s, 10s, and 1s decimal place value. For example, the value 0xAF (175 decimal) should be split into 3 binary values for 1, 7, and 5. The 100's value should be output to port_id 0x41, the 10's value should be output to port_id 0x42, and the 1's value should be output to port_id 0x43.

2. Create a subroutine that adds 2 8-bit registers. Any overflow from the addition is added to a separate register from the result. Use the subroutine in a program that reads 2 8-bit unsigned values from port_id 0x9A and multiplies them together, splitting the 16-bit result in 2 8-bit registers. The high 8-bits should be output to port_id 0x41 and the lower 8-bits should be output to 0x42.

Software Assignment 8 - 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 RAT MCU assembly language. Make sure your code is in proper form including comments. Use the RAT MCU Simulator to ensure you program performs as desired.

1. Write an interrupt driven program that will turn the LEDs (port_id 0x42) on or off. Every interrupt will alternately turns on/off the LEDs. The LEDs that change (toggle) are the ones corresponding to the value on the switches (port_id 0x9A). 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). Minimize the number of instructions you use in your neatly written solution.

   Example Case: LEDS: 1010 1100
   Interrupt occurs and SWITCHES: 0110 1001 -> LEDS: 1**10**0 **0**10**1**

2. Write an interrupt driven program that toggles the LED (port_id 0x42) based on the switches (port_id 0x9A) when an interrupt occurs just like problem 1 above. However, if on two consecutive interrupts the switches have the same value, the program will stop outputting to the LEDs until BUTTON(0) (bit 0 on port_id 0x9B) is pressed. 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 appear on more than two consecutive interrupts.

Peripheral Assignments

Part Kit for the Peripheral Assignments

Part Name	Quantity	Documentation Datasheet
12 Button Keypad	1	https://cdn.sparkfun.com/assets/7/e/f/6/f/sparkfun_keypad.pdf
Mini Speaker	1	https://www.cui.com/product/resource/cem-1203-42-.pdf
2N4401 BJT	1	https://www.fairchildsemi.com/datasheets/2N/2N4401.pdf
220 Ω (⅛ w)	1	
1 kΩ (⅛ w)	3	
1.6 kΩ (⅛ w)	1	
100 nF	1	
7 pin header	1	For connecting the keypad

Peripheral Assignment Report Format

Black Box Block Diagram **(5)**

Include a block diagram of the designed peripheral(s). Show all inputs and outputs, specifying bus widths where applicable.

Behavior Description **(10)**

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

Structural Design **(5)**

Include an image from Vivado of the RTL elaborated design schematic. Use this image to show the design is built effectively. Efficient designs should not include long chains of gates due to timing and propagation delay.

Specification **(20)**

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 you were to buy this peripheral as a physical device, what specifications would be listed in its data sheet? If someone wanted to use this peripheral, what would be needed to know to connect this part and use it correctly?

SystemVerilog Source Code **(20)**

Provide the SystemVerilog source code for the peripheral(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, it can be easily highlighted with https://emn178.github.io/online-tools/syntax_highlight.html Select Verilog language and the Xcode style.

Example Use Code **(20)**

Write an example RAT assembly program(s) using this peripheral to show how it can be used. This code should be properly commented to explain what the code is doing and how it is utilizing the designed peripheral peripheral. This code should be sufficient for a fellow student to be able to reuse this peripheral in their own MCU.

Demonstration **(20)**

Demonstrate the working peripheral in class to the instructor or TA.

Peripheral Assignment 1 - Speaker Driver

General Design Notes: Speaker Driver

A speaker requires an oscillating signal to create a sound. The speaker cone needs to vibrate by being driven high and low. This vibration will cause the air to move creating a sound wave. The frequency of this vibration is what determines what tone or note is played by the speaker. The human ear can typically hear in the 20 Hz to 20 kHz range.

A microcontroller can create these frequencies in a speaker by driving it with a 50% duty cycle square wave. The speaker used for this project is not designed to play sound for the full range of human hearing, but it is optimized for a range around 2 kHz. For this reason we will limit the frequency range for the driver to 3 octaves, going from 1046 Hz to 7902 Hz. Table 2 below shows musical notes and the equivalent frequency for the 3 octave range. Each of these notes can be given a designated value to correspond to each note. The value of 0 can be added to specify no sound.

Amplifier Circuit

The digital output pins from a microcontroller are typically limited to 10 - 20 mA. That may not be enough current to drive a speaker, so an amplifier circuit is needed to interface the speaker with the digital output. This can be made easily with a resistor and transistor as shown in Figure 1 below.

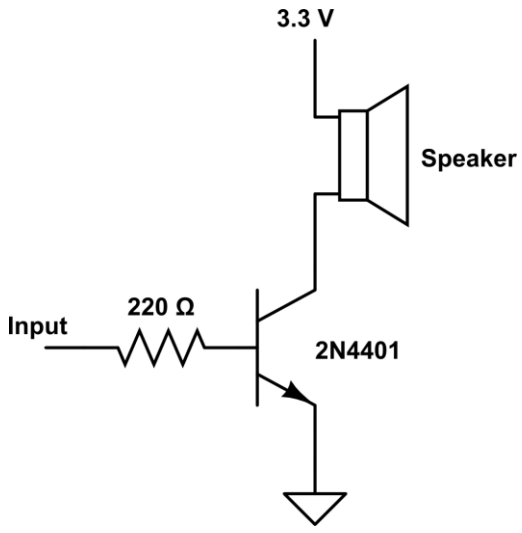


Figure 1: Speaker Driver Circuit

Input Value	Note	Octave	Frequency (Hz)
0	none	none	0
1	C	6	1046.502
2	C# / Db	6	1108.731
3	D	6	1174.659
4	D# / Eb	6	1244.508
5	E	6	1318.51
6	F	6	1396.913
7	F# / Gb	6	1479.978
8	G	6	1567.982
9	G# / Ab	6	1661.219
10	A	6	1760
11	A# / Bb	6	1864.655
12	B	6	1975.533
13	C	7	2093.005
14	C# / Db	7	2217.461
15	D	7	2349.318
16	D# / Eb	7	2489.016
17	E	7	2637.021
18	F	7	2793.826
19	F# / Gb	7	2959.955
20	G	7	3135.964
21	G# / Ab	7	3322.438
22	A	7	3520
23	A# / Bb	7	3729.31
24	B	7	3951.066
25	C	8	4186.009
26	C# / Db	8	4434.922
27	D	8	4698.636
28	D# / Eb	8	4978.032
29	E	8	5274.042
30	F	8	5587.652
31	F# / Gb	8	5919.91
32	G	8	6271.928
33	G# / Ab	8	6644.876
34	A	8	7040
35	A# / Bb	8	7458.62
36	B	8	7902.132

Table 1: Musical Notes and Frequencies

Basys3 PMOD

The Basys3 allows multiple external connections via the 4 PMOD connectors labeled JA, JB, JC, and JXADC. Each 12 pin PMOD has VCC (3.3V) on 2 pins, GND (0V) on 2 pins, and 8 pins for digital signals. The JXADC connector is wired differently than the other 3 with a partially configured filter for the Atrix7 ADC inputs. This results in a decreased speed performance for digital signals using this connection. Because of this the JXADC connection should not be used for this project. The pin numbering (0 - 11) for each PMOD connection is defined as shown in Figure 2 below. The pin assignments needed for the constraints (XDC) file for each PMOD pin can be found in the Basys3 reference manual.

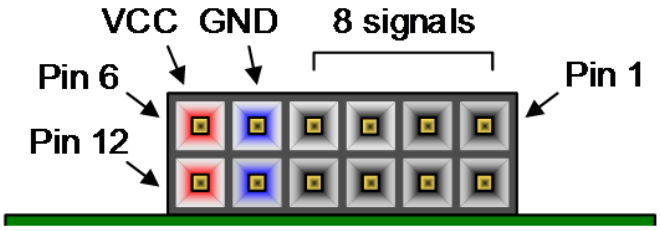


Figure 2:PMOD Pin Configuration (from the Basys3 reference manual)

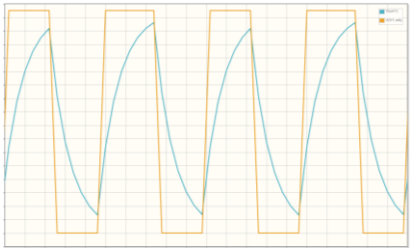
Demonstration

Use a breadboard (not provided) to build the speaker circuit and connect it to the Basys3. Demonstrate the working speaker at various notes from Table 1 above. Also connect the oscilloscope to the output of the Basys3 PMOD (the input to the speaker circuit) to verify the correct frequency is being generated.

Assignment

1. Create a speaker driver module that uses an 8-bit input and system clock to create a square wave that corresponds to each of the frequencies in Table 1. Use the oscilloscope to verify the frequencies are accurate.
2. Connect 8 switches to form the 8-bit input and any PMOD signal to the output of the speaker driver module.
3. Build the amplifier circuit to the connected PMOD signal.
4. Demonstrate the speaker driver to a TA or instructor by connecting the output from the PMOD pin (before the resistor and transistor) to the oscilloscope to verify the frequency.

Peripheral Assignment 2 - Digital to Analog Converter using PWM and an RC Filter



General Design Notes: Digital to Analog Converter (DAC)

Microcontrollers interface with the multiple types of peripherals, some of which are not digital devices. Many microcontrollers include output peripherals that are able to convert a digital value into an analog voltage. When such a peripheral is not available a simple DAC can be created using a pulse width modulated signal and an RC low pass filter. This component will use an 8-bit value digital value to specify the analog voltage to output. This will allow 256 different analog voltages, including 0 V, to be output with a resolution of 3.3 V / 255 = 0.013 V.

Pulse Width Modulation (PWM)

Pulse width modulation describes a modulation technique used to vary square waves. A PWM signal varies the width of the high or low of a square wave without changing the frequency of the pulses. This changes the signals duty cycle, varying from 0% (always low) to 100% (always high). The resolution of the PWM is determined by how many different duty cycles can be created from a digital value. An 8-bit value can define 256 different values, signifying a 256 different duty cycles from 0% (0) to 100% (255). The Basys3 has a system clock of 100 MHz. The highest frequency PWM signal that will allow 256 variable widths (including 0 width) is 100 MHz / 255 = 392.157 kHz. This will be the frequency for the PWM signal used in this component.

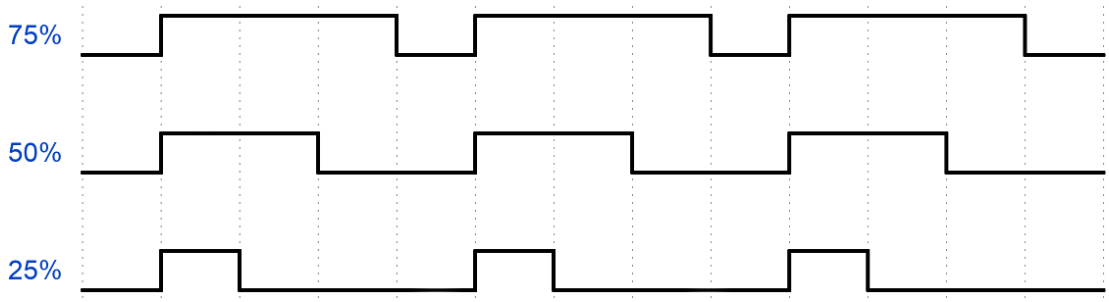


Figure 1 : Examples of Pulse Width Modulation

Low Pass Filter

Passing a square wave through a low pass filter RC circuit will cause the capacitor to charge and discharge on the high and low cycles. The time constant of the RC circuit will determine how quickly the capacitor will charge and discharge. A small time constant in respect to the frequency of the PWM will create a ripple on the output as the capacitor charges and discharges quickly. A larger time constant will keep the output stable, but will not allow the output voltage to respond quickly to changes in the PWM duty cycle. To achieve a reasonable

ripple and response time, a time constant of 160 μs can be used. This will give a maximum ripple of 0.013 V with a worst case scenario of 50% duty cycle. This circuit will have a settling time of 0.366 ms to reach 90% of the final voltage which means that the output analog signal can only be changed every 0.366 ms without creating distortion. The time constant for an RC circuit is given as $\tau = RC$, so the desired time constant can be achieved with a resistor of 1.6 kΩ and capacitor of 100 nF.

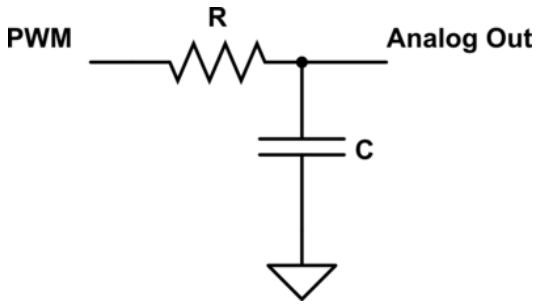


Figure 2: RC Low Pass Filter

To implement this component on the Basys3 board the RC circuit can be connected to a PMOD connector, using a logic signal for the PWM and ground. The logic signal on the Basys3 is 0V for logic 0 and 3.3 V for logic 1. This will give give an output range of the analog signal between 0 and 3.3 V. Using an 8-bit value to specify the 3.3 V range gives a resolution of 3.3 / 255 = 0.013 V. Notice this resolution matches the maximum ripple to assure the 8-bit value precision is not greater than the RC filter is capable of producing.

Demonstration

On the same oscilloscope, use 2 probes to show the PWM signal from the Basys3 and the Analog out from the RC circuit. Demonstrate outputs for low (0000 0000), high (1111 1111), and mid (1000 0000) inputs.

RAT Sample Code

When writing the sample code for this assignment you can assume you have a delay subroutine. You will need to comment how long the delay function is or how parameters can be passed to the subroutine to adjust the delay length. The delay subroutine can be executed or called with the CALL instruction as **`CALL delay`**

Assignment

1. Create a PWM module that uses an 8-bit input and the system clock to create a 392.157 kHz PWM signal that varies from 0% duty cycle to 100% duty cycle.
2. Connect 8 switches to form the 8-bit input and any PMOD signal to the output of the PWM module.
3. Build the RC filter to the connected PMOD signal and verify the DAC works properly with the bench oscilloscope.

4. Demonstrate the DAC to a TA or instructor.

Peripheral Assignment 3 - Keypad Driver

General Design Notes: Keypad Buttons

The keypad is a completely passive device which means that it does not create output signals that are 1 or 0 (high or low voltages). The keypad is made up of 12 buttons, but to reduce the number of wires and connections, the buttons are connected in a matrix of rows and columns. Pressing a button the keypad does not drive an output signal to a high or low voltage like the typical switches or buttons on the Basys3. Instead it makes a connection between a row and column wire. By determining which row and column is connected, the specific button press can be determined. This is done by controlling one dimension (ie Rows) while reading the other (ie Columns). A simple control system to perform this task is an ideal application for a custom digital device on the FPGA. This will significantly reduce the software coding needed to utilize the keypad as in input device on the RAT MCU.

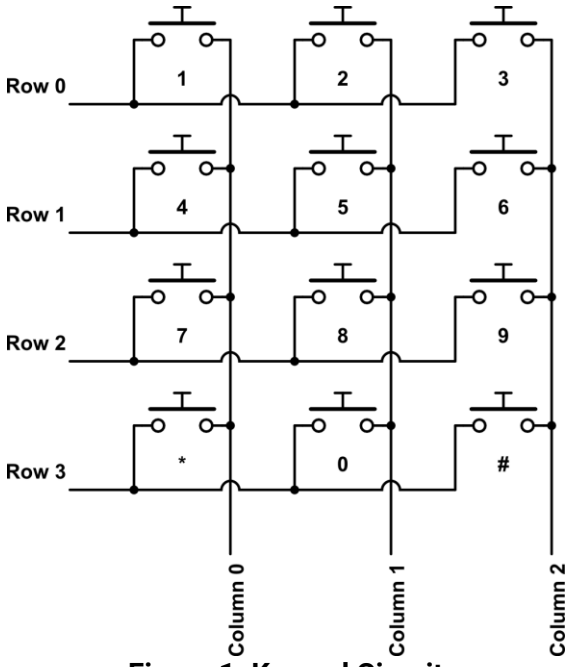


Figure 1: Keypad Circuit

Keypad Button Detection

Determining the button press on a keypad is performed by detecting a connection between the row and column wires. This determination can be made by selectively cycling one set of wires, for example rows, high and simultaneously reading the other set, columns in this case. If button 8 is pressed, setting Row 0 high while

setting the others low and checking the Columns results in reading all 0s. Cycling the Rows to set Row 1 high results in the same all 0s for the Columns. Only by setting Row 2 high will Column 1 go high signifying a connection at button 8.
The keypad driver can cycle one set of wires, either rows or columns, while reading from the other, columns or rows respectively. The driver can start by setting Row 0 (or Column 0) high while setting the rest low and read the Columns (or Rows). If the read wires are all 0, the driver can cycle to Row 1 (or Column 1) and read from the Columns (or Rows) again. The driver will continue to cycle rows (or columns) as long as the input from the columns (or rows) is 0. Once a button is pressed is pressed and the corresponding row (or column) is set high so that the input is non-zero, the driver can determine which button is pressed.

Keypad Pinout

The pinout for the keypad is not intuitive in that neither the rows or columns are grouped together. The order of the rows and columns is not sequential either. Figure 2 below shows how the rows and columns are connected to the bottom pinout. Note the first and last through hole is not connected to anything on the keypad. The middle seven are connected to the individual rows and columns as specified.

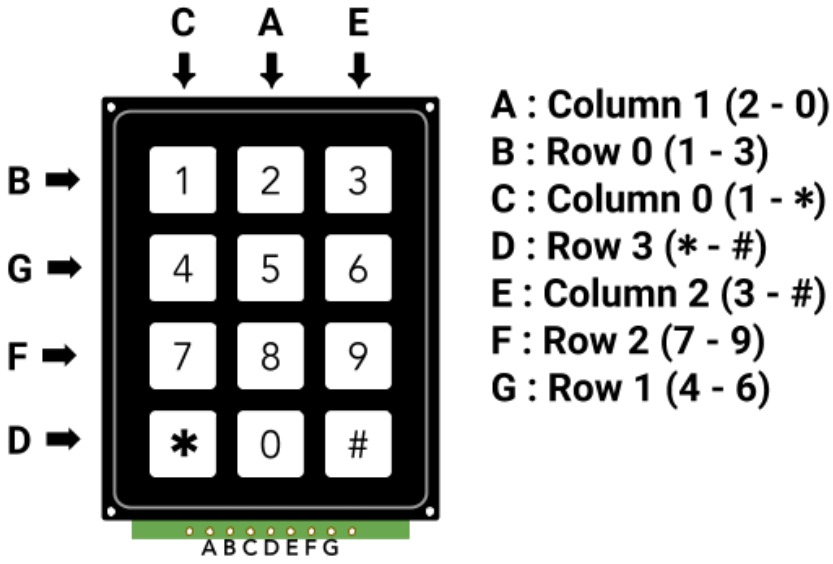


Figure 2: Keypad Pinout

Keypad Connection to PMOD

When no key is pressed on the keypad, the connections between the rows and columns are open. When his occurs, the input signals being read from the columns (or rows) will not be connected to a low or high voltage. This means the input signal will float. Floating signals can create havoc on digital circuits because when the inputs on a digital signal float they may be read as low (0) or high (1). This will create unpredictable behavior in the keypad driver when no keys are pressed. To keep the inputs from floating, the input signals can be forced to go high or low rather than float. This is accomplished by adding resistors that pull the signal up or pull the

signal down. These resistors are called pull up or pull down resistors. When using pull up resistors, the button connection would need to drive the input low. When pull down resistors are used, the button connection would need to drive the input high. In the example behavior described above, the button connections would drive the input high, so pull down resistors will be needed. The resistors are added as shown below in Figure 3.

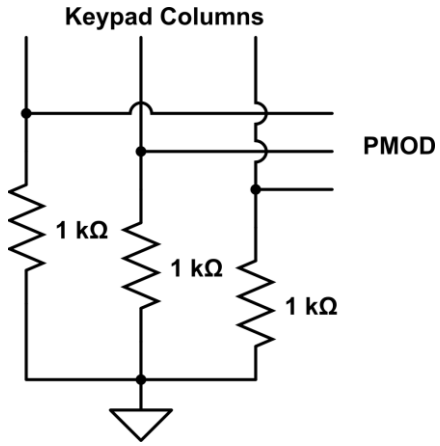


Figure 3: Keypad Pull Down Resistors

Interrupt Signal

To better utilize the keypad with an MCU, the keypad driver can create an interrupt signal whenever a new keypress is detected. The interrupt signal will notify the MCU that a button press has occurred so that it can read the input and handle the keypress accordingly. This avoids the need for the MCU to continuously poll or read from the keypad driver when no key is pressed. The interrupt signal should be a single pulse of 60 ns to trigger only a single interrupt. The interrupt signal should not be repeated until another key is pressed. You can assume that the first button must be released before another key is pressed so that 2 keys will not be pressed at the same time.

The interrupt signal could be created as a separate logic block (always block) that creates the 60ns pulse when it receives a trigger input. This can simplify the FSM for controlling the rows and columns and detecting button presses. The interrupt pulse generator can be implemented as a one shot to only 1 pulse per button press.

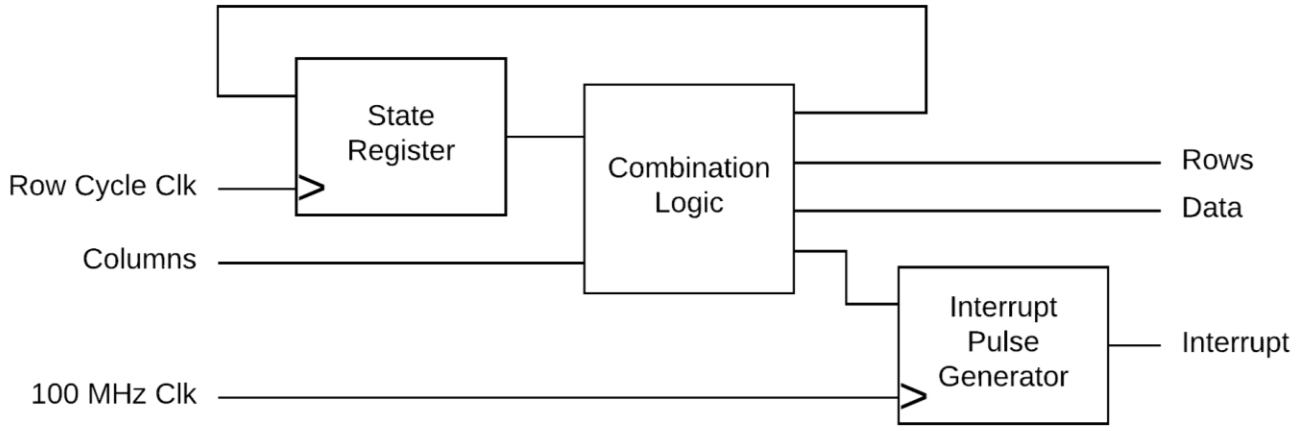


Figure 4: Keypad Driver Architecture

Demonstration

Demonstrate the working keypad by connecting the output of your keypad driver to a 7 segment digit as shown below in Figure 5. You can reuse any 7 segment driver you may have built in CPE 133. The * key can display A and the # key display b. When no key is pressed, the digit should be blank. The interrupt signal will be demonstrated on the oscilloscope in single run mode. (Hint: Soldering the header pins from your kit to the keypad will make it easier to connect and use)

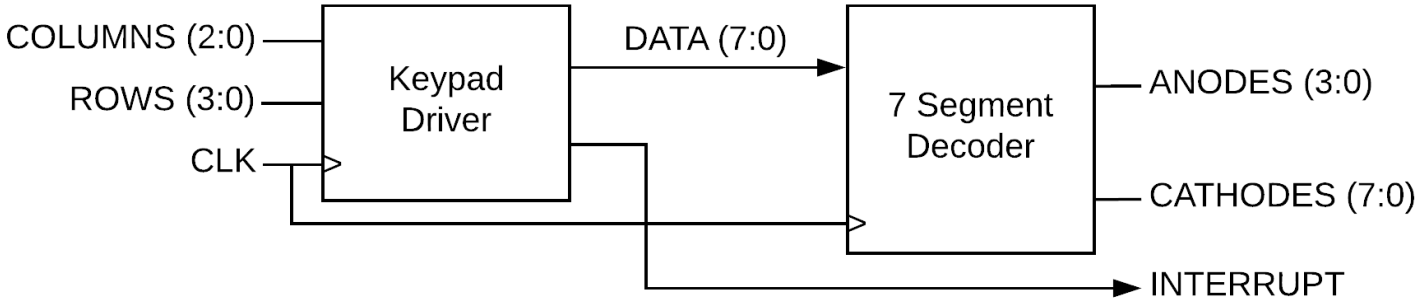


Figure 5: Keypad Driver and 7 Segment Decoder

Assignment

1. Create a keypad driver that is able to detect a key press and output the key value as an 8-bit signal.
2. Connect the keypad driver to a 7 segment decoder (you should have one from 133) to display the key press value. This should only display a value while a key is being pressed. When the button is released, the 7 segment display should go blank. (Hint: You may need to modify the 7 segment decoder to display nothing)

3. Connect the keypad to a PMOD connector, connect the interrupt output to the oscilloscope, and demonstrate the keypad driver to a TA or instructor.

Suggestions and Hints

- Implement an FSM by first drawing a state diagram.
- View a keypress on the oscilloscope to get an estimate for the length of a time of a keypress. You may want to try a few times to see how quickly you can create a single button press.
- Use the button press timing from the oscilloscope to determine a sufficient speed to scan the rows or columns (not both). (Hint: It is slower than 25 MHz. OK, so maybe that wasn't much of a hint.)
- Cycling the rows or columns too quickly will cause issues. It is quicker to spend the time looking at a button press on the oscilloscope than to try to debug high frequency noise in the keypad wires or randomly guess timing values.