FreeRange Computer Design:

The RAT MCU
Lab Activity Manual

Image used by permission from David Schoonover

Table of Contents

Acknowledgements

The Free Range Computer Design textbook outlines the complete history of the RAT MCU-based course, but some of the highlights are worthy of mention here. While good ideas are a dime-a-dozen, it takes an inspired person to actually make good things happen, particularly in academia. While the original idea/approach for this version of CPE 233 was Bryan's, Jeff Gerfen made this course happen by being the first to teach the course in the Winter 2010 quarter, including the lab portion of the course. Jeff took the idea and gave it legs. Specifically, Jeff is responsible for the initial creation/offering of the Computer Design course as well as the design of its original set of experiments.

Lab Submission Guidelines

This document contains the guidelines of how your submitted lab work ("lab work" is different from a lab report) documentation should appear. Adhering to these guidelines strongly indicate that you put the time and effort into understanding the lab material. Moreover, properly formatted submissions are easier to verify that your work is correct. Because the EE Department bean counters continue to act unfairly by overcrowding the digital labs, you need to submit quality reports so that they are easier to read. You can find more formatting-related information in the VHDL Style File and RAT Assembly Style File documents. Most importantly, if you have questions, ask your instructor before you submit sub-par work.

> The Guiding Question: ***Would I submit this work to my supervisor?***
>
> If the answer is no, then consider quality improvements before submission.

Submission Ordering:

1) Cover sheet with course names & sections, group member names, executive summary of experiment
2) Diagrams (schematics, state diagrams, etc. from experiment assignment)
3) Source code (VHDL, assembly code, etc. from experiment assignment)
4) Answers to question set (include question with answer)
5) Design Problem solutions (HW & SW problems: code and/or diagrams)

General Comments:

- Only submit one lab submission per lab group, unless directed otherwise
- Lab reports should be stapled high in the upper-upper-left corner
- Lab work submissions should be "stand alone", which means that anyone can pick up the submission and know what they're looking at. This means:
    - All problem-type questions include the problem statement in an appropriate location
    - Answers to questions should include the question
- Write concisely; save the schmooze for courses where lazy profs have students do their grading for them
- Lab submissions should be neat; use whitespace make submissions organized, and readable
- All diagrams (timing, state, circuit diagrams, etc.) should include a title and/or caption
- Do not break tables or diagrams across pages
- Use hex notation for all signal values wider than four bits
- Use engineering notation for all numbers
- Do not put source code in the body of a report; print and include with the report instead
- Do not include source code for anything you did not write

Lab Activity Questions:

- Include the question you're answering before you write the answer to the question
- Use white-space to delineate questions and answers to make them more readable

Hardware (Schematic) Diagrams:

- Must be neat; can be hand drawn or drawn using drawing program
- Do not route control and status signals to and from FSMs, but clearly label both ends of signals
- Do not route clock signals, but clearly label at clock inputs and outputs
- Do not use "hump notation"

Simulator Printouts:

- Timing diagrams should be annotated (handwritten is fine, but must be neat)
- Timing diagrams should be simulator outputs and never cut-and-paste screen shots (when possible)
- Timing diagrams should primarily be "things of interest" by using the correct amount of magnification
- Timing diagram annotations should generally show the causality of signals changes and/or values

FSM State Diagrams:

- State diagrams can be handwritten but must be neat
- FSM state bubbles should have symbolic names that roughly indicate purpose of state
- FSM input and output signals should have symbolic names that roughly indicate their purpose
- FSM states should only list outputs that are critical to that state; do not include output signals associated with a state if they are a "don't care" relative to that state
- State diagrams should include legends indicating inputs, output, and state name

General Source Code (VHDL & RAT Assembly)

- Each separate file of code should begin on a new page
- Source code files should have header that include names of group member, lab activity number, and a brief but complete description of the file's content
- Never use the tab key for any reason when writing source code
- Use the "courier new" font for all submitted source code (VHDL, RAT Assembly, C, etc.)
- Each file of code should have header describing what the code does and who wrote the code
- Do not allow lines of source code to wrap
- Code should be properly indented (see appropriate style files)
- Code should use white space to increase readability of code
- Code should be sufficiently commented
- Code should use self-commenting labels (labels, signal, variable, entity, architecture names, directives, constants, etc.)
- Code should make no more than one assignment per line including entity declarations
- Code should be printed using separate files and included with report

VHDL Source Code Specific:

- Code should define only one item (signal, variable, etc.) per line
- Code should use "s_", "v_", or "r_" prefix for signals, variable, or registers, respectively
- Structural models use direct mapping only
- Code should declare no more than one variable per line (entities & declarations)

RAT Assembly Source Code Specific (CPE 233 only):

- All subroutines should have proper headers describing what the subroutine does which registers are permanently modified by the subroutine
- All instructions & directives, the left-most instruction operands, and comments should be aligned

Lab Activity Report Guidelines

Lab reports provide a description of your work in the lab. There are two basic forms of information in the lab report: an objective portion and a subjective portion. The objective portion of the lab report should document what you did in the lab (designed stuff, took data points, etc.) while the subjective portion should provide your interpretation of the results you obtained during the lab (namely the conclusion).

As you'll find out in Cal Poly Land, each instructor has their own set of rules and expectations regarding the lab report. Generally speaking, I grade lab reports based on how well you follow those rules and meet those expectations. The goal of the following comments is to help you generate a quality lab report while minimizing the amount of time you spend writing the report. Please do not hesitate to ask if you have questions. Lab write-ups are not exams so feel free to ask me to look over your report before you turn it in; the more feedback you get, the better your report will be.

General Rules:

- Lab reports should be written using a word processor and submitted in hard copy form.
- Each lab group should submit one lab report. Each member of the lab group should write their own conclusion independently from other group members and submit it with the lab report.
- Reports should not have a title page. The experiment title should be boldly centered on the first page. The course and section number, and the names of group members should also appear on the first page.
- Wording in the lab report should be brief but concise. Longer lab reports rarely correlate to higher quality lab reports. Moreover, using concise wording save you time writing the report and saves someone else time reading it.
- Wording in the lab report should use "good English" and appropriate technical style. Correct spelling, appropriate use of technical terms, and appropriate technical style allows you to create a professional document that highlights your lab activities. If you are new to technical report writing, strongly consider having someone else read over your report before you submit it. There are also many features in MS Word that can help you with the grammar.
- Lab reports should be neat and intelligently organized. Hand-written and hand-drawn items such as circuit diagrams should be neat (use appropriate drawing software if necessary).
- All figures and diagrams should contain captions and/or titles and should be referenced from the body of the report (do not say things like "in the figure below"). Plan to use the referencing feature of your word processor.
- Different sections and diagrams in the lab report should be well delineated from each other. Using extra space to in the lab report generally creates a more professional looking document.

Report Format:

Each lab report should at least contain the following clearly indicated sections. More sections are permissible but the document should remain concise and well organized.

- **Objectives** (Say why you're bothering): This section states that point of performing the lab activity. This section is generally a rewording of the stated activities objectives in such a way as to show that understood what you're attempting to do with in the lab activity. Don't just copy the state lab activity's objectives; use your own words instead.

- **Procedures** (Say what you did): Describe what you did during the lab activity.

    - The use of words in this section should be minimized in favor of more expressive items such as truth tables, equations, circuit diagrams, block diagrams, timing diagrams.
    - Someone should be able to read though this section and know exactly where you started, where you ended, and the steps you used to arrive at your destination. The flow of this section should match the flow of tasks during the lab activity. This section should not depend on the description of the lab. In other words, assume that the person reading your lab report does not have a copy of the lab description.
    - This section should be written using normal English sentences and paragraphs as opposed to bulleted or numbered lists of tasks and/or commands.

- **Testing** (Say why you think you did it correctly): Describe the testing procedures you used to verify your circuitry met the design criteria.

- **Conclusion** (Sum up the experience): See comments below.

- **Questions** (Say what you're supposed to say): Be sure to include the answers to any questions that may appear at the end of lab activity description in your lab report.

    - Answer the questions in such a way as to re-state the original question.
    - If the question requires that you perform some non-trivial calculations, include those with the lab report.

The Conclusion:

- Conclusions should contain the following information:

    - A brief description of what was done in the experiment (2-3 sentences).
    - Wording that implies you understand the concepts presented in the experiment.
    - Wording that describes how the experiment relates to other experiments and/or topics discussed in lecture or in the world in general.
    - Wording that indicates the objectives of the experiment were met. Don't simply state that the experiment's objectives were met; support the assertion indirectly with your wording.

- Conclusions should *not* contain the following:

    - Detailed descriptions of the procedure followed during the experiment.
    - Detailed descriptions of the circuits designed or used during the experiment.
    - Comments regarding whether you liked the experiment or not.
    - Comments regarding how much you learned during the experiment.
    - Comments that state directly that the experiments objectives were met.

Experiment #1:

Digital Design with VHDL Memory Models

Learning Objectives:

- To refresh your VHDL skills regarding sequential, combinatorial and arithmetic circuits
- To review the basic operation of the Xilinx simulator
- To implement a simple but potentially useful circuit using a RAM modeled in VHDL

The Big Picture: This experiment is most likely your first time working the structured memory. Out there in computer-land, there are many types of structure memory devices, but you can classify them as either RAM or ROM. VHDL allows you to model both types of structured memory using familiar VHDL syntax. As long as your structured memory models do not become overly complex, modeling these devices in VHDL is straightforward and generally involves finding a model similar to what you need and adjusting the model to suit your particular needs.

This is a course in computer design, which is a subset of digital design. You'll soon find out that a computer is nothing more than a large digital circuit containing many standard (and familiar) sequential and combinatorial elements under the control of an FSM. This experiment has all those characteristics but cannot do as many interesting things as a computer can. The good news is that if you complete and understand this experiment, you'll not see too much new material as you build your computer in this course. In other words, the lab experiments associated with this course do not become any harder to understand than the concepts contained in this experiment.

The Modern Approach to Digital Design: The best place to start is to define "digital design". Digital design involves using digital circuits to solve problems. There are many approaches to use for solving problems; digital design is just one of them. Moreover, there are also different "levels" of digital design. For example, I can buy a digital thermostat to replace the analog thermostat on the heater that warms my home. While this fits well under the definition of digital design, it is not an overly impressive level of digital design.

The information associated with this course deals with digital design at the "engineering" level. We'll avoid defining this level, but suffice to say it involves something more than determining an off-the-shelf solution to problems. Engineering is simply an efficient process from going from problem to solution. Once again, there are many paths to the solutions to your problems, but the engineering path is the efficient path. Additionally, part of the engineering solution is to question your solution once you arrive at it: could you have solved the problem better? Could you have taken a different and better path from problem to solution? In summary, engineering is not solely about solving problems; it about solving problems in an efficient manner and learning enough along the way to think of a better solution once you arrive at your first solution.

The modern engineering-based digital design paradigm includes two basic aspects: 1) modern digital designs are hierarchical in nature, and, 2) modern digital designs are modular in nature. How these aspects relate to the engineering-based approach to digital are the following.

- In order to understand anything other than simple digital designs, you must constantly abstract the designs to higher levels. If you remain at lower design levels, you find it harder to find the best path to your solution in an efficient manner. There are of course times where you must deal with lower-level design issues, but you avoid this where possible in the name of efficiency. The general model is to collect your lower-level designs, and then place them in a box (an abstracted module). The box is a higher-level than the components in the box, and you thus now have a hierarchical design.
- In order to design efficiently, your approach is to take various boxes (or modules), place them into your design, and connect the modules in such a way as to solve the problem at hand. There was a time when most digital designs started from scratch (at a low level) and slowly built upwards, but like the dinosaurs, those times have generally passed. Modern engineering-based digital design primarily concerns the interfacing of pre-designed components in such a way as to solve your problem. Inherent in this definition is the need to understand all "usage" aspects of the modules you use in your design, which facilitates your search for an efficient path to the solution. In other words, if you don't understand how the module's control inputs affect the operation of the modules, you won't be able to effectively use that module and thus the overall efficiency of your design will suffer. Most conveniently, many complex digital circuits are created using the most basic digital modules, which are relatively few and simple to understand (modules such as MUXes, decoders, registers, counters, and basic logic gates).

Lab Assignment A: Design a circuit that writes the first 16 Fibonacci sequence numbers (1, 1, 2, 3, 5, 8…) to a RAM and then displays those numbers continuously. Figure 1 shows the top-level black box diagram for this circuit. Minimize the amount of hardware you use in your design.

- The circuit initially displays the RAM's contents upon power-up. In other words, the circuit continuously indexes through the RAM in counting order until the user presses the button. When the user presses the left-most button on the development board, the FSM proceeds to write the Fibonacci numbers to RAM and then constantly displays them in ascending order after it writes them. Drive the 7-segment display as you write the Fibonacci numbers to the RAM, recalling that the 7-segment driver uses a faster clock.
- Your FSM should include a "display" state that does nothing more than displays the contents of the RAM in a sequential manner. The FSM should return to this state after it writes the Fibonacci sequence to the RAM.
- All synchronous circuit operations should be at 3Hz.
- Display the address input of the RAM to the four right-most LEDs on the development board. In this way, you'll display every RAM access (reading or writing) on the LEDs.
- Demonstrate your working circuit to the instructor.

Lab Assignment B: Design a circuit that counts the number of bits that are set in all a 16x8 RAM's storage locations and then displays that count on the 7-segment display. The circuit first loads the data to be set-bit-counted into RAM from a ROM, which the experiment provides for you. Figure 1 shows the top-level black box diagram for this circuit. Minimize the amount of hardware you use in your design.

- The circuit initially displays the RAM's contents upon power-up. In other words, the circuit continuously indexes through the RAM in counting order until the user presses the button. When the user presses the left-most button on the development board, the FSM proceeds to count the set bits in each RAM location. The circuit displays the set-bit count on the 7-segment display as the bits are being counted. When the count is complete, the 7-segment display shows the final set-bit count.
- All synchronous circuit operations should be at 5Hz.
- Display the address input of the RAM to the four right-most LEDs on the development board. In this way, you'll display every RAM access (reading or writing) on the LEDs.
- Demonstrate your working circuit to the instructor.

Lab Assignment C: Design a circuit that, upon a button press, implements a bubble sort on the data stored in a 16x8 ROM. The circuit first loads the data to be sorted into a RAM from a ROM; the experiment provides the ROM for you. When the sort is complete, the circuit displays the sorted contents of the RAM continuously on the 7-segment display one RAM location at a time; the sort should be in ascending order. Consider the values in RAM to be unsigned 8-bit values. Figure 1 shows the top-level black box diagram for this circuit. Minimize the amount of hardware you use in your design.

- The circuit initially displays the RAM's contents upon power-up. In other words, the circuit continuously indexes through the RAM in counting order until the user presses the button. When the user presses the left-most button on the development board, the FSM proceeds to write the contents of the ROM to the RAM and then sorts the numbers in the RAM. When the sort is complete, the circuit displays the RAM contents in sorted order.
- Your FSM should include a "display" state that does nothing more than displays the contents of the RAM in a sequential manner. The FSM should return to this state after it sorts the values in the RAM.
- All synchronous circuit operations should be at 6Hz for display purposes; you may want to slow the clock to help with debugging during the development phase of this experiment.
- Display the address input of the RAM to the four right-most LEDs on the development board. In this way, you'll display every RAM access (reading or writing) on the LEDs.
- Use only on RAM in your circuit
- Demonstrate your working circuit to the instructor.

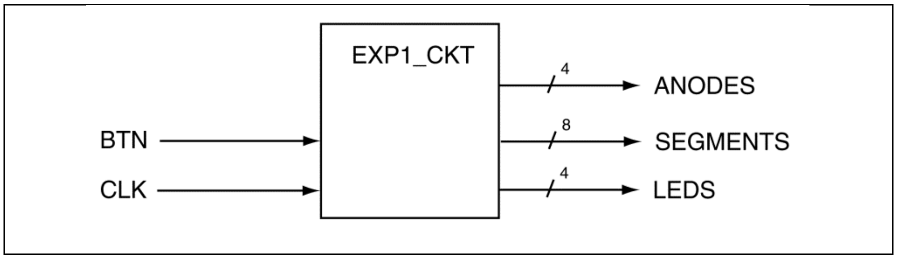


Figure 1: The top-level black box diagram for this experiment.

Constraints:

- Use the same clock frequency for both reading and writing the RAM.
- You need an FSM to control your circuit; don't use more than ten states in your FSM.
- Only use standard digital modules in your design. Don't use VHDL to model non-standard circuits. Standard circuits include registers, counters, MUXes, decoders, etc.
- Don't use more than one RAM in your design.
- Don't use VHDL mathematical operators anywhere except in RCA or counter circuits
- Your FSM should be 100% independent of the length of the requested Fibonacci sequence, the size of the RAM for the bit-count circuit, or the number of items in the bubble sort circuit. This means if the size of one of these items changes, you will not need to change the FSM in your design.
- The only memory in your FSM should be the state variables
- The FSM should not respond to another button press until the assigned task has completed and the button is released

Hints:

- There are many ways to implement these designs; whatever way you choose, make it work. Think about it before choosing what you feel is the best approach. Strive to keep your circuit as simple as possible by only using standard digital modules. If you run into issues, you'll need to use the simulator to work through them.
- Ignore all debounce issues associated with the button

Questions:

1. Briefly explain why was it is not problematic to ignore the bounce issues associated with the button in this experiment?
2. Briefly but completely state and explain in your own words the two main aspects of the modern digital design paradigm.
3. In your own words, provide a complete written description of how the circuit you designed in this experiment operates. Be sure to reference the block diagram for your circuit. This description should be more than a detailed description of the state diagram.

4. Based on your circuit design, how many clock cycles does your design require complete the assigned task in this experiment after the user presses a button? If appropriate, state the best and worst case number of clock cycles.
5. Consider a Moore-type FSM output; if an output becomes asserted when it enters a given state, what is the soonest that output could take effect for an external hardware module? For this problem, assume the external hardware module is synchronous and shares the same clock as the FSM. Also, assume the FSM and external module are both RET devices.
6. Describe whether you implemented your circuit as a Mealy or Moore-type FSM. Briefly state the reasons why you choose one FSM model over the other.
7. Limiting the number of states in the design forces you to create a "generic" design, which roughly means you "reused" some states in your FSM as it completed the assigned task in this experiment. Briefly but completely explain the main benefit of implementing this particular style of generic design in the FSM. One of the constraints of the FSM in this experiment was to make the FSM such that if the length of the sequence changed, the FSM would still work. This question does not refer to that type of genericity.
8. Briefly explain what you think is the limiting factor in decreasing the amount of time required to write to complete the assigned task in this experiment. In other words, speculate on what part of your circuit would limit the system clock speed for your design. For this problem, assume the development board can provide any possible clock frequency. HINT: The answer is *not* the number of states in the state diagram.

<u>Programming Assignment:</u>

1. Write a fragment of C code that mimics the circuit you designed in this experiment. Don't exceed 20 lines of code for this problem.

<u>Hardware Design Assignment:</u>

Show a block diagram that you could use to solve the following problems. Also include a state diagram that describes the operation of an FSM you could use to obtain the requested result. For this problem, do not use modules other than standard digital modules with typical inputs and outputs (counters, registers, shift register, comparators, RAMs, MUXes, RCAs, and decoders). If you use a decoder, be sure to provide an adequate model for it. Minimize the amount of hardware you use in your design including bit-widths of various modules.

- Design a circuit that adds the contents of a 16x8 RAM and stores the result in an accumulator. The circuit adds these values when a button is pressed.

Lab Submission Deliverables:

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.

2. A complete solution to the Programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness and neatness counts.

4. Black box circuit diagrams for the two highest levels of your design. (Figure 1 shows the black box model for the highest level). Neatly hand-draw these diagrams. Neatness counts.

5. The VHDL code for all models that you wrote in this experiment. Proper VHDL syntax and structure counts.

6. The state diagram for the FSM you used in this experiment. Make sure your state diagram uses symbolic names for the states, inputs, and output signals. The labels you use in your state diagram should match the labels you use in your VHDL code. Neatness counts. Proper state diagram syntax counts also.

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #2

Working with VHDL RAM Models

Learning Objectives:

- To refresh your VHDL skills regarding sequential, combinatorial and arithmetic circuits
- To review the basic operation of the Xilinx simulator
- To implement a simple but rather pointless circuit using a RAM modeled in VHDL

The Big Picture: This experiment is most likely your first time working the structured memory. Out there in computer land, there are many types of structure memory devices, but you can classify them as either RAM or ROM. VHDL allows you to model both types of structured memory using familiar VHDL syntax. As long as your structured memory models do not become overly complex, modeling these devices in VHDL is straightforward and generally involves finding a model similar to what you need and adjusting the model to suit your particular needs.

Assignment Design a circuit that displays the values stored in a RAM. You'll feed the output of the RAM to the standards 7-segment display module you used extensively in CPE 133, which mean the values you'll store in the RAM are BCD values. **Error! Reference source not found.** shows the BBD for the RAM module you need to model while Figure 1 shows the top-level BBD.

Experiment Details The following bullets describe the complete operation of the circuit.

- When no button is pushed, the circuit feeds the contents of the RAM to the sseg_disp module in order to display the individual RAM contents are decimal numbers. Use a counter to automatically step through the 16 individual RAM locations. The counter should display the RAM contents at about a 2Hz rate.
- When you hit BTN0 (button 0), the circuit fills the RAM with a binary count. Specifically, RAM location 0 should contain "0000"; RAM location 1 should contain "0001", etc. Write to the RAM fast enough so that the user can't detect anything strange on the development board's 7-segment displays. This step results in the 7-segment display showing a decimal count (0→15).
- When you hit BTN1 (button 1), the circuit fills every RAM location with the value represented by the development boards four lowest switch positions. So if the switches have are set to "1011", fill every RAM location with "1011". This results in a decimal 11 being displayed on the 7-segment display; this number will not appear to change as "1011" should

now be in every RAM location. Once again, the writing to RAM should be fast enough so that the user can't detect anything funny on the development board's 7-segment displays.

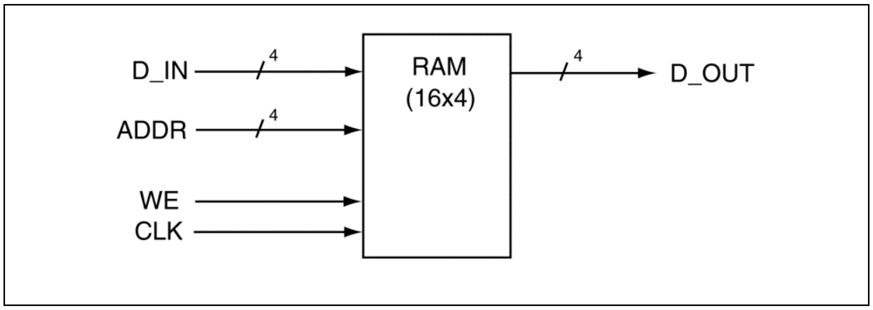


Figure 2: The RAM you need to model and use in this experiment.

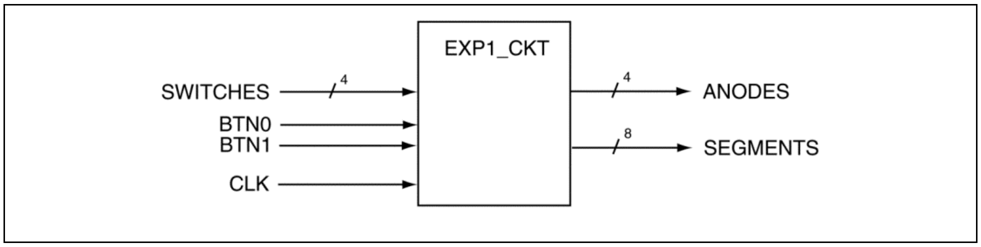


Figure 3: The top-level black box diagram for this experiment.

Constraints:

- Sorry folks, you're going to need a FSM to control this circuit. Do not use more than twelve states in your FSM.

Hints:

- There are many ways to do this; make it work.
- You're going to need both a fast clock (for writing to the RAM) and a slow clock (for displaying the RAM contents). You can connect the system clock to the sseg_disp modules.
- There are bound to be debounce issues with the buttons; don't worry about them.
- This circuit is both complicated and simple at the same time. It's complicated when it does not work and it's simple when it does work. If your circuit is complicated, you'll know it's time to fire-up the simulator in order to help you see what is not going right with your circuit.

Questions:

1. In your own words, provide a written description of how your circuit operates. Be sure to reference the block diagram for your circuit.

2. Could you have completed this circuit without using a FSM? Briefly but completely explain.

3. Briefly explain why you think it was that the number of states in the FSM was limited to 12.

4. Briefly explain an approach you could use to determine how fast you can write to every location in memory.

5. Briefly explain what you think is the limiting factor in decreasing the amount of time it requires to write to every RAM location.

6. This experiment used on RAM to store two different types of data. The issue was that when the set of data you wanted to use was not in the RAM, your circuit had to take the time to load the new data. Briefly describe the changes you would need to make to this circuit if you had two RAM devices, each of them holding the appropriate data. Be sure to state if you would need the button inputs or not.

Deliverables:

1. Answers to the questions in the previous section.
2. Black box circuit diagrams for the two highest levels of your design. (**Error! Reference source not found.** shows the black box model for the highest level).
3. VHDL code for all modules that you write; don't submit the modules that the experiment provides for you.
4. State diagram for FSM. Make sure your state diagram uses symbolic names for both the states in your state diagram as well as the input and output signals.

Experiment #3:

Counters as Program Counters

Learning Objectives:

- To understand how to use an n-bit register (counter) and MUX, which can implement the features typically required of a "program counter"
- To become familiar with integration the prog_rom.vhd file into a VHDL wrapper

The Big Picture: While there are many different computer architectures that include many different modules, one module in most architectures is the Program Counter (PC). The basic definition of a computer is a circuit that executes instruction stored in memory to give a desired result. The PC is the component that provides the address of the instruction in program memory that is either executing or is scheduled for execution (depending on the time you examine it). Nevertheless, as important as the functionality of a PC sounds, the PC (and supporting hardware) is a relatively simple device.

General Notes: A counter is a special form of register that performs operations typically associated with counting. Counters generally operate synchronously (an active clock edge synchronizes all output changes) but they also can have functions that are asynchronous (such as load, clear, etc.). Modeling counters in VHDL is relatively simple as you can model an "n-bit" VHDL counter model as with a relatively few simple statements.

The PC in this experiment includes flexible external loading control for the PC with the addition of a MUX. When the PC is operating in a computer, it operates "normally" by incrementing the count in order to access the next instruction in program memory. The PC must also be able to parallel load values to support computer instructions that are not the next instruction in program memory sequence (such as branch instructions, subroutine calls, return from subroutines, etc.).

Program memory stores the machine code associated with a given RAT MCU program. The RAT MCU architecture refers to program memory as the "prog_rom" as the assembler automatically generates a VHDL model for the program memory: "prog_rom.vhd". The assembler assigns values to the prog_rom; these values can only be read not be changed (written to) by any aspect of the RAT MCU hardware or firmware. The prog_rom is 1024 locations deep by 18 bits wide (1024 x 18 or 1k x 18); hence, it has 10 address lines and 18 data lines. The data lines are for the RAT instructions, each of which is 18-bits wide. The RAT MCU architecture configures the prog_rom to have synchronous reads where the ten-bit output of the PC acts as an address input to the prog_rom. Because the PC has ten address lines, the program memory can address 2^{10}, (1024 or 1k) unique instructions in program memory.

Assignment:

1. Design a PC and associated input selection MUX. Figure 5 shows the PC while Figure 6 shows the PC and the associated input selection MUX. Use at least two modules in your design, one for the PC and one for the MUX; these modules can be either two separate entities (two-level design) or two processes in the same entity (one-level design). If you start from the counter template, make sure you remove all unused functionality from the code before you submit it.

2. Use the provided prog_rom.vhd file or assemble the RAT MCU code in Figure 4 to generate a new prog_rom.vhd file. Include this model with the PC hardware you modeled in the previous step.

```
.EQU LED_PORT    = 0x30              ; port for output
.EQU SWITCH_PORT = 0x40              ; port for input

.CSEG
.ORG         0x10                    ; program starts here

main:        IN      r10,SWITCH_PORT
             ADD     r10,0x01
             OUT     r10,LED_PORT
             BRN     main            ; endless loop
```

Figure 4: RAT MCU assembly language test program.

3. Write a testbench that tests the operation of your PC hardware connected to the prog_rom from the previous step. Verify each of the following operations are operating properly in the PC/prog_rom circuit:
    - RST
    - PC_LD: test each MUX data input (FROM_IMMED, FROM_STACK, & 0x3FF)
    - PC_INC: test by loading an immediate value (FROM_IMMED) of 0x10 into the PC. Show each of the instructions in the given program.
4. Verify that the output of the prog_rom matches the instruction bits by comparing the 18-bit IR output to the output of the assembly listing file generated by the RAT assembler. Note that the assembly listing file will have the same name as the assembly source file but with an ".asl" file extension. Include the well annotated simulator output and a ***partial*** ".asl" file (showing the machine code for the instruction in the given program) in your lab submission.

Circuit Details:

Figure 5 shows the top-level black box model of the PC you'll implement. Table 1 provides an overview of the PC's operation in the context of the signals in Figure 5.

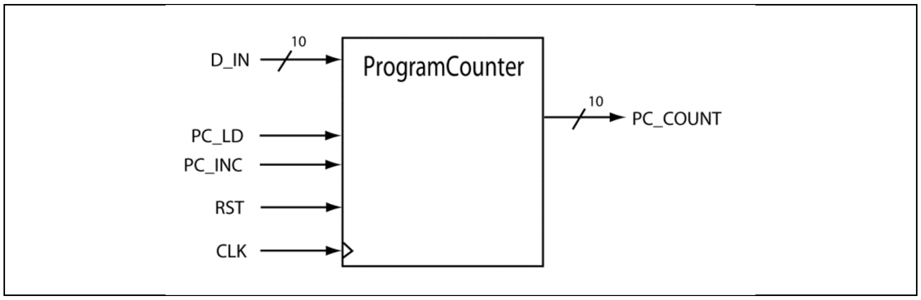


Figure 5: Program counter black box diagram.

Signal	Comment
PC_COUNT	The current value in the PC; the PC uses this value as an address to access values in the program memory.
RST	An active-high synchronous reset.
PC_LD	An active high signal that synchronously loads D_IN into the PC. This signal has a higher precedence than the PC_INC input.
PC_INC	An active high signal that synchronously increments the value in the PC.
CLK	Synchronizes all PC operations on the rising edge

Table 1: Tabular explanation of the program counter.

Figure 6 shows that you can load the PC from various sources. The MUX allows the PC to change according to the currently executing instruction. Figure 7 shows the PC connected to the prog_rom. This circuit is the circuit you will verify using a testbench. The MUX inputs include:

- FROM_IMMED: Branch and Call instructions obtain the new value of the PC from the immediate value included as part of those individual instruction formats.
- FROM_STACK: Return instructions (return from subroutine or interrupt) obtain the new PC value from the stack (you'll learn about stacks and interrupts later). Instructions such as RET utilize this path to the PC.
- 0x3FF: Interrupts (when the RAT acts on them) set the PC to the interrupt vector: 0x3FF.

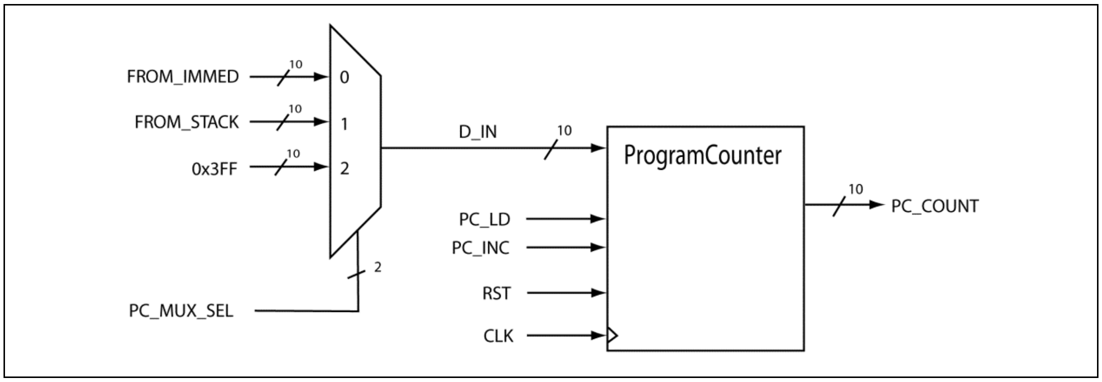


Figure 6: Program counter with input selection MUX.

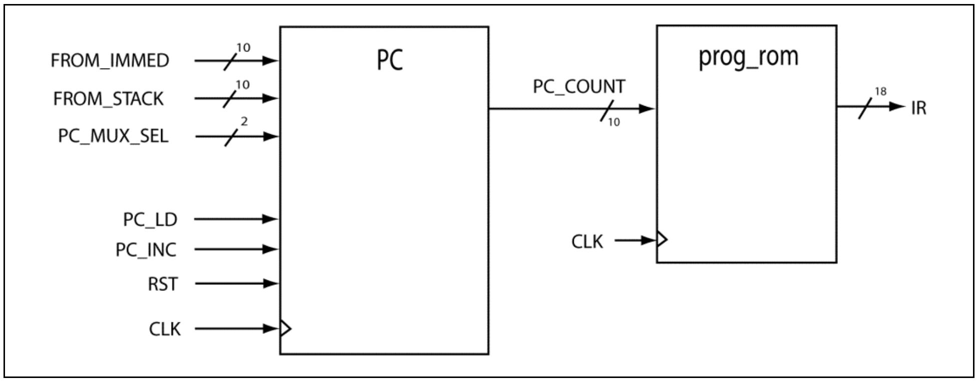


Figure 7: Program counter connected to the prog_rom.

Programming Assignment:

Write two separate RAT MCU assembly language programs that endlessly do the following tasks. Make sure your programs are in proper form including comments and file banner (consider viewing the style file located at the end of this lab activity manual).

- Reads data from input associated with port_id 0x99, multiplies the data by three, and outputs the data to the output associated with port_id x033. Don’t worry about the data overflowing the registers for this program.

- Reads data from input associated with port_id 0xA7, changes the sign of the data, and outputs the data to the output associated with port_id x0B8. For this input, assume the data read to the input port is in RC form.

Hardware Design Assignment:

Show a block diagram and a state diagram that you could use to solve the following problem. For this problem, do not use modules other than standard digital modules with typical inputs and outputs (counters, registers, shift register, comparators, RAMs, MUXes, RCAs, and decoders). If you use a decoder, be sure to provide an adequate model for it. Minimize the amount of hardware you use in your design including bit-widths of various modules and datapaths.

- Upon receiving a "GO" signal, the circuit finds the minimum value in a 16x8 RAM. Upon completion, the circuit continually outputs both the minimum value and the RAM address of that value until another GO signal is detected.

Questions:

1. The PC has two types of input signals: control signals and data signals, where we consider the clock input a type of control signal. List which inputs are data inputs and which are control inputs. For this problem, use the "PC" as it appears in Figure 7.
2. If the PC were to increment every clock cycle, could the prog_rom output ever show the data associated with the current value on the PC's output? In other words, could the output of the prog_rom ever be associated with the current output of the PC. Briefly but fully explain your answer.
3. Suddenly, your boss wants to use a different prog_rom: instead of one 1024x18, she want to use two 512x18 ROMs. Show the circuit that you would need to make this work. This circuit should contain two ROMs, the same 10-bit PC, and whatever other circuitry you need to make the circuit function properly. Draw a schematic of your solution.
4. In terms of an executing computer program, briefly describe why the FROM_IMMED input is associated with branch and-type instructions and why FROM_STACK is associated with return-type instructions. You will need to consult the assembler manual for this question.
5. List two instructions from the RAT instruction set that would use the PC's "FROM_IMMED" input. You will need to consult the RAT MCU Assembler Manual for this question. In particular, examine the RTL associated with each instruction.
6. List two instructions from the RAT instruction set that would use the PC's "FROM_STACK" input. You will need to consult the RAT MCU Assembler Manual for this question. In particular, examine the RTL associated with each instruction.
7. Briefly describe the general purpose of the assembly language listing file.
8. What would be the total bit and byte capacity of the prog_rom if you doubled the number of instructions the prog_rom could store?
9. What would be the capacity of the prog_rom if you kept the number of storage locations the same bus increased the size of each instruction from 18 to 24 bits? Show your calculations for this question.
10. If the prog_rom increased from 1k x 18 to 4k x 24, what would be the total bit capacity and how many addresses lines would be required for the device? Show your calculations for this question.

Deliverables:

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.

2. A complete solution to the Programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness and neatness count.

4. VHDL code for the PC and support hardware you modeled in this experiment.

    - DO NOT PRINT THE prog_rom.vhd FILE

5. Schematic diagram showing the three modules you used in this experiment and their interconnections.

6. Fragment of the assembly language listing file (.asl) showing the machine code for the program used in this experiment.

    - DO NOT PRINT THE prog_rom.vhd FILE or the entire .asl file

7. Fully annotated simulator output for your testbench

    - DO NOT INCLUDE THE prog_rom.vhd FILE

    - DO NOT INCLUDE YOUR TESTBENCH CODE

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #4:

Introduction to Assembly Language Programming and Disassembly

Learning Objectives:

- To understand the basics of assembly language programming
- To observe how an assembler converts an assembly language program into machine code
- To use an assembly language simulator to analyze an assembly language program

General Notes: This lab uses the concept of disassembling machine code as an approach to presenting the architecture of the RAT MCU and basic assembly language programming. This experiment uses the painful approach of disassembling a program in the form of RAT MCU machine code (roughly known as "reverse engineering") to demonstrate the structure, form, and content of assembly language instructions. Stated more plainly, you need to perform "disassembly" of machine code in this experiment, meaning you start with machine code and generate the associated assembly code. This experiment refers to disassembly as "reverse engineering" simply because it sounds cooler.

Assignment: This assignment has two different parts.

Part 1:

a) Figure 8 shows Program #1, an example RAT MCU assembly language program fragment (it's a fragment because it's missing a proper file banner). Enter and assemble this program in the RAT MCU Simulator. Be sure to include a blank line after the last line of code.

```
.EQU LED_PORT    = 0x10             ; port address for output
.EQU SWITCH_PORT = 0x40             ; port address for input

.CSEG
.ORG         0x40                   ; instruction storage starts here

main:        IN      r20,SWITCH_PORT
             MOV     r10,0xFE
             MOV     r11,0x02
             ADD     r10,r11
             ADD     r10,0x14
             SUB     r10,0x03
             OUT     r10,LED_PORT
             BRN     main           ; endless loop
```

Figure 8: Experiment 4 test Program #1.

b) Step through the program with the RAT simulator and analyze Program #1 to determine various values as the program executes. Complete Table 2 by including the following information ***after*** the simulator executes the instruction in the "instruction" column. The value in Table 2 is an example of how one line in the completed table should appear. The completed table should have one line for each instruction in the program.

PC Val (hex)	Instruction	Dest/Value	Source/Value	C Val	Z Val	Port Address (if applicable)
	`MOV   R10,0x05`	`R10 / 0x05`	`Immed / 0x05`	1	0	N/A

Table 2: Program #1 analysis table template.

Part 2:

a) Figure 9 shows excerpt from the prog_rom.vhd file associated with a certain program. Your mission is to disassemble the machine code and provide a source code listing for the program that generated the machine code. Note that you may disregard any other lines from this VHDL file, as the code below provides you with all the information you need to disassemble the machine code. Also, note that because the code is located in INIT_03 and fills starting from the least significant byte of the given listing, the address of the first instruction is located at 0x30. See the example of reverse engineering a RAT MCU prog_rom.vhd file at the end of this experiment description. You need to generate a table similar to that of Table 3 for this part of the experiment.

```
   attribute INIT_00 of ram_1024_x_18 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
   attribute INIT_01 of ram_1024_x_18 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
   attribute INIT_02 of ram_1024_x_18 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
   attribute INIT_03 of ram_1024_x_18 : label is
"0000000081800001030181ABC201030001010001030180006300620861676023";


   attribute INITP_00 of ram_1024_x_18 : label is
"00000000000000000000000000000000014955FF000000000000000000000000";
```

Figure 9: Machine code listing for disassembly.

b) Enter the reverse engineered program in the RAT MCU simulator; make sure this program has proper programming style, including headers, indentation, comments, etc. Step through the program using the RAT MCU simulator. For this part of the experiment, print out the assembly language program you tested on the simulator.

c) The code in Figure 9 swaps data in two registers. If you don't see this happen in the RAT Simulator, you made an error in your disassembly process. You may not understand all the instructions in the program (feel free to ask about them), but you still are able to see what the working program does.

d) Demonstrate your working program to the instructor.

Programming Assignment:

Write a RAT assembly language program that endlessly does the following: inputs ten values from the input associated with the port_id of 0x9A. Sum these values and store the results in register r30 (lower byte) and register r31 (upper byte). After you sum these ten values, continually divide the total by 2 as many times as it takes to make the upper byte (register 31) equal to zero. Use the RAT MCU Simulator to ensure you program performs as desired.

Hardware Design Assignment:

Show a block diagram that you could use to solve the following problems. Also include a state diagram that describes the operation of an FSM you could use to obtain the requested result. For this problem, do not use modules other than standard digital modules with typical inputs and outputs (counters, registers, shift register, comparators, RAMs, MUXes, RCAs, and decoders). If you use a decoder, be sure to provide an adequate model for it. Minimize the amount of hardware you use in your design including bit-widths of various modules.

- Design a circuit that counts the number of values in a 16x8 RAM that are in the range [38,78] upon the pressing of a button. Consider the contents of RAM to be unsigned binary numbers. Use a counter module to store the resulting quantity value.

Questions:

1. The prog_rom.vhd file in this experiment models a synchronous circuit. Briefly describe what the notion of synchronous means in the context of this ROM.
2. The prog_rom.vhd file models a ROM. Briefly examine the prog_rom.vhd code and state what particular signal assignment officially turns the model into a ROM.
3. Briefly describe what (or who) determines the number of instruction formats a given instruction set contains.
4. List the five RAT MCU distinct instruction formats. Explicitly state the factors that differential these instruction types.
5. The five RAT MCU instruction formats contain different fields for different instructions. In the context of the different instruction formats, briefly describe how it is possible for the same sets of bits to satisfy these different field requirements.
6. You can typically use assembly language to make a given program written in a higher-level language run faster. Briefly but fully explain this concept.
7. We refer to computer languages such as C and Java as being "portable". Define this term in your own words; be sure to reference the role of assembly languages in your answer.

8. Briefly but completely explain how you can disassemble the machine code without the presence of text-based label. Specifically, branch instructions seem to rely on labels in the assembly code; how then can you successfully disassemble a program without the original labels.
9. Is it possible to have two of the same labels in an assembly language program? Briefly but completely explain.
10. Is it possible to have two or more labels in an assembly language program with the same numerical value? Briefly but completely explain.
11. The programs you worked with in this experiment were implemented as “endless loops”. Using the RAT MCU assembly code, could you write a program in such a way as to stop the program from executing instructions under program control? Briefly but completely explain.

Deliverables:

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.

2. A complete solution to the Programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness and neatness counts.

4. Procedures: Part 1:
   i. Source code listing for the assembly language program
   ii. Completed table (similar to Table 2) for the program

5. Procedures: Part 2:
   i. A table (similar to Table 3) showing the disassembly process
   ii. Typed assembly code for the disassembled assembly language program using proper assembly language program format
   iii. The instructor’s signature showing that your code worked properly

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Example of Reverse Engineering RAT Assembly Code

1.) Consider the following RAT assembly language program:

```
.EQU LED_PORT = 0x10              ; port for output

.CSEG
.ORG         0x00                 ; data starts here

main:        MOV     r10,0x05
             MOV     r11,0x64
             ADD     r10,r11
             ADD     r10,0x14
             MOV     r20,r10
             OUT     r20,LED_PORT
             BRN     main          ; endless loop
```

2.) This program produces the following lines in the prog_rom.vhd file. To reverse engineering the hex values listed in the associated prog_rom.vhd file, extract the contents of the populated rows from prog_rom.vhd. The underlining shows the bytes of interest.

```
attribute INIT_00 of ram_1024_x_18 : label is
"0000000000000000000000000000000000008000541054518A142A586B646A05";


attribute INITP_00 of ram_1024_x_18 : label is
"0000000000000000000000000000000000000000000000000000000000000C8F";
```

3.) Extract the lower 28 bytes of INIT_00 and divide them into two-byte pairs. Notice that there are seven 2-byte pairs which are non-zero and that our original assembly program had seven instructions:

```
Two-byte pair      equivalent program address
-------------      --------------------------
    0x6A05                     0
    0x6B64                     1
    0x2A58                     2
    0x8A14                     3
    0x5451                     4
    0x5410                     5
    0x8000                     6
    0x0000                     7
```

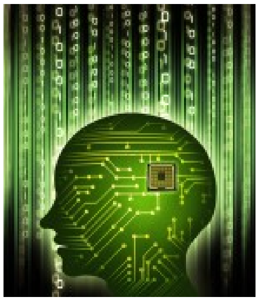
4.) Extract the four non-zero bytes of INITP_00, break into binary equivalent values, and then break these into 2-bit pairs. Note that you only want the lower seven 2-bit pairs since the original assembly program had seven instructions:

```
0C8F = 0000 1100 1000 1111
     = 00 00 11 00 10 00 11 11
     = 0x00 0x00 0x03 0x00 0x02 0x00 0x03 0x03
```

5.) Concatenate each 2-bit pair (MSBs) with each 2-byte pair from step #3 to create the 18-bit instruction. This 18-bit field is the machine code representation of one assembly language instruction. For example, the right most 2-bit pair of 11 combines with the right-most 2-byte pair of 0x6A05 to create the 18-bit machine code for the equivalent assembly instruction. You can now recreate the assembly instructions from the 18-bit binary equivalent of each instruction by comparing the opcodes and field codes of this instruction to the instruction format in the RAT Assembler Manual.

2-bit pairs from INITP_00	2 bytes from INIT_00	Concatenation of left two columns of this table	Assembly Instruction	ProgRom Mem Loc.
11	0x6A05	11 0110 1010 0000 0101	MOV R10,0X05	0
11	0x6B64	11 0110 1011 0110 0100	MOV R11,0X64	1
00	0x2A58	00 0010 1010 0101 1000	ADD R10,R11	2
10	0x8A14	10 1000 1010 0001 0100	ADD R10,0X14	3
00	0x5451	00 0101 0100 0101 0001	MOV R20,R10	4
11	0x5410	11 0101 0100 0001 0000	OUT R20,LED_PORT	5
00	0x8000	00 1000 0000 0000 0000	BRN main_loop	6

Table 3: Disassembly format table for RAT MCU instructions.



Experiment #5:
The RAT MCU Memory Modules

Learning Objectives:

- To implement the register file in the RAT MCU architecture
- To learn about the register file and how it works when integrated with the RAT MCU

General Notes: A computer is generally comprised of three major modules: 1) memory, 2) Input/Output, and 3) a CPU. We can further break down a high-level view of "memory" in a computer architecture into various types of memory based on their purpose. Though a typical computer has many types of memory elements (such as registers in the datapath), when we speak of "memory" in a computer, we are generally referring to "structured" memory, which are the relatively large "chunks" of memory in a given architecture such as RAM and ROM elements.

The register file is a relatively small but essential memory subsystem that crunch data in a relatively fast manner (faster than accessing scratch RAM) using the RAT instructions. If you need to crunch data residing the scratch RAM, you first must transfer that data to the register file because the data crunching modules (the ALU) not connected to the scratch RAM outputs. The register file is 32 locations deep by 8-bits wide (32x8). We refer to the register file as a "dual port RAM", which means you can simultaneously read from two separate memory locations in the register file, but you can only write to one of those locations. The location you write to is one of the locations you read from. The register file reads asynchronously (no clock) and writes synchronously (on the rising edge of a clock). Figure 10 shows a black box diagram for the register file.

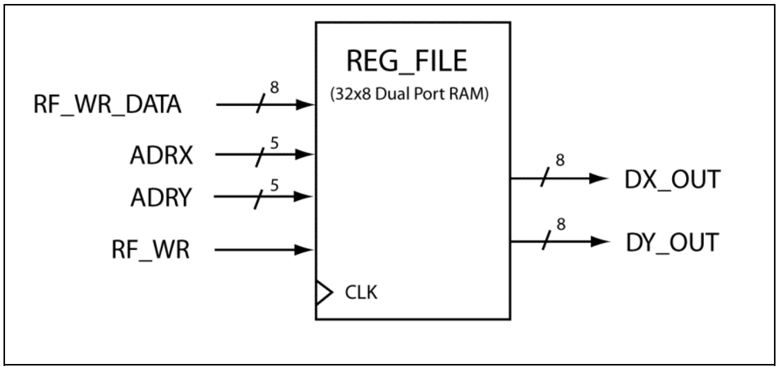


Figure 10: Register file black box diagram.

Scratch RAM: The RAT MCU implements the scratch memory as a random access memory (RAM). The Scratch RAM serves two main purposes. First, the scratch RAM provides temporary storage of data that is accessible using the RAT instruction set. In this manner, RAT instructions transfer data between the register file and the scratch RAM. Second, the RAT architecture also uses (shares) the scratch RAM as the storage device for the stack. The RAT Scratch RAM is 256 locations deep and ten bits wide (256x10). Data storage on the scratch RAM only uses eight of the ten bits while address storage uses all ten bits (prog_rom memory addresses). The RAT MCU reads the scratch RAM asynchronously and writes to it synchronously (rising edge of clock).

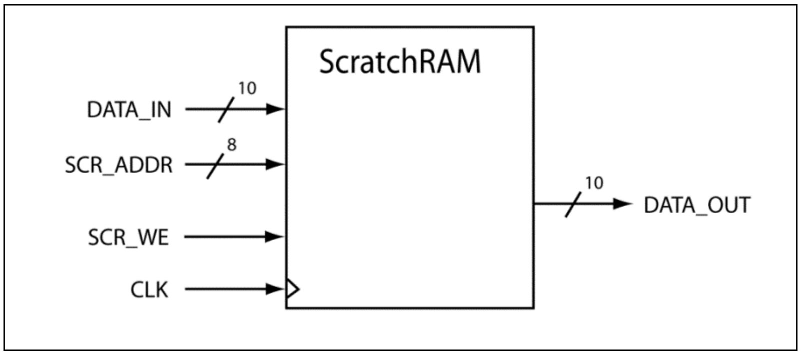


Figure 11: Scratch RAM black box diagram.

Assignment:

Configure a dual port RAM to the same storage size as the RAM in Experiment 1. Redo Experiment 1 to use a RAM in place of the single port RAM. It is not permissible to simply drop in the dual-port RAM to Experiment #1's circuitry; you must use the full functionality of the dual-port RAM for this experiment. Use the provided dual-port RAM model as a starting point; modify it to fit the parameters of this experiment. Demonstrate your working circuit to the instructor.

Programming Assignment:

Write a RAT assembly language subroutine converts a three-digit binary coded decimal value to binary value. As indicated below, the 100's digit of the decimal value is stored in the lower nibble of r21, while the 10's and 1's digits are stored in the upper and lower nibbles of r20, respectively. The CDC (Center for Disease Control) assures you that the given decimal number is not greater than 255. Place the final binary number in register r31. Minimize the number of instructions in your solution.

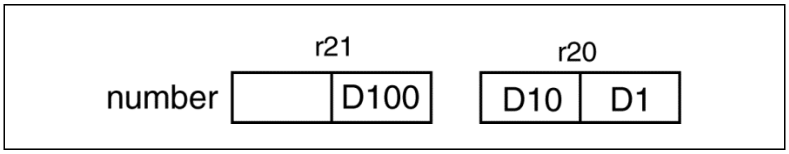


Figure 12: Diagram for programming assignment.

Hardware Design Assignment:

Show a block diagram that you could use to solve the following problems. Also include a state diagram that describes the operation of an FSM you could use to obtain the requested result. For this problem, do not use modules other than standard digital modules with typical inputs and outputs (counters, registers, shift register, comparators, RAMs, MUXes, RCAs, and decoders). If you use a decoder, be sure to provide an adequate model for it. Minimize the amount of hardware you use in your design including bit-widths of various modules.

- Design a circuit that, upon the pressing of a button, sums the values in a 16x8 RAM and stores the results in a two 8-bit registers. Consider the contents of RAM to be 8-bit unsigned binary numbers. Use no more than one 8-bit RCA for this problem.

Questions:

1. Semiconductor memory has three types of signals. State these signal types and briefly describe each one.
2. Briefly but completely describe how your circuit for this experiment operates. Don't wimp out here and don't simply describe your FSM on a state-by-state basis.
3. If the dual-port RAM you used in this experiment could store two different values at the same time, could you implement this circuit in a fewer number of states. Relate your answer to the state diagram for this experiment and support your answer with intelligent commentary.
4. The RAT MCU register file is 32x8 while and the scratch RAM is 256x10. The programmers typically use the scratch RAM to store intermediate data values. Why is this the way it's "usually done"? In other words, why not just make the dual-port RAM 256x8 so you would never need to store values in the scratch RAM? Briefly but completely explain.
5. Describe a situation in a computer architecture where you would require a three-port RAM (reads three addresses, writes one); support your answer with an example assembly language instruction (that you make up) that would require a three-port RAM.
6. Provide a diagram of a digital circuit that you could use to model a 4x10 RAM using four ten-bit registers and some supporting circuitry. For this problem, consider the registers to have "chip selects" that turn on the RAMs. Note that when the RAM is turned on, it drives a common output bus shared by all the RAMs. Don't use a MUX in your solution.
7. Currently, the RAT MCU hardware uses Y output of the register file as a possible address to the scratch RAM. Which instructions (and forms of those instructions) actually use this connection?
8. Relative the previous problem, how would I modify the hardware to have either the X value or Y value act as an address to the scratch RAM. For this problem, your RAT MCU hardware should not have anything "bigger" than a 4:1 MUX.

Deliverables

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.

2. A complete solution to the Programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness and neatness count.

4. Redone Experiment 1 circuit using a dual-port RAM

    a. Black box diagram for your circuit showing all modules and their interconnections

    b. VHDL code for FSM model only

    c. Demonstrate your working circuit to the instructor

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #6:

The Arithmetic Logic Unit (ALU)

Learning Objectives:

- To utilize the relationship between the Arithmetic Logic Unit's (ALU) requirements and the ALU-based RAT MCU instructions it supports
- To design a basic ALU that supports the RAT MCU bit-crunching operations
- To utilize both VHDL variables and signals in a VHDL model

General Notes: The ALU is part of the RAT MCU's central processing unit (CPU). As with many computers, the ALU is responsible for performing all "number crunching" operations including those falling outside of the label of "arithmetic" or "logic". In short, the ALU is responsible for performing the bit-crunching operations required by many of the RAT MCU instructions. The ALU is a relatively complex device, but as you'll see in this experiment, the power of VHDL can successfully manage this complexity using behavioral modeling. This experiment requires that you use VHDL variables in your design; these are slightly different from signals so you must understand these differences.

Assignment:

1. Implement the 15 RAT MCU arithmetic and logic functions in an ALU. Refer to the RAT MCU Assembler manual for an explanation of each of the associated RAT MCU instructions. Figure 13 shows the black box diagram for the ALU; Table 4 shows the SEL values you should use for the RAT MCU instructions that require the ALU. You must use at least one VHDL variable as part of your ALU model; do not use a RCA in your design.

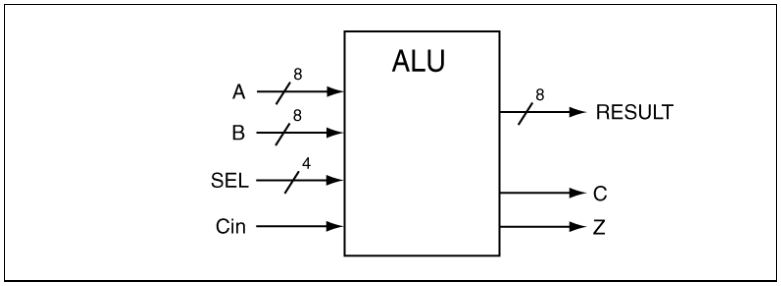


Figure 13: Black box diagram for the RAT MCU's ALU.

SEL	Instruction		SEL	Instructions
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	not used

Table 4: Selection table for the RAT MCU ALU.

2. Complete the test cases in Table 5 and include it with your lab report. Note that the ALU always has a "result" for all operations, as it is simply an output. If you can't ascertain form the given information how an instruction affects the C or Z flag, place an "X" in the respective column. Consider each line in Table 5 to be independent of previous lines.
3. Verify the proper operation of your ALU by verifying that your hand calculated results from Table 5 match the results from the simulator output. This means you must include all the operation in Table 5 in your testbench. Make sure you test the operations in the simulator in the same order they appear in Table 5 and confirm that they match your hand calculations from the previous step.

Instruction	Arguments (A, B, Cin)	Expected Result (hex)	Result-based C_FLAG	Result-based Z_FLAG
ADD	0xAA, 0xAA, 0			
ADDC	0xC8, 0x37, 1			
SUB	0xC8, 0x64, 0			
SUBC	0xC8, 0xC8, 1			
CMP	0xAA, 0xFF, 0			
CMP	0xAA, 0xAA, 0			
AND	0xAA, 0xCC, 0			
OR	0xAA, 0xAA, 0			
EXOR	0xAA, 0xAA, 0			
TEST	0xAA, 0x55, 0			
LSL	0x01, 0x12, 1			
LSR	0x81, 0x33, 0			
ROL	0x01, 0xAB, 1			
ROR	0x81, 0x3C, 0			
ASR	0x81, 0x81, 0			
MOV	0x50, 0x30, 0			

Table 5: Instructions to implement and verify with testbench.

Hints:

1) There are a few examples of ALUs modeling using VHDL in the textbook. These will be a good starting point for this experiment.
2) Keep your code as generic as possible. Your ALU can generate any Z and C value it wants; but those values don't necessarily have to be saved in the C and Z registers (the C and Z flags are implemented as flip-flops in a future experiment). The ALU model can fit on one page if you "take the cool approach".
3) Keep in mind that you can assign a value to the C & Z flags in the ALU but not have the control unit latch those values into the actual C & Z flags.

Programming Assignment:

Write a RAT assembly language subroutine that converts an 8-bit unsigned value in register r25 to a three-digit decimal value stored in registers r21 and r20. The result should be in BCD format; Figure 14 shows the weighting associated with the result. Be sure to clear the upper nibble of register r21. Minimize the number of instructions in your solution.

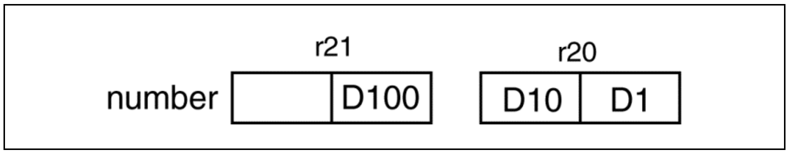


Figure 14: Number format for programming assignment.

Hardware Design Assignment:

Add instructions that do barrel shift. The form of these instructions are "**`BSL rx,ry`**" (barrel shift left) and "**`BSR rx,ry`**" (barrel shift right). For these instructions, rx is the register to shift and ry holds the number of shifts. The three LSB's of the ry register can be either 2, 4, or 6 for barrel shifts of two, four, or six in the respective directions. If the value of ry is not 2, 4, or 6, the instruction does not alter the rx register (meaning, don't mess with ry to fit this instruction in the ALU). All barrel shifts automatically insert 0's into the rx register. The C flag should set to indicate whether the value in ry was valid (ry = 2, 4, or 6) or not. Your solution should not use more than the two LSB of the ry value.

For this problem, modify your VHDL models for the ALU to include these two instructions. Do what you need to do to fit these two instructions into the current 15 ALU-based instructions. Don't increase the number of ALU selection bits for this problem. In the comments of your VHDL code, describe what you did to fit these two instructions in the ALU without changing the width of the SEL signal. No need to simulate your VHDL models. You only need to show the code for these instructions and a description of how you fit these two instructions into the ALU without increasing the width of the ALU_SEL signal.

Questions:

1. Briefly describe at least two limitations of the RAT ALU.
2. Briefly but explicitly describe how the CMP instruction allows the assembly language programmer to determine “less than”, “greater than”, and “equal to” relationships between the instruction’s operands.
3. The TEST instruction is massively useful as it allows you to do certain functions using fewer instructions. If the TEST instruction did not exist, show the RAT instructions that would be required to perform the same operation as the TEST instruction.
4. In general, adding new instructions to an instruction set requires that you also provide extra hardware to support that instruction. Did adding instructions such as TEST or CMP increase the amount of hardware required by the RAT ALU architecture. Support your answer with brief but adequate explanation; *make sure you can back your answer with actual evidence*.
5. The RAT MCU uses the TEST and AND instructions for bit masking. Describe how you would implement an AND-type bit masking using the RAT MCU instruction set but without using the TEST or AND instruction. For this problem, you need to “clear bits” using the OR instruction.
6. Briefly describe a scenario in which you would you utilize the ADDC vs. the ADD assembly language instruction.
7. Write an assembly language subroutine that would implement an ADDC instruction functionality but without using the ADDC instruction. For this problem, add the two 8-bit unsigned values in register r0 & r1; store the result of this calculation registers r31 & r30, where r31 is the most significant byte. Thus, the r31 & r30 pair represents a valid 9-bit result using the 16 bits in the two registers.
8. Since the AND instruction does not use the C flag, you can use it for something of your choosing. For this problem, show the VHDL code for the AND instruction if the C flag is used to describe the parity of the result; assume C = ‘1’ indicates odd parity. Don’t use any type of VHDL loop construct in this solution. Make sure your solution can calculate parity in one clock cycle.
9. The particular form of the LSL and LSR instructions necessitate the need for the CLC and SEC instructions. Briefly but completely describe why this is so.
10. Briefly describe why there is no need for instructions that do nothing other than alter the value of the Z flag.

Deliverables:

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answers if it makes your answer clearer.

2. A complete solution to the Programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness counts.

4. VHDL source code for your ALU

5. A completed Table 5 (do the operations and include the correct values where possible)

6. An annotated simulator printout that includes each of the instructions in Table 5. Do not include your VHDL testbench model.

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #7:

It's Alive!

Learning Objectives:

- To implement the basic functionality and implementation (FSM) of a MCU control unit
- To implement a basic fetch/execute instruction cycles for the RAT MCU
- To actuate the appropriate control signals for a the RAT MCU's instructions
- To implement your RAT MCU on the development board

General Notes: The purpose of this experiment is to assemble the various RAT MCU modules from the previous experiments into a working computer. After adding some special interface functionality, you will then synthesize this computer, download it to your development board, and run the provided program. Though this computer will be able to execute an actual RAT MCU assembly language program, it will only use a limited subset of the RAT MCU instructions. This computer will be a subset of the final RAT MCU, as you'll be leaving out several important modules in order to reduce the scope of this experiment; you'll add those modules in a later experiment.

This experiment has many important aspects, none of which is by any means trivial. While previous experiments involved assembling individual RAT modules, this experiment requires a true systems-level approach in order to successfully complete the assignment. Understanding the RAT MCU at a systems level, including the interface to actual hardware, should provide you with a complete understanding of the internal workings of the RAT MCU and basic computers in general.

Assignment:

1) Assemble the program fragment in Figure 15 (it's a fragment because it is missing proper file banners, comments, etc.) using the RAT simulator and add the resulting prog_rom.vhd file to your Xilinx project.

```
;-----------------------------------------------------------------
;- I/O Port Constants
;-----------------------------------------------------------------
.EQU SWITCH_PORT = 0x20      ; port for switch input
.EQU LED_PORT    = 0x40      ; port for LED output
;-----------------------------------------------------------------
.CSEG
.ORG 0x01

main:  IN    r10,SWITCH_PORT
       MOV   r11,0x01
       SUB   r10,r11
       OUT   r10,LED_PORT
       BRN   main
```

Figure 15: Test program for this experiment.

2) Enter the appropriate values for each CU output signal for the individual instructions appearing in Figure 15 in the execute state of the combinatorial process.

   NOTE: The control unit is an FSM that provides control signals to the various RAT MCU modules. Recall that all of the RAT MCU modules have various control inputs; these inputs are the outputs of the control unit. The control unit provides the proper sequencing of module operations to create the RAT MCU. The structure of the control unit template model schedules the assignment of all control outputs to a "non-action" state when the process combinatorial process initiates. Despite knowing that these values are zero, reassign these values to zero for the individual instructions if that instruction uses that control signal. Do not include control signal assignments with the instruction implementation that are not associated with that instruction.

3) Figure 16 shows the black box diagram for the RAT_MCU module, which contains all the components you've made so far. Use the provided RAT MCU architecture diagram to complete the VHDL code that models the RAT MCU architecture. Note: the minimum modules you'll be port mapping for this lab include: the control unit, the program counter, the prog_rom, the register file, two flag registers, and the ALU (see Figure 17). Implement the carry and zero registers using the provided flag register VHDL template. The flag registers are flip-flops; the data inputs to these flip-flops are the associated outputs from the ALU; the flip-flops outputs are inputs to the Control Unit.

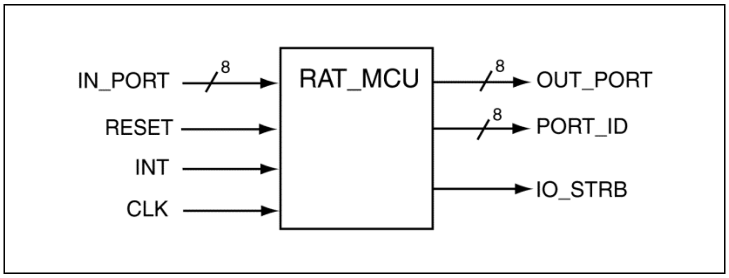


Figure 16: Black box diagram for RAT MCU.

4) Create a test bench to test the operation of your RAT MCU with the simulator using the program listed in Figure 15. If the simulator shows that you do not have the correct output from the given program, consider watching the provided video on how to find and fix errors in your design. The idea behind fixing your errors is that you know what the program is doing and you know how the

RAT MCU hardware should be handling the given instructions in the program. The best approach is to find an incorrect value and trace it backwards to find the source of that error.

5) Write the implementation constraints file for the wrapper. Synthesize, map, and generate the programming file for your entire project and download the bit file to your development board. Demonstrate the correct operation of your RAT MCU to the instructor.

   NOTE: The RAT Wrapper provides the interface between the RAT MCU and the development board. Study the provided RAT_Wrapper.vhd file to understand how the RAT MCU interfaces (handles inputs and outputs) to the development board. Note that the switch and LED constants in the RAT wrapper match the port_IDs associated with the IN and OUT instruction in the provided program. The RAT_Wrapper file is the top module to your project; the RAT MCU is a sub-module of the RAT wrapper.

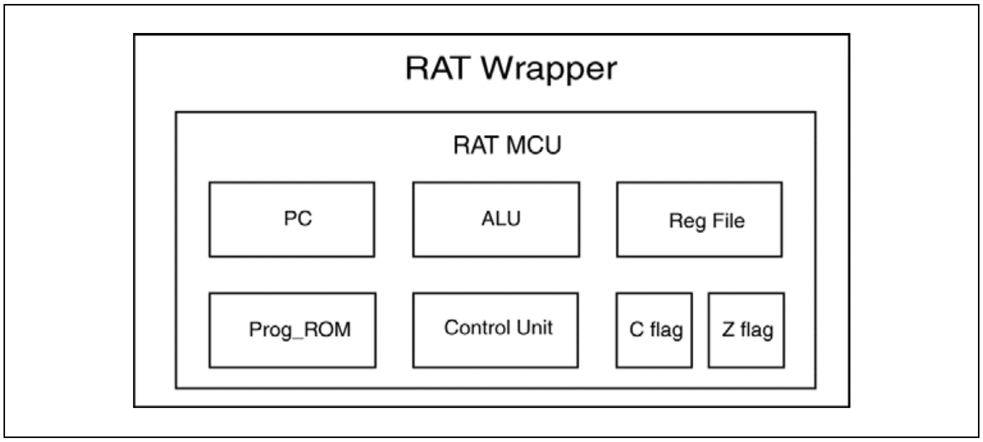


Figure 17: Overall high-level black box diagram for Experiment 7.

Other Important Comments:

There are many areas for problems in this experiment. Make sure you read over the “Troubleshooting Ideas” document before you call someone over for help. This document lists items that typically form 75% of the errors people make that prevent them from completing this experiment.

The RAT MCU uses both 7-bit and 5-bit opcodes. There are several valid approaches to modeling these opcodes; the partial VHDL model presented as part of this lab activity uses one of those methods. Figure 18 shows a fragment of code from the provided control unit template file. This code shows the opcodes for two instructions: the SUB (reg-reg) and the IN (reg-immed). The code for these instructions is different because the length of the opcodes for these two different instruction-types is different: the reg-immed instructions have five opcode bits while all other instruction types have seven opcode bits.

The control unit template model takes of the approach of treating the five-bit opcode as a seven-bit opcode for decoding purposes. Note the first five bit in the opcode of the IN instruction as always the same for each of the four different opcodes listed in the IN instruction, while the last two bits form a two-bit binary count (00→01→10→11). Since reg-immed instructions have five-bit opcodes, the control unit does not use the last two bits to decode the instruction. These two bits could have been listed as “don’t cares”, but a golden rule of VHDL is to never use don’t cares in your model.

```
        -- SUB reg-reg  --------
              when "0000110" =>

        -- IN reg-immed  ------
              when "1100100" | "1100101" | "1100110" | "1100111" =>
```

Figure 18: Code fragment from provided control unit template.

Programming Assignment:

Write a RAT MCU assembly language subroutine that counts the number of bits that are set in register r10 and r11. If both registers have the same number of bits set, clear both registers. Otherwise, replace the value in the register with the higher number of set bits with the number of bits that are set in that register (don't alter the other register). For example, if four bits are set, place 0x04 in that register. Minimize the number of instructions you use in your solution.

Hardware Design Assignment:

You decided that you need a NBSWP instruction. This instruction swaps the nibbles of the source register. The precise form of this instruction would be "NBSWP **r22**".

a) describe the changes you need to make to the RAT MCU hardware to support this modification

b) describe any changes you need to make to the RAT assembler to support this modification

c) state any changes in RAT MCU memory requirements resulting from this modification

d) state how this instruction could be useful

Device	Size	# bits
prog_rom	1024 x 18	18432
register file	32 x 8	256
program counter	10	10
stack pointer	8	8
flags(I,Z,C,shZ,shC)	5 x 1	5
scratch RAM	256 x 10	2560
	TOTAL	**21,271**

Questions:

1. How many different instruction formats does the program in this experiment use? Make a list of the instruction and their formats for this question.
2. Briefly but completely describe the relationship between the IN and OUT instructions and the port_IDs assigned in the RAT Wrapper VHDL model.
3. This experiment implements the C and Z flags as flip-flops. Briefly describe why the RAT MCU architecture requires that these two flip-flops have different control features.
4. This experiment asked you to include only the control signals that a particular instruction used regardless of whether they were previously ***scheduled to be assigned to zero(s).*** Briefly state why this approach represents excellent VHDL coding style.
5. How much memory does the control unit use in this experiment contain? Also, state which signal represents that memory.
6. In assembly language-land, we refer to instructions that do nothing as “nops” (pronounced “know ops”). In academia, we refer to “nops” as administrators. Many assembly language instructions actually have a dedicated nop instruction, but the RAT MCU does not. There are two approaches to faking a nop instruction in VHDL using the existing RAT MCU instruction stet. For this questions, list those two approaches. HINT: one of these approaches involves a branch instruction.
7. Describe a situation where a NOP instruction or a NOP-type instruction would be useful.
8. Briefly describe the differences in how the control unit code handles 5-bit opcodes as opposed to 7-bit opcodes.
9. Remember those setup and hold times? Whatever happened to those? Is this something we should be worried about in this Experiment? Briefly explain.

Deliverables:

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.

2. A complete solution to the Programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness counts.

4. An annotated simulator printout showing correct functionality of your RAT MCU for the test program. At the very least, show that you can read in a value from the switches and output one less than that value to the LEDs on the development board (according to the provided assembly language program). Be sure to include the PC output, the prog_rom output, and the IO_STRB output in your simulator timing diagram.

5. VHDL source code for the control unit module

6. Instructor signature indicating a working experiment

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #8:

Full-Boogie MCU – Get 'er Done

Learning Objectives:

- To implement the RAT MCU instructions utilizing the scratch RAM
- To implement stack-oriented instructions to control scratch RAM access
- To implement all remaining non-interrupt oriented RAT MCU instructions
- To debug and test a complex digital system

General Notes: Most every MCU design contains both the notion of a stack and an "other" generic memory to store data under program control. Many MCUs, such as the RAT MCU, use the same physical memory device (the scratch RAM) for both the stack for generic data storage. The overall purpose of the RAT MCU's scratch RAM is to store information vital to the operation of the RAT MCU.

The notion of "stack oriented instructions" includes instructions that modify both the scratch RAM and stack pointer. Memory oriented instructions use scratch RAM without modifying the stack pointer as part of generic data storage operations under program control. Table 6 lists the instructions associated with and/or using the scratch RAM.

Stack-Oriented Instructions (stack pointer tweakers)	**Memory-Oriented Instructions** (non-stack pointer tweakers)
`WSP`	`LD (register indirect)`
	`ST (register indirect)`
`CALL`	
`RET`	`LD (immed indirect)`
`PUSH`	`ST (immed indirect)`
`POP`	
`RETIE (Experiment #9)`	
`RETID (Experiment #9)`	

Table 6: RAT MCU Instructions that use the Scratch RAM.

The scratch RAM in the RAT MCU is a memory device that can of store two types of data. The RAT MCU needs the ability to store 8-bit values data from memory-type instructions (data values) and 10-bit data from stack-type operations (address values). The ST (store) or PUSH instructions write 8-bit data values to the scratch RAM from registers; the LD (load) or POP instructions read 8-bit data from the scratch RAM and write the data to registers. The 10-bit values represent the address of instructions in program memory; CALL instructions write these addresses to scratch RAM and return-type instructions (RET, RETIE, & RETID) read these addresses from scratch RAM and copy them to the program counter. While LD and ST instructions do not involve the stack, the PUSH/POP and CALL/RET instructions utilize the stack. Using the scratch RAM for both stack and generic data storage operations creates extra management work for the programmer because the programmer is responsible for maintaining the integrity of the stack while having the program operate properly.

The key to stack integrity starts with the stack pointer. The RAT MCU uses the stack pointer as a storage device to hold the address of the current "top-of-the stack". Recall that the top-of-the-stack is address of the most-recent item stored on the stack. Thus, the stack pointer is a register that contains the address of the most recent data stored on the stack. Integral to the proper stack operation is to place the stack pointer in a known state, or in other words, intentionally place a value into the stack pointer; the RAT MCU uses the WSP (write stack pointer) instruction for this purpose. The result is that the initialization portion of every viable RAT MCU assembly language program necessary contains a WSP instruction. The MCU developer may also use the reset signal to initialize the stack pointer to zero, ensuring that a known the stack pointer value upon system reboot, but many embedded systems contain no such "magic" reset-ability.

The RAT Assembler Manual states that the stack pointer is pointing to the most recent item stored on the stack, or the "top of the stack". You can ascertain this approach by examining the RTL equations associated with various stack-oriented instructions. For this description, we'll examine only the PUSH and POP instructions, which we highlight in Table 7.

The implementation of the PUSH instruction involves decrementing the stack pointer. Though it may seem strange, the fact is that most computer architectures point the stack pointer to some relatively high memory address and use decrements as part of the storage process (and increments as part of the retrieval process). In this way, the stack "grows" from higher memory addresses towards lower memory addresses, staying as far away as possible from memory that may have been defined as part of the start-up code or under program control (using LD-type instructions). This provides the opportunity for the stack to be as large as possible given the memory constraints provided by programmer-initiated data definitions (assembler directives). Once again, maintaining stack integrity is part of the programmer's responsibility; programmers need to understand the software tools (primarily the assembler) and the underlying hardware (scratch RAM usage) in order to ensure stack integrity.

PUSH		**POP**	
RTL	**Description**	**RTL**	**Description**
(SP-1) ← Rd, SP ← SP - 1	Register data is written to scratch RAM at one less than the address pointed to by the SP. The SP is simultaneously decremented.	*Rd ← (SP), SP ← SP +1*	Data from the scratch RAM at the address in the SP is written to a register. The SP is simultaneously incremented.

Table 7: A low-level description of PUSH & POP instructions.

Assignment:

1. Create your stack pointer module according to Figure 19. The best starting point for this step is the generic counter template.
    - RST is synchronous signal that resets the stack pointer
    - LD is a synchronous control signal that enables the loading of input data to the stack pointer. When LD is '1', the stack pointer loads the DATA input value.
    - INCR signal synchronously increments the stack pointer when asserted
    - DECR signal synchronously decrements the stack pointer when asserted

    Keep in mind that if neither RST, INCR, DECR, or LD is asserted, the stack pointer does not change state. The precedence of these signals does not matter as you'll design your control unit to never assert more than one of these signals at a time. Reset inputs generally have precedence over load inputs, and load inputs have precedence over the increment and decrement inputs.

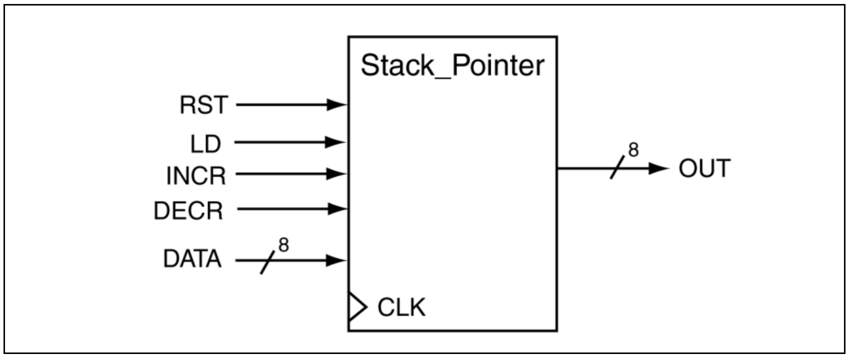


Figure 19: Black box diagram for the Stack Pointer.

2. Add the stack pointer and scratch RAM to your RAT MCU. Recall that you created the scratch RAM in a previous experiment. Implementing this step includes adding MUXes for address and data selection into the scratch RAM. Be sure to adjust the RAM's memory attributes to support the RAT MCU's architectural requirements.
3. Complete your control unit by including the control values for most of the remaining RAT MCU instructions. For this step, the only RAT instructions that you won't implement are the ones dealing with interrupts (SEI, CLI, RETID, & RETIE).
4. Demonstrate the running of clean_testall.asm program on your RAT MCU implementation to your instructor.

Special RAT MCU Testing Requirements: You can't overstate the importance of ensuring that your RAT MCU is working properly. Accordingly, there a test program that verifies *most* of your instructions are operating properly: clean_testall.asm. This program provides a means to visually verify your hardware is working correctly. If your program passes the visual test, you probably implemented your hardware correctly; otherwise, you'll need to delve down into the guts of the assembly language program in order to ascertain what instruction caused the test to fail, and/or debug the program in the simulator. This is potentially an experiment that requires a significant amount of debugging. You can find a more complete description of the clean_testall.asm program in the program's file header. There is also a video of a working clean_testall.asm linked on the course website.

Here's the initial approach you should take to debugging your hardware (though you can take any approach you feel is appropriate).

1. Revisit the troubleshooting document provided in the previous experiment and verify your models contain none of the listed common errors.
2. Make sure your hardware properly synthesizes. You don't necessarily need to simulate the hardware in this step, but you may want to start a simulation to compile your code using the simulators's compiler, which often does a better job of finding errors.
3. Download the clean_testall program to the hardware. You should have the number "0" light up in each digit of the 7-segment displays, one at a time. If you don't see this, you probably have an error in your CALL and/or RET instructions.
4. If you don't see the numbers lighting up in the previous item, it's time to use the simulator. Before you do this, there are two delays in the clean_testall.asm program. You must swap the names of these delays in order for you to use the shorter delay, which allows you to inspect your timing diagram without scrolling through a major amount of the simulation.

Programming Assignment:

You decided that you need another stack, but your only option is to create a firmware-based stack. Using r31 as a "stack pointer", write "push" and "pop" subroutines that would allow you to simulate a stack starting at scratch RAM address 0xD0. Assume register r30 will be the register you use to pass data to your stack (the "push" subroutine) and from your stack (the "pop" subroutine). Don't allow these subroutines to permanently change any register values other than r30 and r31. Also for this problem, state where these subroutines could be useful.

Hardware Design Assignment:

You decided that you need a CPYCZ instruction (copy C & Z flags to register). The precise form of this instruction would be "**`CPYCZ      r23`**". This instruction copies the values of the C & Z flag to the two LSBs of the given register. This instruction does not change the current flag values.

a) changes you need to make to the RAT hardware

b) changes you need to make to the RAT assembler

c) changes in RAT MCU memory requirements

d) why this modification would be useful

Device	Size	# bits
prog_rom	1024 x 18	18432
register file	32 x 8	256
program counter	10	10
stack pointer	8	8
flags(I,Z,C,shZ,shC)	5 x 1	5
scratch RAM	256 x 10	2560
	TOTAL	**21,271**

Questions:

1. Rewrite the RTL equations of Table 7 to reflect the notion of a PUSH incrementing the stack pointer and a POP decrementing the stack pointer. Additionally, add intelligent comments on whether you feel there is an advantage to doing it one way or the other.
2. The current RAT MCU hardware supports a bunch of instructions. For this question, make up your own instruction without changing the existing hardware, other than the control unit to send out the proper control signals to support your new instruction. Your answer to this problem should be similar to how the control unit models an instruction, including a valid opcode and appropriate control signal assignment.
3. We refer to the scratch RAM as a “higher level” of memory compared to the register file. Briefly describes what this means in the context of the RAT MCU.
4. In the context of using an FPGA to implement your RAT MCU, briefly but completely describe whether you can be sure of the value of the stack pointer if you don’t specifically initialize it using a WSP instruction.
5. In this lab activity, you implemented the stack pointer as a counter. Provide a diagram of the supporting hardware if you had to implement the stack pointer as a loadable register.
6. Write a RAT assembly language subroutine that swaps the data in two locations in scratch RAM. Registers r30 and r31 contain the addresses in scratch RAM of the two locations that require swapping. Make sure the subroutine does not permanently change any register values; don’t use an EXOR instruction in your solution.
7. Write a RAT assembly language subroutine that initializes all scratch RAM memory locations to 0x00. Don’t use PUSH instructions in your solution.
8. Write a RAT assembly language subroutine that initializes all scratch RAM memory locations to 0xFF. Don’t use ST instructions in your solution.
9. The previous two questions were clever, but not realistic. Briefly describe why are these problems not 100% doable in real life.
10. Write a RAT assembly language subroutine that swaps the values in two different registers without accessing memory or changing the value in any register other than the two registers in question. Do not use more than five instructions in your solution.
11. Repeat the previous problem using only shift-type instructions, but use as many instructions as you need to solve the problem.
12. Briefly but completely describe the changes to the RAT MCU architecture required if you were to implement the stack and data storage RAM as separate modules; draw a schematic to help describe your approach.
13. Briefly but completely discuss the advantages, if any, of implementing the stack and generic storage RAM as separate modules.

Deliverables:

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.

2. A complete solution to the programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness counts.

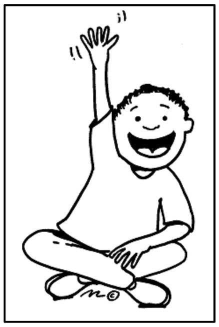
4. VHDL source code for your stack pointer

5. A signature from the instructor showing that the test_all program worked properly.

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #9:

Interrupts



Learning Objectives:

- To implement the interrupt architecture on the RAT MCU hardware
- To learn how to simulate interrupt operation on the RAT MCU

General Design Notes: Interrupts: If you're like most humans, you occasionally need a haircut. It would be a strange world if the person who cuts your hair called you every five minutes and asked you if your hair needs cutting. Naturally, a better approach (more efficient? less annoying?) would be that when you needed a haircut, you simply call the person who cuts your hair and schedule an appointment. Requesting some type of service is the general approach humans take in most facets of their lives (unless you work in the sales where you're required to continually ask others if they want service).

Not surprisingly, an analogous situation exists in microcontroller applications. Programs you write generally do something, *i.e.*, they execute some finite number of relatively useful tasks. The two approaches to implementing tasks are analogous to the example above: you either constantly check to see if tasks need doing and perform them if they do, or you perform those tasks only when they "ask you" to perform them (generally, they'll only ask when the task needs doing). Microcontroller lingo refers to the act of constantly asking if a task needs attention as *polling*. We refer to "handling" these tasks only when the task requests service as *interrupt driven*, or "*real-time",* and thus require the use of the MCU's *interrupts*.

This experiment introduces some of the theory and concepts behind implementing the hardware that support writings assembly language programs that utilize interrupts. Because of the simple nature of the RAT MCU's interrupt architecture and the fact that an in-depth study of interrupt-driven systems is beyond the scope of this experiment, the approach taken in this experiment only touches upon the basics of interrupt handling. Please refer to the course text for further details.

Polling is a relatively simple concept but it does have one large drawback: it's inefficient to continually ask a device if it needs something when the device has nothing that needs doing. In terms of MCU processing, if the MCU is polling, it is not doing something else that could be potentially more important (as in time critical). The result is that you lower the overall throughput of your system if you're wasting clock cycles in a polling loop. Once again, a more efficient approach in terms of MCU processing is to allow individual circuit elements that occasionally need attention from the MCU to have those circuit elements directly request processing from the MCU. The notion of a hardware interrupt provides a mechanism for such a request. Note that it's comfortable to say polling is bad, but in reality, it's only bad if the processor has something more important to do. In real life, your MCU may be idle sometimes when nothing needs doing; during those times, you can consider polling acceptable. Thus, there are gray areas in this discussion.

The term *interrupt* comes from the fact the normal operation of the microcontroller is temporarily *interrupted* to handle some other task. Once microcontroller handles the other task, the microcontroller returns to the task it was executing when it received the interrupt. Though microcontrollers in general use three types of interrupts (internal interrupts, external interrupts, and software interrupts), the RAT MCU only handles one external interrupt. Unfortunately, there is no single method used by all microcontrollers to handle interrupts, so examining the interrupt architecture is one of the first things you typically do when working with a new microcontroller.

We refer to the interrupt handling mechanism in the RAT MCU as a *vectored interrupt*. Once the RAT MCU receives an interrupt, program control transfers to a pre-determined address in instruction memory after execution of the current instruction completes. We refer to this address as a *vector address*, and the instruction at this address generally contains a BRN instruction (unconditional branch). The instruction branched to is the starting address of the *interrupt service routine (ISR)*. The code in the ISR generally implements some task that appropriately *handles* the interrupt, which is why people sometimes refer to ISRs as *interrupt handlers*. When the MCU finishes executing the ISR, the MCU returns program control to the instruction following the instruction the MCU was processing when it detected the interrupt.

The concept of an interrupt is essentially provides a method for hardware to call a special subroutine: the *interrupt service routine*, or ISR. When the RAT MCU receives an interrupt, part of the automatic response of the hardware is to prevent the RAT MCU from acknowledging any further interrupts until it completes processing of the current interrupt (the ISR is executed). We refer to the act of preventing the RAT MCU from processing interrupts as *interrupt masking.* Remember, preventing the RAT MCU from acting on interrupts does not prevent the interrupts from happening; it only allows the RAT MCU to ignore any occurrence of an interrupt. You can use the SEI instruction (set interrupt enable) to allow the RAT MCU to unmasking interrupts or the CLI instruction (clear interrupt enable) to mask interrupts.

The concept of ISRs is similar to a simple subroutine. The big difference is that programs "call" subroutines under program control (firmware) while ISRs are "called" under external hardware control. The ramifications of this difference is that ISRs are asynchronous in that they can happen anytime, while subroutines are under program control and are thus synchronous.

The RAT MCU Interrupt Architecture: 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 subroutine CALLs and RETs, which makes interrupt processing simply another form of flow control management in the RAT MCU. When the RAT MCU senses that the signal attached to its interrupt input is asserted and the interrupts are currently unmasked, it completes execution of the current instruction before it starts processing the interrupt. The first step in the processing of the interrupt is to place the *vector address* in the program counter, which forces the RAT MCU to next execute the instruction at location of the vector address. The RAT MCU hardwires the vector address to 0x3FF. At the same time, the RAT MCU pushes the address of the next instruction that would have been executed had it not acted on an interrupt onto the stack. In addition, the RAT MCU simultaneously "saves the context" of the RAT MCU by saving the current values of the carry and zero flags into the "shadow" carry and zero flag registers.

Processing the ISR begins with the RAT MCU acting on an interrupt 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 PC; this instruction should be the address of the next instruction that would have been executed had the MCU not acted on the 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 into the actual carry and zero flags. Figure 20 provides a summary of these steps in a rough order of occurrence (many items occur simultaneously).

- The RAT MCU detects an asserted signal on the interrupt input (assume the RAT MCU has not masked the interrupt)
- Execution of the current instruction completes and the RAT MCU goes into an *interrupt cycle*
- The RAT MCU automatically mask the interrupt so it will ignore further interrupts
- Address of next instruction (not in ISR) is stored on the stack
- The program counter is loaded with 0x3FF
- The present state of the RAT MCU is saved (zero and carry flags are copied to shadow flags)
- The instruction at location 0x3FF is executed (a branch instruction that directs program control to the address of ISR)
- Execution of the ISR begins.

- Execution of the ISR completes
- The program issues a RETIE or RETID instruction
- The RAT MCU pops the address of the next instruction to execute off the stack and places it in the program counter
- The context is “restored” by copying the shadow carry and zero flags to the actual carry and zero flags
- Execution resumes at the instruction following the one that was executing when the RAT MCU got the interrupt

Figure 20: An outline of the RAT MCU interrupt architecture.

<u>General Design Notes: The RAT MCU Control Unit Modifications for Interrupts:</u> Interrupts on the RAT MCU are primary handled by the control unit. For this experiment, you must modify the control unit so that it contains an extra state that handles the needs of the RAT MCU’s interrupt architecture. The proper vernacular for this modification is to include an *interrupt cycle* in the current FSM. Try not to panic; Figure 21 shows the required modifications to the RAT’s control unit in order to support an interrupt cycle. Note that entry into the interrupt cycle occurs when the RAT’s INT input is asserted at the end of the execute cycle. In the INTERRUPT state, the control unit asserts the controls signals required to implement the various hardware aspects of interrupt architecture.

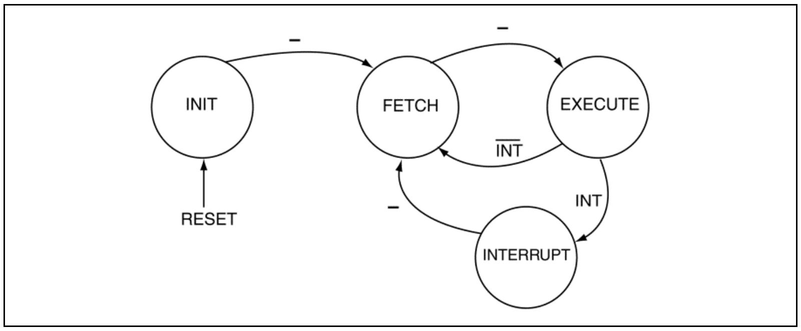


Figure 21: The RAT control unit state diagram showing interrupt cycle.

As you can gather from Figure 21, modifications to the control unit are relatively trivial due to your burgeoning expertise in the area of modeling FSMs in VHDL. Possibly the hardest part with this modification is that you’ll need to remember to add INT to the process sensitivity list of the combinatorial process associated with the FSM. Figure 22 provides you with some code to get your modification juices flowing. Make sure you place this code in the proper place in the CU model.

```
when ST_exec =>
   if(INT = ‘0’) then
      NS <= ST_fetch;
   else
      NS <= ST_interrupt;
   end if;
```

Figure 22: Control unit modification overview for the interrupt cycle.

Assignment:

1) Add/modify the following modules in your RAT MCU (consult the architectural diagram for details):

   - Modify the FLAGS module to include the shadow carry and zero flags
   - Add the hardware associated with interrupt control

2) Modify your RAT MCU such that it can process interrupts and execute the following RAT assembly instructions: RETIE, RETID, SEI, and CLI. This step includes control unit modifications for both the structure of the FSM as well as the implementation of these instructions.
3) Test your interrupt functionality in the simulator using the provided test program. Make sure you show both an interrupt cycle and a return from interrupt. Also include the C & Z, shadow C & Z, PC output, the interrupt signal, and FSM state (PS) in your timing diagram output. Be sure to annotate your output in such a way that you make it obvious to the reader that you understand the various concepts associated with the RAT MCU interrupt architecture.

Programming Assignment:

Write an interrupt driven program that blinks the LEDs. Every interrupt alternately turns on/off the LEDs. The LEDs that turn on are the ones corresponding to the most recent value on the switches. Minimize the number of instructions you use in your neatly written solution. **Constraint:** *Do not use IN instructions inside the ISR.*

```
.EQU    LEDS      = 0x74
.EQU    SWITCHES  = 0x75
```

Hardware Design Assignment:

You must modify the RAT MCU architecture in order to support a second interrupt. The second interrupt should have all the same functionality of the current interrupt. This problem is purposely written rather vaguely, which allows for many different solutions. Whatever solution you choose, make sure it is complete and consistent.

a) changes you need to make to the RAT hardware

b) changes you need to make to the RAT assembler

c) changes in RAT MCU memory requirements

d) why this modification would be useful

Device	Size	# bits
prog_rom	1024 x 18	18432
register file	32 x 8	256
program counter	10	10
stack pointer	8	8
flags(I,Z,C,shZ,shC)	5 x 1	5
scratch RAM	256 x 10	2560
	TOTAL	**21,271**

Questions:

1. Briefly but completely describe how the provided program verified the proper operation of the shadow C & Z flags.
2. List the differences between a RETIE instruction and a RET instruction.
3. Based on the state diagram in Figure 21, would it be possible to assert the interrupt signal and not enter into an interrupt cycle? Briefly but completely explain. Assume the interrupt is unmasked.
4. Interrupt architectures generally always automatically mask interrupts upon receiving an interrupt. Briefly but completely describe why this is a good approach, and a better approach than attempting to rely masking the interrupts under program control.
5. For the RAT MCU, there is only one state associated with the interrupt cycle, which means there is only one associated clock cycle. Briefly but completely describe in general what dictates how many states (or clock cycles) a given MCU requires for the interrupt cycle. Once again, for this question, consider "states" and "clock cycles" as the same thing.
6. We generally consider interrupts "asynchronous" in nature. However, looking at Figure 21, the interrupts appear to be synchronous. Briefly describe what it means for the interrupts to be asynchronous and why it is that they don't appear asynchronous in Figure 21.
7. Briefly but completely describe the major problem with the RAT MCU receiving an interrupt while it is in the act of processing an interrupt. For this problem, consider the interrupts as being masked when the MCU receives the interrupt.
8. Briefly but completely describe the major problem with the RAT MCU receiving an interrupt while it is in the act of processing an interrupt. For this problem, consider the interrupts as being unmasked when the MCU receives the interrupt.
9. Write two subroutines: Store_CZ, and Restore_CZ. The store subroutine stores the values of C & Z to r20. The restore subroutine restores the values in r20 to the C & Z flags. Don't use more than 12 instructions total for both subroutines.
10. Briefly but completely describe a situation where the previous subroutines be potentially useful.
11. For the following code fragment, what is the minimum value for "?????" to make the program happy?

```
;-------------------------------------------------------------------
;-- .DB Directives
;-------------------------------------------------------------------
.DSEG                ; we’re in the data segment
.ORG 0xC0

Roof:
Floor:   .DB 0x50, 0x61, 0x82, 0x33
         .DB 0x35, 0x36, 0x37, 0x38, 0x39, 0x4A, 0x3F
Wall:    .DB 0xAA, 0xBB, 0xCC, 0xDD, 0x39, 0x4A, 0x4C, 0x3E

.CSEG
.ORG    ?????

Main:   BRN   Main
```

<u>Deliverables:</u>

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.

2. A complete solution to the programming Assignment. Completeness counts.

3. A complete solution to the Hardware Design Assignment. Completeness counts.

4. VHDL code showing the modifications to the RAT MCU control unit for this experiment. Please don’t print the entire control unit module.

5. A simulator generated timing diagram (with annotations) showing that your interrupt works properly (described previously).

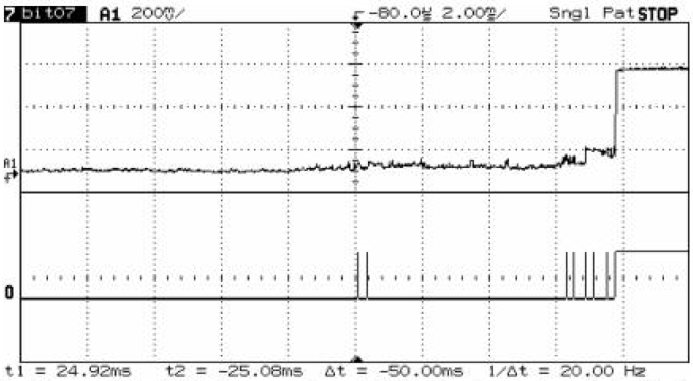
Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #10:
Switch Debouncing and Firmware Display Multiplexing in and a Real-Time Environment

<u>Learning Objectives:</u>

- To review the RAT interrupt architecture and use interrupts in an actual application
- To learn about switch bounce and debouncing requirements for external input
- To learn the concepts of seven-segment display multiplexing
- To implement a firmware look-up table on the RAT MCU

<u>General Design Notes: Switch Bounce:</u> Mechanical switches present design challenges when you use them in conjunction with "fast" devices such as MCUs. Mechanical switches tend to "bounce" when actuated; this means that when you actuate a switch (move it from the off to on position), the switch 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 effectively result in many unintentional button presses. Because MCUs typically execute instructions on the nanosecond level, hardware designers and/or firmware programmers must account for these unintentional actuations in order to ensure proper operation of their designs. Figure 23 shows an example of switch bounce switch. (source: http://www.ganssle.com/debouncing.htm).



Switch A at 2 msec/div. Note 8 msec of unsettled behavior before it finally decides to open.

Figure 23: Oscilloscope output for debounce problem.

Approaches to handling switch bounce generally fall into two categories: hardware solutions or firmware solutions. The preferred solution for the RAT MCU is to handle debounce in hardware as we have a significant amount of unused hardware resources on the development board's FPGA. Moreover, the firmware solution unnecessarily complicates your firmware, which is something you always want to avoid (particularly when we only have one interrupt available). This experiment depends on button presses, so you'll need to debounce the button in hardware in order for the experiment to work properly.

The good news is that this experiment provides you with a viable hardware switch debouncer; you're mission will be to declare it and instantiate it in your design. This device actually serves two functions: it debounces the button and provides an output with "one-shot" characteristics. The one-shot ensures

Real-Time Program Architecture: The notion of real-time programming essentially means that you're using interrupts in your program. The "real-time" label comes from that fact that if you're using interrupts, your program can react *instantly* to something in your system that requires attention. This is somewhat of an advanced concept, so we'll not go there in this course. What is important to know is that we divide the assembly code in real-time programs into two distinct parts: the main code and the ISR code, also referred to as the foreground task (the main code, the task code) and the background task (the ISR code).

You can model an interrupt driven program as the MCU executing the foreground task while it is essentially waiting for an interrupt; when it receives an interrupt, the MCU executes the code associated with the background task. The important thing to realize is that the RAT MCU is always executing some instructions; what we can officially say is that the MCU executes the main code while waiting for an interrupt. When the MCU receives an interrupt, it switches to executing the ISR. What you need to be thinking is for relatively simple programs such as the one in this experiment is what are the responsibilities of the main code and what are the responsibilities of the ISR code.

Once again, many of the details of real-time programming are outside of the scope of the course. But, the one embedded systems principle you should follow is the notion that you should keep your ISR as short as possible. Why? Because it protects the real-time nature of your program. Recall that when you receive an interrupt, the RAT MCU hardware automatically masks the interrupts. When the interrupts are masked, you can't receive any more interrupts, which means if there is some device that requires the MCU attention (such as your pacemaker), the device won't get that attention until the interrupts are unmasked under program control. Massively interesting!

Assignment:

Write a RAT assembly language program that does the following:

1) Counts the number of interrupts received by the system; a push of the left-most button on the development board should generate an interrupt.
2) Displays the number of interrupts received by the system (in decimal) on the two right-most 7-segment displays of the development board. After the interrupt count reaches 49, the display rolls over to zero and the interrupt counting continues. Be sure to employ lead-zero blanking on the 7-segment display.

Modify your RAT MCU to include the debounce/one-shot circuitry. The output of the debounce/one-shot circuitry connects to the interrupt hardware portion of the RAT MCU. Make your modifications on the RAT MCU level and not in the RAT Wrapper. Demonstrate your working program to your instructor .

NOTE: Avoid the *ghosting* effect; the code you write should drive the two 7-segment displays in a clear manner. If a display or segment appears dim, then you are driving that segment when you should not be.

To avoid the ghosting effect, make sure that your code enables the correct 7-segment displays only after you write the correct data to the display's segments.

Suggested Implementation Approach: This experiment uses a hardware interrupt, which means that there is a signal physically attached to the interrupt pin on the RAT MCU. When this signal is asserted (goes to '1' in this case), the RAT MCU enters into a preset sequence of events. Future interrupts will only be handled if the interrupts are re-enabled under software control (with either a RETIE or SEI instruction). For this experiment, you connected the left-most button (an arbitrary choice) to a debounce/one-shot circuit, which you then connected to the RAT's interrupt pin.

The general structure of an interrupt-driven program is to have the program spend most of the time in the foreground task and enter the background task when the MCU processes an interrupt. The best way to approach this experiment is to implement the foreground task, and then add the background task (the interrupt processing) once the foreground task is working properly. The foreground task you should implement is this: have the RAT MCU read from two registers; each of these registers holds a BCD number (a tens digit value and a ones digit value). Display these two numbers on two digits of the 7-segment display. You'll need to multiplex the displays in order to view both numbers simultaneously.

Once you complete the above foreground task, you then include the background task in your program. For this experiment, the background task consists of handling button presses connected to the RAT MCU interrupt. The background task tracks the number of button presses in two registers (the tens value and the ones value). In this way, the background task handles updates of the count while the foreground task handles the display of the count. Keep in mind that you should place all major functions in subroutines and the main body of your code should primarily be comprised of a series of subroutine calls.

Any time you work with interrupts, the first order of business when debugging your "unit" (the firmware and hardware package) is to verify you're program makes it to the ISR. In other words, you need to verify you're receiving an interrupt and the "unit" is acting properly when you're receiving that interrupt. The best approach to do this is to first verify you can turn on an LED outside of the ISR (in your main code). The next task is then to turn on an LED inside of the ISR and immediately stop your program. You of course can't stop your program but you can make it seem as if it is stopped by inserting the following line into your assembly code: "**`killme:  BRN    killme`**". And if you have any better tricks, I'd love to hear about them.

Programming Assignment:

Write an interrupt driven RAT MCU assembly language program that does the following. The program outputs the most recent value on the switches to the LEDs when it receives an interrupt. If the program outputs the same switch value on two consecutive interrupts, it stops outputting values until BUTTON(0) is pressed, at which point it returns to normal processing.

- Don't use an IN instruction in the interrupt service routine
- The same switch values will never appear more than two consecutive interrupts
- Assume an external device generates the interrupt
- Minimize the number of instructions in your solution

```
.EQU    LEDS     = 0x44
.EQU    SWITCHES = 0x45
.EQU    BUTTONS  = 0x46
```

Hardware Design Assignment:

You must modify the RAT MCU architecture by changing the size of the register file from 32x8 to 8x8 and the prog_rom from 1k instructions to 4k instructions. For this problem, describe the following:

a) changes you need to make to the RAT hardware

b) changes you need to make to the RAT assembler

c) changes in RAT MCU memory requirements

d) why this modification would be useful

Device	Size	# bits
prog_rom	1024 x 18	18432
register file	32 x 8	256
program counter	10	10
stack pointer	8	8
flags(I,Z,C,shZ,shC)	5 x 1	5
scratch RAM	256 x 10	2560
	TOTAL	**21,271**

Questions:

1. Briefly but completely explain why, in general, why is it important to keep your ISRs as short as possible.
2. We often consider the ISR code as having a "higher priority" than the non-interrupt code. Briefly but fully explain this concept.
3. If you were not able to use a LUT for this experiment, briefly but completely describe the program you would need to write in order to translate a given BCD number into its 7-segment display equivalent.
4. What is the minimum execution time of an ISR requires? This is the time from when the MCU switches to the interrupt state to when the processor begins executing the next intended instruction, which was scheduled to be executed before the interrupt processing started. Provide a concise answer with adequate description. Assume the interrupts return from the ISR in the disabled state. Recall that that the shortest possible ISR is what you should consider for this problem, which would simply be a RETID instruction.

5. As you know, part of the interrupt architecture includes the RAT MCU automatically masking the interrupts. This being the case, why then is there a need to keep the overall output pulse length of the interrupt relatively short?
6. Nested subroutines are quite common in assembly language programming. Accordingly, there is also the notion of nested interrupts? Briefly explain how you could allow nested interrupts to occur under program control on the RAT MCU. For this problem, don't worry about the issues with the C and Z flags.
7. In theory, how deep could you nest interrupts on the RAT MCU? Briefly but concisely explain.
8. Write some RAT assembly code that would effectively implement a firmware debounce for a given button. For this problem, assume the button you're debouncing is associated with the LSB of buttons addressed by port_ID 0x55. Make sure to fully describe your code with instruction comments and header comments. Also include a flowchart that models your code.
9. You wrote assembly code for this experiment. Briefly but completely explain whether this code was software, firmware, or wankware.
10. State whether the RAT MCU is a RISC or CISC processor. Support your statement with at least three characteristics regarding those two types of computer architecture.
11. Describe the difference between the level and pulse outputs of the debounced one-shot module.

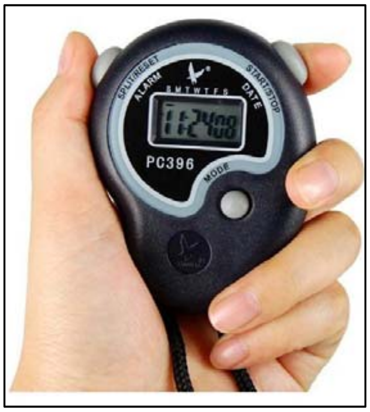
<u>Deliverables:</u>

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.
2. A complete solution to the Programming Assignment. Completeness counts.
3. A complete solution to the Hardware Design Assignment. Completeness counts.
4. Well-structured and well-commented RAT assembly code for your RAT program
5. A flowchart(s) for your RAT assembly code
6. A signature from the instructor indicating your hardware and firmware are working.

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #11:

Interfacing the RAT MCU to an External Peripheral



Learning Objectives:

- To use the RAT MCU to interface with an external a timer-counter peripheral device
- To write an interrupt driven RAT assembly program

General Design Notes: External Device Interfacing: Microcontrollers are useful because of their ability to interface with other digital devices in a digital system. We typically refer to such digital devices as “computer peripherals”, or “peripherals”. Having peripherals out there fulfills the microcontroller promise in that that “micro” refers to computer and “controller” describes the main purpose of that computer device. Let’s face it, embedded systems are what currently run the world; at the heart of an embedded system is an embedded computer. While there are many devices out there such as “computers” (desktops, laptops, pads, etc.) there are actually orders of magnitude more embedded systems out there.

This experiment is uses a relatively simple version of one of the most popular peripherals out there: the timer-counter. This peripheral is so common and so powerful that many high-end MCUs have more than ten different independent timer-counters on one MCU device. Timer-counters typically come in the form of “internal peripherals”, meaning that they are included on the same piece of silicon as the MCU itself. There are also many external peripherals dedicated to timing-type tasks, which you can include in your designs in case your MCU does not provide all the internal timers you need. The timer-counters on hardcore MCUs are much more feature-laden then the timer-counter in this experiment.

High-Level Timer-Counter Description: Figure 24 shows the black-box diagram for the timer-counter module. The timer-counter is an up-counter that automatically counts up when the device is enabled. The user can preset a “terminal count” for the timer-counter; when the timer-counter reaches that terminal count, the timer-counter outputs a positive pulse that is two clock periods wide. You can use this pulse output as an input to the RAT’s interrupt input. In this way, the timer-counter can automatically generate interrupts in at a frequency specified by the programmer by writing to the timer-counter’s registers that hold the terminal count. The timer-counter also has a clock pre-scaling feature to allow programmers to further reduce the frequency of generated interrupts.

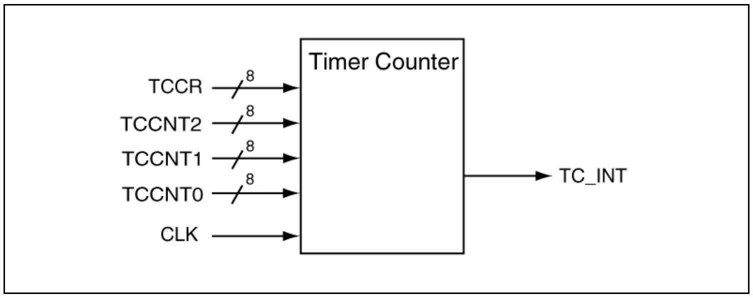
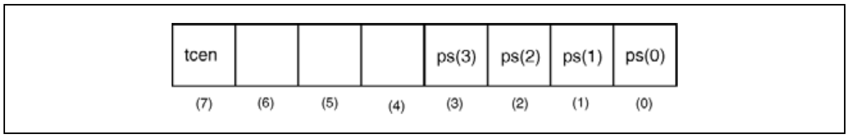


Figure 24: Black box diagram for the timer-counter peripheral.

Low-level Timer-Counter Description

The timer-counter module consists to two main registers: the timer-counter control register (TCCR) and the terminal count registers (TCCNTx); the TCCR is an 8-bit register while the terminal count register is a 24-bit register (implemented as three 8-bit registers). Figure 25 shows the format of the TCCR while Table 8 provides a detailed description of the bit positions in TCCR.



tcen				ps(3)	ps(2)	ps(1)	ps(0)
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

Figure 25: Timer-counter control register TCCR.

Signal	Description
tcen	This signal enables or disables the timer-counter. Writing to this bit turns on ('1') or turns off ('0') the timer-counter module. When the timer-counter is turned off under program control, the count associated with the timer-counter resets to zero.
ps(3:0)	This four-bit signal represents a number that the timer-counter uses to "prescale" the clock input to the timer-counter. In this way, the programmer can effectively reduce the input clock frequency before the signal is "counted" by the timer-counter. Scaling the inputs clock signal allows for longer count periods beyond the value held in the timer-counter terminal counter registers. The programmer can write these pre-scale bits at any time in a count sequence.

Table 8: Timer-Counter Control Register (TCCR) description.

Assignment "A":

Write an interrupt driven program that uses the provided timer-counter module to blink an LED at various rates. Specifically, implement the following functionality:

- If only the left-most switch on the development board is on, the right-most LED blinks at a 0.5Hz rate
- If only the right-most switch is on the development board is on, the right-most LED blinks at a 10Hz rate
- For any other switch setting, all the LEDs are off

Constraints:

You must use the timer-counter module to generate interrupts to create the required blink rates. For this experiment, you don't need to debounce the switches, so make sure you remove your debounce module from the previous experiment.

Assignment "B":

Write an interrupt driven program that uses the provided timer-counter module to blink an LED with various attributes. Specifically, implement the following functionality:

- If only the left-most switch on the development board is on, the right-most LED blinks at a 0.5Hz rate and a 50% duty cycle.
- If only the right-most switch is on the development board is on, the right-most LED blinks at a 1Hz rate with a 10% duty cycle.
- For any other switch setting, all the LEDs are off

Constraints:

You must use the timer-counter module to generate interrupts to create the required blink rates. For this experiment, you don't need to debounce the switches, so make sure you remove your debounce module from the previous experiment.

Assignment "C":

Re-implement Experiment 10 using a timer-counter. Specifically, implement the following functionality:

- Experiment 10 used external hardware to handle the switch debouncing requirements. For this experiment, debounce the switch in firmware. Provide at least 10 μs of debounce "protection".
- Use the timer-counter module to handle the delays associated with the display multiplexing.
- Make sure your firmware checks for a button release before looking for another button press. Make sure you also debounce the button release for at least 10μs.

- To be clear, Experiment #10 used implemented the multiplexing delay in firmware and the debounce in hardware. For this experiment, you'll implement the multiplexing delay using the timer-counter and the debounce using firmware.

Constraints:

There are many approaches to architecting the firmware for this problem. A good approach to this problem would be to first handle the display multiplexing with the timer-counter. Once that is working properly, implement the debounce part of the circuit.

HINTS:

- You only need to do the assignment that your instructor directs you to; don't do them all.
- While we always consider polling to be a bad thing, you must poll the switches for all parts of this experiment. This is because the timer-counter module is using the RAT MCU's interrupt input, which would be the most desirable option.
- Be sure to examine and understand the example timer-counter program associated with the timer-counter module. Understanding this program makes this experiment straight-forward.
- Be sure not to use the debounce/one-shot model from Experiment #10 in this experiment.

Programming Assignment:

Write an interrupt driven RAT MCU assembly language program that does the following. The program outputs the largest of the three most recent values that were on the switches when the RAT MCU received an interrupt to the LEDs. Assume there are eight LEDs and eight switches. Interpret the eight-bit switch values are an unsigned binary number.

- Don't use an IN instruction in the interrupt service routine
- Assume an external device generates the interrupt
- Minimize the number of instructions in your solution

Hardware Design Assignment:

You want to add the following instruction to the RAT MCU: "`IN     (r3),PORT_ID`", where r3 is the destination operand that holds the address in scratch RAM of where you want to place the data and "PORT_ID" is the same as in the current IN instruction. For this problem, do the following:

a) changes you need to make to the RAT hardware

b) changes you need to make to the RAT assembler

c) changes in RAT MCU memory requirements

d) why this modification would be useful

Device	Size	# bits
prog_rom	1024 x 18	18432
register file	32 x 8	256
program counter	10	10
stack pointer	8	8
flags(I,Z,C,shZ,shC)	5 x 1	5
scratch RAM	256 x 10	2560
	TOTAL	**21,271**

Questions

1. Briefly describe why it is “more efficient” to use a timer-counter peripheral to blink an LED and using a dumb loop (delay loop) to blink an LED.
2. The timer-counter module uses a “clock pre-scaler”. Briefly describe how you would model the timer-counter without the pre-scaler yet still be able to attain approximately the same blink rates as with the prescaler.
3. Examine the VHDL model for the timer-counter and briefly but completely describe how it operates. Be sure to mention both the counter portion as well as the pre-scaler.
4. If you configured the timer-counter to generate an interrupt on the RAT MCU every 10ms, what is the highest frequency blink rate of an LED using that interrupt? Briefly explain your answer.
5. Briefly but completely explain how using the “clock prescaler” will prevent the firmware programmer from blinking the LED at all possible frequencies lower than the system clock frequency. For this problem, assume the timer-counter module uses the system clock.
6. Changing frequencies of the timer-counter can possibly create a timing error for one clock period. Briefly describe what causes this one hiccup.
7. The timer-counter provided for this experiment provided a means to easily blink an LED with a 50% duty cycle. Using the RAT MCU and the timer-counter, there is a *programming* “overhead” associated with blinking the LED at an exact frequency if the duty cycle is not 50%. Briefly describe how and when this overhead can interfere with the frequency output of the blinking LED.
8. How would you change the timer-counter module such that you could use it to “time” the length of time a given signal is asserted. Briefly but completely explain. HINT: The answer for this problem is ***not*** along the lines of keeping track of the number of generated interrupts
9. If you tied the output of the timer-counter to a one-shot and the output of the one-shot to the ISR input of the RAT MCU, would you be able to generate an interrupt? Briefly explain your answer.
10. ***Only answer this question if you did not do part C of this experiment.*** The previous experiment involved the firmware multiplexing of displays. While it is desirable to have external hardware multiplex the display, a compromise would be to use a timer-counter in conjunctions with the firmware algorithm. For this problem, briefly but completely describe an approach that implements Experiment 10, but uses a timer-counter instead of firmware-based delay loops.

Deliverables:

1. Complete, well-reasoned answers to the questions in Questions section. Completeness counts. Feel free to include diagrams in your answer.
2. A complete solution to the Programming Assignment. Completeness counts.
3. A complete solution to the Hardware Design Assignment. Completeness counts.
4. Well-structured and well-commented RAT assembly code for your RAT MCU program
5. Calculations showing how you generated the timer-counter settings (both the prescaler and count) for the assignment that you did.
6. A signature from the instructor indicating your hardware and firmware are working.

Make sure your deliverables are submitted in the correct sequence, which is described at the beginning of this lab manual.

Experiment #12:

The Final Project

Learning Objectives:

- To gain further experience programming the RAT MCU
- To gain experience interfacing with various peripheral devices

General Overview: Though you technically know everything there is to know about the RAT MCU, you're still a bit thin on actual RAT programming experience. Specifically, while Experiment 10 used many aspects of the RAT MCU including indirect memory access and interrupts, the programming required by the solution was actually quite short. Additionally, you've not had to interface with a significant number of peripheral devices in the experiments thus far.

The underlying goal of this experiment is to provide you with a path that will enable you to tie together any loose ends in this course by allowing you to choose implement a your own project rather than have someone tell you what to do. This open-ended approach to the final experiment has led to many cool and interesting projects by past students taking this course.

The RAT MCU provides you with a relatively generic engine to drive other system "peripherals". The technical field uses the term peripheral to describe devices that devices such as MCUs commonly interface with. This somewhat good news about this experiment is that there are several peripheral devices out there that you can use with your final project. These devices exist in one form or another and you can find the list on the course website. The listed devices are a growing list; these devices have been out there for a while but they lacked sufficient documentation or operated in a way that made them hard to use for students with only a few weeks remaining in the quarter. Most of the peripherals are now tested and documented to the point that you can use quickly use these devices in your projects.

Final Project Issues: The final project represents a relatively small portion of your final grade in this course. Embarking upon and completing a final project is not going to make or break your final course grade. Thus, the main goal of the final project is to allow you to develop and/or improve your programming skills so that you can do well on both the lecture and lab final exams. In other words, the final project is a tool for you to use to develop programming and interface skills that will help you in other areas of this course and beyond. You should keep these notions in mind when selecting a final project. The main issues is that you don't have to selection a project that saves the world or a project that is better than every other previously done project in this or a similar course. It's completely OK to re-implement a project that has already been done and it is OK to complete a relatively simple project. This is your learning experience; choose wisely.

Available Peripherals and Demonstration Programs:

The VGA Driver: The VGA driver is a peripheral that allows you to interface with a VGA display using a reduced resolution. The current peripheral comprises of the actual driver, a RAT Wrapper VHDL file, an assembly language driver program, and some additional documentation. The assembly language driver program is based on the famous "draw-dot" function, which the program subsequently uses to draw the display background, draw horizontal and vertical lines, and draw a dot on the display.

The Timer-Counter Module: The Timer-Counter Module allows you to generate interrupts at frequencies you can program into the device. In this way, you don't need to time things in your program using dumb loops. The Timer-Counter Module comprises of the associated VHDL models, an assembly language driver program, and an associated user's manual.

The Advanced 7-Segment Display Module: This module provides various seven-segment display abilities. This external VHDL module operates in several modes which allows you to have several types of decimal counting displays including one [0,9999] counter (unsigned), one [0,255] counter (signed), or two [0,99] counters (unsigned). You can also apply one decimal point on the first counter. The module includes the associated VHDL models and a user's manual.

The Pseudo-Random Number Generator Module: This module allows you to use an 8-bit pseudo random number in your projects. This module includes the VDHL modules and an associated user's manual.

The Keyboard Driver Module: This module provides a basic interface between an external keyboard and the Nexys2 board. The external keyboard is a PS/2 interface and uses the mini-DIN connector on the Nexys2 board. This module includes the VHDL models for the driver, a RAT assembly language driver program, and supporting documentation including a user's manual. The RAT assembly language driver provides the hooks to using the keyboard; you generally need to expand on this driver if you do in fact use it in your program.

The Mouse Driver Module: This module was always problematic. Jeff Gerfen gave me a new version of this module late Spring 2014 quarter and swore to me that it worked great. As of this writing, I have yet to look at it; I provided to you untested.

The Etch-A-Sketch Demonstration Program: This program expands on the keyboard driver module by providing pseudo Etch-A-Sketch functionality. The program uses the external keyboard to guide the "cursor" on the display. The fun thing here is that the cursor does not erase the areas it has been to and does not allow the user to drive the cursor off the display. The overall package includes a RAT Wrapper program, constraints file, an assembly language driver program, and supporting documentation.

The DrawBlit Demonstration Program: This program is similar to the Etch-A-Sketch program in that is uses an external keyboard to "make things happen" on the display. This program draws a "blit" on the display and allows the user to move it using the keyboard. In this case, a blit is an object that you can draw on the display. Once again, this program provides the hooks from which you can modify to do something non-boring in your own program. One main idea presented by this module is that you only change the parts of the display that require changing, and not to the entire display; it is desirable to avoid redrawing the entire display if only a small portion of the display changes. The overall package includes a RAT Wrapper program, constraints file, an assembly language driver program, and supporting documentation.

Assignment:

1) Choose a project and submit a one-page outline of your project to your instructor for approval.
2) Implement your approved project using your RAT MCU.
3) Document your project with both code and a general description.
4) Demonstrate your final project to the course instructor.

Deliverables: This experiment has three main deliverables; I'll determine your grade for this experiment based on the overall quality of these deliverables.

1. A white-paper describing your final project: This paper is an executive summary of your project, which means is should as short as possible while still being complete. The project should be no more than two pages in length. Issue you should include in this paper are a description of your projects including block diagrams, a description of any pressing issues you faced, and an operating manual where applicable. Make this paper professional looking a reading; plan on writing in active voice.

2. The RAT assembly code associated with your experiment and any significant VHDL code you wrote also. Keep in mind that this code represents your terminal experience for this course so I'll be grading it for proper form and appearance. This means items such as properly commented, file banners, subroutine banners, assembler directives, proper indentation, etc. Be sure to check out the RAT MCU style file before you submit anything. Also, feel free to allow me to look over your code before you submit it.

3. Demonstrate your final project to the course instructor. For this part, make sure you find me during the class meeting time of during office hours to allow me to "check off" your project. Submitting the other two deliverables is not sufficient to skip an actual demonstration.

Appendix

Requiem for the Digital Logic Designer

Digital design is the process where you create a digital circuit to solve a given problem. There are many approaches you can use to solve given problems, designing a digital logic circuit is simply one of them. What makes digital design so useful is that the design can generally interface with other digital circuits such as computer-type circuits. The two basic tenets of digital logic are:

- **Digital logic circuits are hierarchical**: We can describe a digital circuit at various levels; the level at which we describe digital logic is generally the one that allows us to transfer as much useful information as possible. Abstracting digital designs to higher levels aids in understanding and designing circuits.

- **Digital logic circuits are decomposable into a few basic digital circuits**: Although there are many ways to describe digital circuits, we strive to make the descriptions an aggregate compilation of standard digital circuits in able to help us understand the circuits.

A given digital design solves problems by having the outputs react to the inputs in a manner such that it solves the given problem. There are two basic types of digital logic circuits:

- **Combinatorial Circuits**: circuit outputs are a function of the circuit's inputs.

- **Sequential Circuits:** circuit outputs are a function of the sequence of the circuit's inputs.

The main ramification of sequential circuits is that they can "remember" the previous "state" of the circuit. Sequential circuits can store (remember) bits; we refer to the bits the circuit is remembering as the "state" of the circuit. Combinatorial circuits, by definition, do not have state.

Figure 1 shows a digital logic circuit containing both sequential and combinatorial modules. We can thus model digital circuits as a controlled interaction between a set of sequential and combinatorial circuits. Solving problems using digital circuits requires controlling the flow of data through the circuit in such a way that it provides a solution to the given problem.

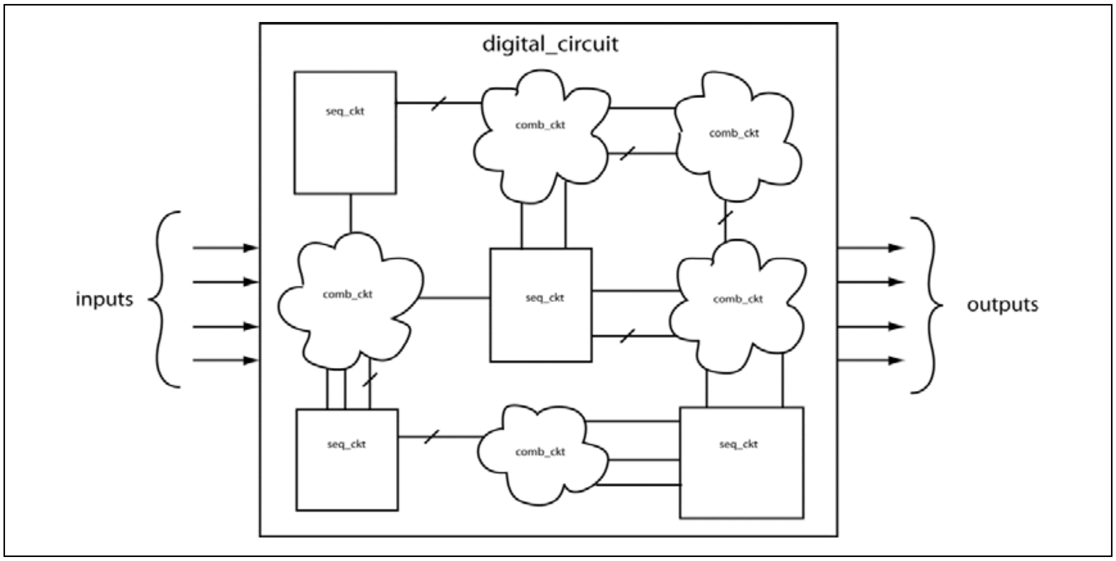
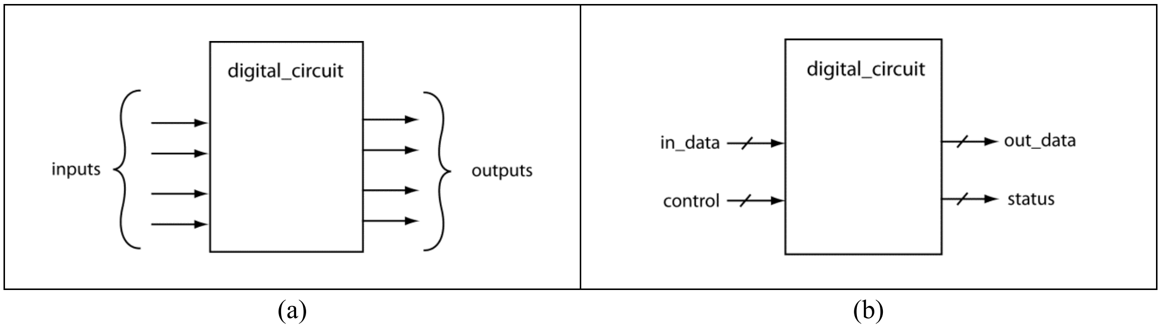


Figure 1: A basic logic circuit.

Figure 2 (a) shows the basic model of a digital logic circuit; we characterize the signals that the outside world sees as either inputs or outputs. Because we need to control the flow of data through the digital circuit, we must more specifically define the inputs and outputs of a basic digital circuit module. Figure 2(b) shows that we further classify the inputs as either “data” or “control” and classify the outputs as either “data” or “status”. This means the various circuit elements in Figure 2(b) are able to 1) pass data from their inputs to their outputs under the direction of the “control” inputs and, 2) output characteristics of the data transfers using the status outputs.



(a) (b)

Figure 2: Models for a basic logic circuit (a), and a more refined basic digital logic circuit (b).

Something must control the flow of data through the generic digital circuit. We therefore must have some other entity that interprets the status signal outputs of the circuit modules and issues control signals to those circuit modules. For the purpose of this discussion, we’ll consider this circuit to be a finite state machine (FSM). The important thing to remember is that something controls the circuit, whether it is an FSM, a computer, or a herd of confused administrators.

Figure 3 shows a generic model of an FSM. The FSM simply interprets the status signal outputs from various digital modules and then outputs the appropriate control signals that are the various digital modules use as control inputs. Other interesting characteristics to note include:

- FSMs generally do not have data inputs and data outputs. You can design FSMs with data inputs and outputs, but they tend to be klunky and non-generic. Non-generic FSMs will require modifications if the data widths within the controlled circuit change.

- The FSM is a sequential circuit because it has the ability to store bits. The FSM only stores bits to represent the “state” of the FSM, which it does in its “state variables”.

- The underlying model of the FSM includes three primary elements: 1) the next state decoder, 2) the output decoder, and, 3) the state variables. The next state decoder is a combinatorial circuit that decides the next state based on the given state and status inputs. The output decoder is a combinatorial circuit that generates control outputs based on either state only (Moore machine) or state and status inputs (Mealy machine). Figure 4shows models for the Moore and Mealy-type FSMs.

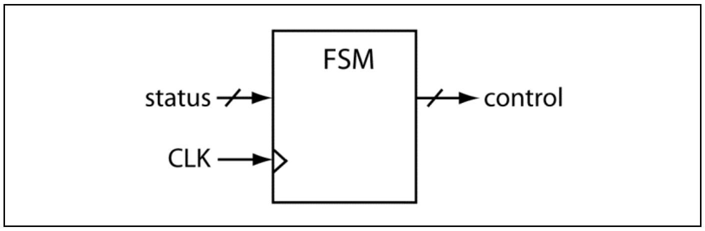


Figure 3: A black box model of a FSM.

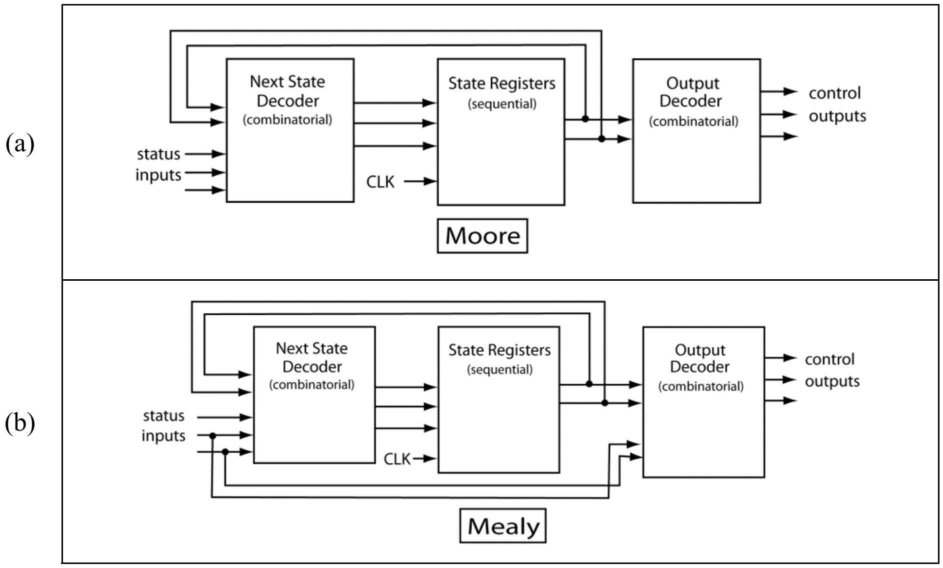

(a)
Next State Decoder (combinatorial)
State Registers (sequential)
Output Decoder (combinatorial)
control outputs
status inputs
CLK
Moore
(b)
Next State Decoder (combinatorial)
State Registers (sequential)
Output Decoder (combinatorial)
control outputs
status inputs
CLK
Mealy

Figure 4: The two types of FSMs: (a) Moore, and, (b) Mealy.

Figure 5 shows a modified version of Figure 2 that includes an FSM as a control element. Figure 6 shows that we can further this abstraction. Figure 6 shows that the circuit control elements can either be hardware (FSMs) or software (microcontrollers). Additionally, Figure 6 shows that the modules that we can control in a digital circuit include "computer peripherals" as well as the low-level digital modules in Figure 2. Figure 6 represents computer peripherals using circles.

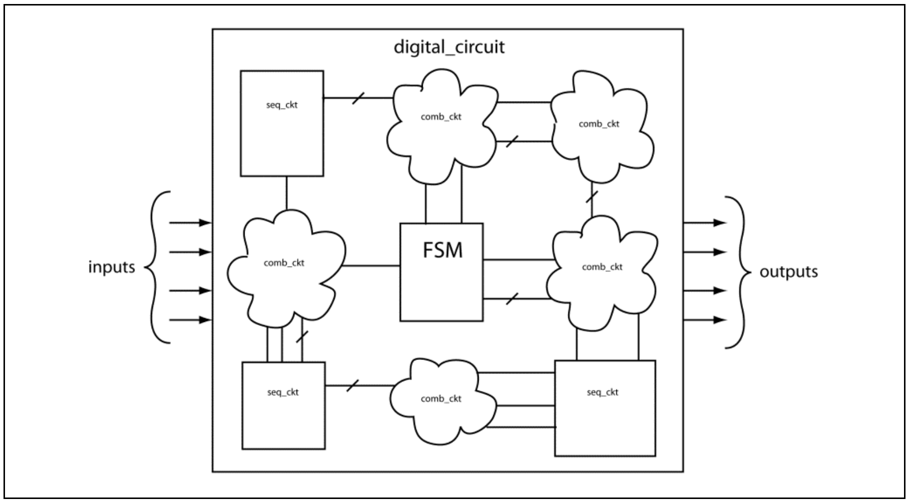


Figure 5: A basic logic circuit controlled by FSM

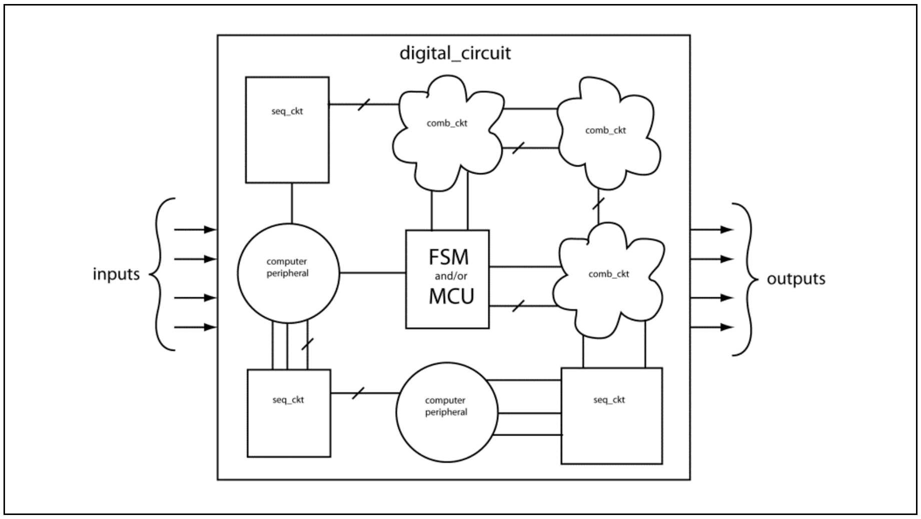


Figure 6: A basic logic circuit with peripherals and various control circuits.

Ripple Carry Adder (RCA)

The RCA is a combinatorial module that performs addition; it comprises of a series of Full Adders (FAs) connected in series such that the Co from one module connects to the Cin of the next higher bit location. The RCA can also perform subtraction by changing the sign of one addend before performing the addition.

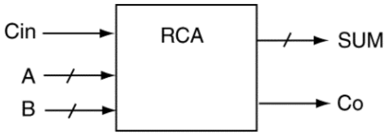
Diagram	Input/Output
	Data In: **A**, **B**, **Cin**. A & B are the addends; Cin is the carry in. Control In: none Data Out: **SUM**. Summation of: A+B+Cin. Status Out: **Co**. The Carry out; indicates if the RCA's addition generated a carry out of the module's MSB.

Figure 7: The RCA module overview.

Multiplexor (MUX)

The MUX is a combinatorial circuit that selects which of many (more than one) data inputs appear on the circuit's single data output. The SEL signal determines which signals transfers to the output, which requires that it have a width of at least: $\lceil log_2(\text{number of data inputs})\rceil$. The width of the data inputs and outputs are equivalent. The most generic forms of MUXes include 2:1, 4:1, 8:1, 16:1, etc.

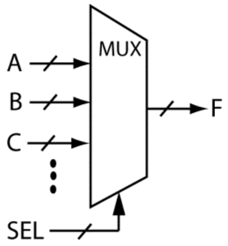
Diagram	Input/Output
	Data In: **A**, **B**, **C, etc**. MUXes can have any number of data inputs. Control In: **SEL**. selects which data input appears on the F**.** The width of the SEL signal is such that 2^{SEL} ≥ to the number of data inputs. Data Out: **F**. A single output which is equivalent to one of the inputs as selected by the SEL signal. Status Out: none

Figure 8: The MUX module overview.

Comparator

The comparator is a combinatorial circuit that generates an equality-type relationship between the two inputs. The comparator has outputs of EQ (equal), LT (less than), and GT (greater than) which are characteristics of the relationship between the two input. The comparator's two input values are typically bundled values of equal width.

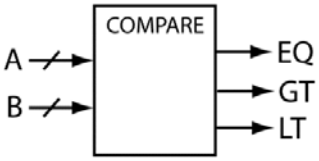
Diagram	Input/Output
	Data In: **A**, **B**. the two bundled values to be compared. Control In: none Data Out: none Status Out. **EQ** (A=B), **LT** (A<B), **GT** (A>B).

Figure 9: The Comparator module overview.

Generic Decoder

The generic decoder is a combinatorial circuit that establishes a functional relationship between the module's data inputs and data outputs. The generic decoder is a digital circuit implementation of a look-up-table (LUT). The generic decoder's inputs and outputs are hard to classify because inputs can include data and/or control and outputs can contain at and/or status. We thus describe this circuit using the IN_DATA input for all the inputs (whether they be data or control) and the OUT_DATA for all outputs (whether they be data or status). Both IN_DATA and OUT_DATA can be bundles or single bits, which allows us to include basic logic gates as generic decoders.

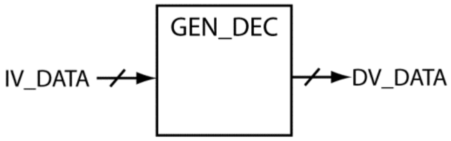
Diagram	Input/Output
	Data In: **IV_DATA**; the function's independent variables Control In: none Data Out: **DV_DATA**; the function's dependent variables Status Out: none

Figure 10: The Generic Decoder module overview.

Standard Decoder

The standard decoder is a combinatorial and is a subset of generic decoders. The standard decoder has a special relationship between the SEL inputs and the outputs. The number of single-bit outputs = $2^{(width\ of\ SEL)}$. The output bits have either a one-hot (only one output bit is set) or one-cold (only one output bit is cleared) form. We often describe standard decoders using the notation: 1:2, 2:4, 3:8, 4:16, etc.

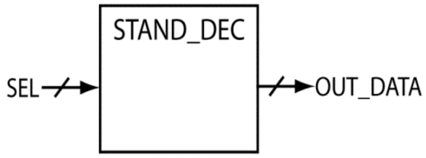
Diagram	Input/Output
	Data In: none. Control In: **SEL**; selects the form of the output. Data Out: **F**; output in one-hot or one-cold form. Status Out: none

Figure 11: The Standard Decoder module overview.

Parity Generator

The parity generator is a combinatorial circuit that establishes a given parity for the aggregate combination of the DATA inputs and PAR output. In other words, the parity generator assigns the parity bit such that the DATA & PAR bits are either odd or even parity. Parity "checkers" circuits are similar to parity generators, where the PAR output indicates the DATA bits are either odd or even parity.

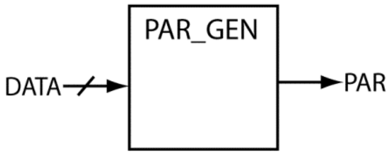
Diagram	Input/Output
	Data In: **IN_DATA.** data used for establishing Control In: none Data Out: none Status Out: **PAR**. the bit that creates the appropriate parity for the **DATA** & **PAR** aggregate value.

Figure 12: The Parity Generator module overview.

Registers

The register is a sequential circuit that to stores bits of data. Registers generally store multiple bits of data; we refer to a 1-bit register as a flip-flop. The register's load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. Any data loaded to the register appears on the register's OUT_DATA output after a given propagation delay. Registers also have inputs such as CLR, which clears each of the bit storage elements in the register. Signals such as CLR are often asynchronous, which means the given action occurs immediately upon asserting the CLR signal.

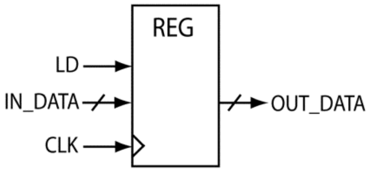
Diagram	Input/Output
	Data In: **IN_DATA**; data to be synchronously loaded into the register. Control In: **CLK, LD, CLR**; The CLK signal synchronizes the loading of data into the register, which happens when both an active clock edge occurs when the LD input is asserted. The CLR input clears each bit storage element in the register either synchronously or asynchronously. Data Out: **OUT_DATA**; the data previously loaded to the register. Status Out: none

Figure 13: The Register module overview.

Counters

The counter is a sequential circuit that is a special type of register, which means it retains all the control inputs associated with a register. The register’s load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. The counter as the ability to count up (adds ‘1’ to stored value) or count down (subtracts ‘1 from stored value). As with the LD signal, changes to the stored register value are synchronized with the active clock edge. Counter typically have inputs such as CLR, which serves to clear each of the bit storage elements in the register. Counter also have inputs such as CLR, which clears each of the bit storage elements in the register. Counters generally automatically “roll over” when they reach their terminal values, which means that the count transitions from its maximum value to zero when counting up and from zero to its maximum value when counting down.

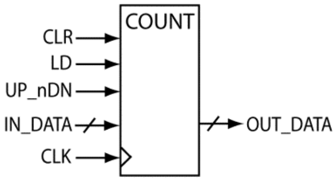
Diagram	Input/Output
	Data In: **IN_DATA**; parallel data for synchronous loading. Control In: **CLK, LD, CLR, UP_NDN, CEN**; The CLK signal synchronizes the loading of data into the register and the changing of the count, which can either be up or down as controlled by the UP_NDN signal. The LD signal controls the loading of new data into the register while the CEN signal enables the synchronous counting of the circuit. The CLR input clears each bit storage element in the register either synchronously or asynchronously. Data Out: **OUT_DATA**; the data previously stored in the circuit and possibly modified as loaded to the register. The OUT_DATA signal is the “count” value of the counter. Status Out: **RCO**; establishes when the counter outputs are at the counter’s terminal value. The terminal value depends on the count direction (UP_NDN).

Figure 14: The Counter module overview.

Shift Registers

The shift register is a sequential circuit that is a special type of register. The register’s load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. The SHR & SHL signals cause synchronous right and left-shifting of the data in the register, respectively; the in_data bit becomes the new MSB for right shift or the new LSB for left shifts. Shift registers can contain CLR and HOLD inputs which, clears each of the bit storage elements or does not change them, respectively. Shift registers most often bundle the control inputs into a single select-type signal.

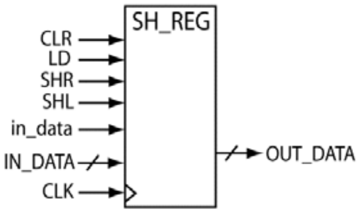
Diagram	Input/Output
 SH_REG CLR LD SHR SHL in_data IN_DATA CLK OUT_DATA	Data In: **IN_DATA, in_data**; IN_DATA is the multibit signal to be loaded into the SR; in_data is the single bit of that becomes the MSB or LSB of the stored value for right or left shifts, respectively. Control In: **CLK, LD, CLR, SHR, SHL, HOLD**; The CLK signal synchronizes the loading of data into the register, which happens when both an active clock edge occurs when the LD input is asserted. The SHR and SHL (shift right & shift left) signals cause the store bits to synchronously shift either right (in_data becomes the new MSB) or left (in_data becomes the new LSB). The HOLD signal prevents changes in the registers content. The CLR input clears each bit storage element in the register either synchronously or asynchronously. Data Out: **OUT_DATA**; the data stored in the shift register. Status Out: none

Figure 15: The Counter module overview.

Random Access Memory (RAM)

The RAM is a sequential circuit that allows for the storage of large amounts of data relative to registers. RAM contains three types of input/output signals: address, data (input and output), and control. The IN_DATA signal is the data that will write to the RAM; the OUT_DATA is the data that is read from RAM. All data reads and write occur at the RAM location specified by the address inputs. The value of the control signals allow the data reads or writes. While memory modules often have many control signals, we'll only consider a CLK and a WE (write enable) signal in order to simplify this description. Reading data from the RAM is an asynchronously operation; writing to the RAM is a synchronous operation. Read Only Memories (ROMs) have most of the same features of a RAM except for the WE signal. Additionally, reading from ROMs can either be synchronous or asynchronous.

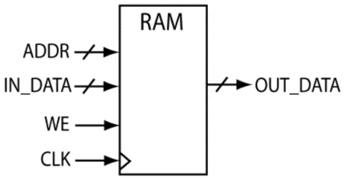
Diagram	Input/Output
	Data In: **IN_DATA**; data to be synchronously written to RAM. Control In: **CLK, WE, ADDR**; The CLK signal synchronizes the writing of data to the RAM; the RAM stores the value of IN_DATA at the address associated with the value of ADDR on the active clock edge (synchronously) when the WE signal is asserted. Data read from RAM is asynchronous; data appearing on OUT_DATA is the data associated with the ADDR value when WE is not asserted. Data Out: **OUT_DATA**; the data currently stored in the shift register. Status Out: none

Figure 16: The Counter module overview.

Finite State Machine (FSM)

The FSM is a sequential circuit that controls other digital circuits. The FSM reacts to status inputs and issues appropriate control outputs. The FSM's control inputs are "status" outputs form other digital modules, while the FSM's status outputs become "control" inputs to other digital modules. The FSM uses the value of the status inputs to transition through the various states of the FSM on the active edge of the CLK signal. The control outputs can either Moore (outputs a function of state only) or Mealy-type (outputs are a function of both state and status inputs).

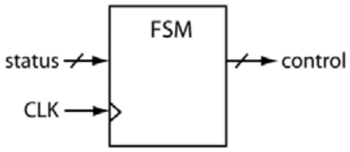
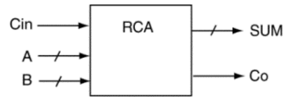
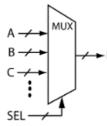
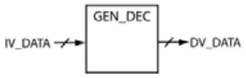
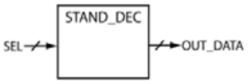
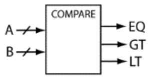
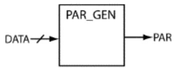
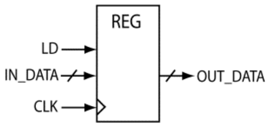
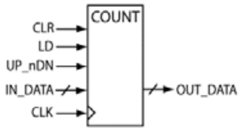
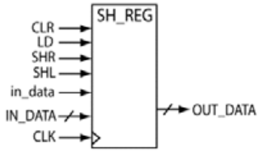
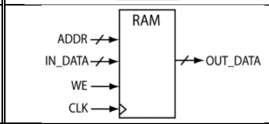
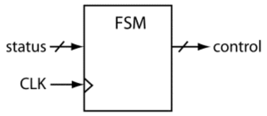
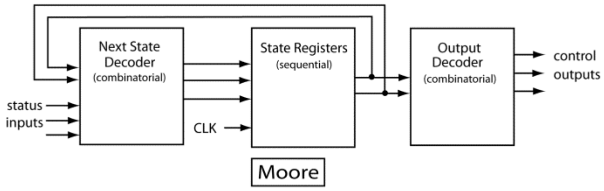
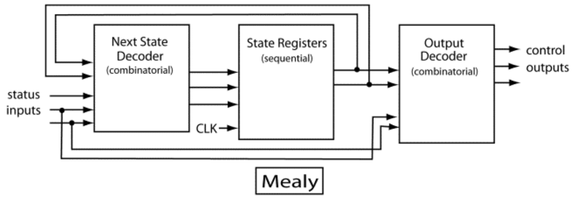
Diagram	Input/Output
	Data In: none Inputs: **CLK, status**; The CLK signal synchronizes state transitions of the FSM; status inputs are the status outputs of modules external to the FSM. Status input values determine the FSM's state transitions. Outputs: **control**; circuit elements external to the FSM use the control outputs to facilitate data handling; control outputs can be either Mealy or Moore-type. Status Out: none

Figure 17: The Finite State Machine.

		Circuit Diagram	Data IN	Control IN	Data OUT	Status OUT
CombInatorial	**RCA**		A B Cin	-	SUM	Co
	MUX		Multiple DATA	SEL	Single DATA	-
	Generic Decoder (LUT)		IN_DATA	-	OUT_DATA	-
	Standard Decoder		IN_DATA	SEL	OUT_DATA	-
	Comparator		A B	-	-	EQ GT LT
	Parity Generator		DATA	-	-	PARITY
SequentIal	**Register**		IN_DATA	CLK LD CLR	OUT_DATA	-
	Counter		IN_DATA	CLK LD UP/DOWN ENABLE	COUNT	RCO
	Shift Register	 SH_REG CLR LD SHR SHL in_data IN_DATA CLK OUT_DATA	IN_DATA	CLK LD SH LEFT/RIGHT data ENABLE	OUT_DATA	-
	RAM		IN_DATA -	CLK WE ADDR	OUT_DATA -	-
				Inputs		**Outputs**
	FSM		-	CLK status	-	control
FSM Models	 Next State Decoder (combinatorial) State Registers (sequential) Output Decoder (combinatorial) control outputs status inputs CLK Moore			 Next State Decoder (combinatorial) State Registers (sequential) Output Decoder (combinatorial) control outputs status inputs CLK Mealy		

DATA = multiple bits data = single bit

COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

CPE 233 External RAT Module

Universal Seven-Segment Display Module Description

Objectives:

- To describe the functionality and use of the Universal Seven-Segment Display module

Somewhat Meaningful Comments:

Computers and other forlorn digital circuits would be rather boring without some type of display device, particularly a display device that shows numbers. Recall that CPE 133 used the infamous "sseg_disp" module in about every experiment in an attempt to make the Nexys2 development board less boring. While that display had its moments, it was rather limited in what numbers it could actually display.

This write-up describes the universal seven-segment display module (USSDM). This module is usable in any design but this document describes its use in conjunction with the RAT CPU. The big notion here is that the USSDM is an external module, which means that the USSDM does most of the work instead of the RAT CPU. Put in another way, the RAT CPU could actually be implementing all the functionality of the USSDM, but if it did so, it would require a significant amount of CPU processing time and program code space, which could both be better spent doing other things. The notion of offloading processing requirements from the CPU to other devices, external or internal, is extremely common out there in computer land. This module is can be useful in conjunction with the RAT CPU if you have a need to display certain types of numbers without but don't want to do all the display multiplexing and other fun stuff via RAT firmware.

High-level USSDM Description

Figure 24 shows the black-box diagram for the USSDM. I'd tell you all about the low-level description, but the details are dangerously boring, so let's stick to the high-level details. The high-level details depend on the notion that the development board contains four seven-segment displays. The USSDM allows you to display either one or two numbers, where the one-number display range can either be [0,255] or [0,9999], while the two number display is two two-digit numbers [0,99]. Additionally, you can also toss in a decimal point on any one of the four seven-segment display. The USSDM has some other potentially useful features as well; Table 9shows the gory details.

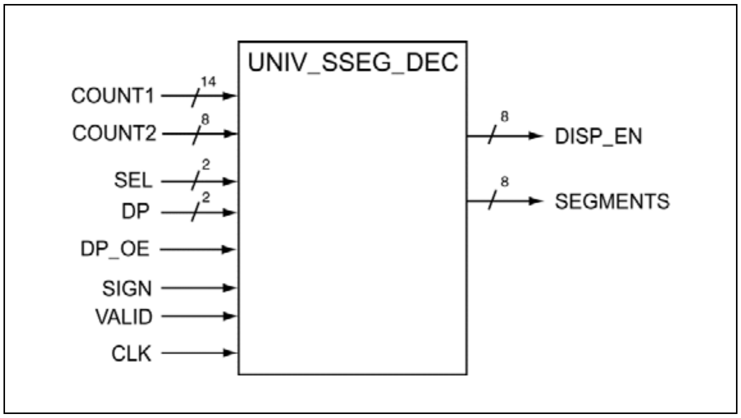


Figure 18: Black box diagram for the USSDM.

Signal	Width	Description
VALID	1	If VALID = ‘0’, four dashes appear on display If VALID = ‘1’, decimal number(s) appear on display
DP_OE	1	If DP_OE = ‘0’, no decimal point appears on display If DP_OE = ‘1’, one decimal point appears on display (see DP inputs)
DP	2	If DP = “00”, DP displayed on right-most 7-seg display If DP = “01”, DP displayed on middle-right 7-seg display If DP = “10”, DP displayed on middle-left 7-seg display If DP = “11”, DP displayed on left-most 7-seg display
SEL	2	If SEL = “00”, displays one count [0,255] with optional '-' sign If SEL = “01”, displays two counts [0,99] (no sign) If SEL = “10”, displays one count [0,9999] (no sign) If SEL = “11”, displays average academic administrator IQ
SIGN	1	If SIGN = 0, no minus sign appears If SIGN = '1', minus sign appears on left-most display NOTE: the sign can only be display in mode SEL = “00”
COUNT1	14	Count for SEL = ”00” ([0,255]) and SEL = “10” [0,9999] modes Count for right-most count ([0,99]) in SEL = “01” mode
COUNT2	8	Count for left-most count ([0,99]) for SEL = “01” mode

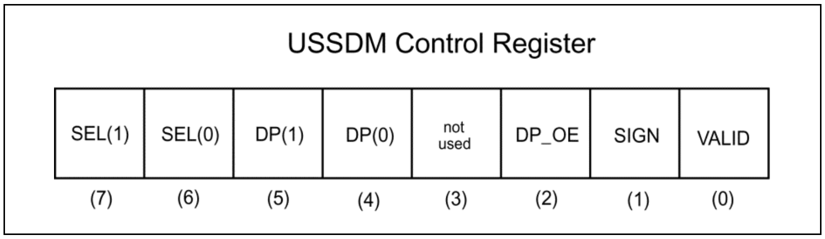
Table 9: Control options and description for the USSDM.

Using the USSDM with the RAT CPU

You must make several modifications to the RAT Wrapper in order to use the USSDM. You can interface the USSDM in many different ways, but you'll need at least four registers. I'll describe how I would do it to give you an idea how it needs to be done; your job is to make it work, so feel free to do it any way you feel comfortable with. The four registers you'll need to interface with the USSDM comprise of two registers for COUNT1, one register for COUNT2, and one register for the other control signals (not including the clock).

The COUNT1 and COUNT2 registers are straightforward and similar to other output registers you've been working with on the RAT CPU. I suggest tossing the other control signals into a single 8-bit "control" register. I put the quotes around control because registers such as this one are massively typical out there in microcontroller land. Put in another way, you the programmer must write to the bits in the USSDM control register in order to control the operation of the USSDM.

Figure 51 shows a possible implementation of a USSDM control register. The way this works is that the RAT CPU writes to this register under program control (OUT instruction). You use your RAT assembly language prowess to change only the bits you need to change while not altering the other bits. Since the USSDM is truly implemented as a register in the RAT Wrapper, the values you write to it would be persistent because they are officially registered.



USSDM Control Register

SEL(1)	SEL(0)	DP(1)	DP(0)	not used	DP_OE	SIGN	VALID
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

Figure 44: A possible implementation of the USSDM control register.

COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

CPE 233 External RAT Module

The VGA Module with the DrawDot Program

Objectives:

- To provide a basic operational over view of the VGA standard
- To describe the functionality and use of the VGA module
- To describe the functionality of the "drawdot" RAT assembly language program

Somewhat Meaningful Comments:

Video Graphics Array, or VGA, is a video standard that was once the latest and greatest thing when it came to computer monitors. Although we now consider VGA monitors as old technology in the context of modern high-definition monitors, they are still great learning tools as they are not overly complicated to work with. This document describes all I can currently recall about VGA monitors, particularly as they relate to how you may want to use this module in CPE 233. I'm writing this from my simple-minded understanding of things; what I know is based primarily on a high-level user approach. If you want more details, check a more useful and knowledgeable source (one great sources is the reference manual for the Nexys2 development board).

The basic job of a monitor is to display stuff on its screen. The screen is typically modeled as a two-dimensional grid (which I like calling "the raster" because it makes it seem like I actually know what I'm talking about). The smallest writable part of the raster is an entity referred to as a *pixel*. The VGA raster has 480 rows and 640 columns of pixels (by definition of VGA). To make an image appear on the raster, you must provide the pixels with meaningful data; the aggregate of the complete set of pixels forms something meaningful.

The data associated with the pixels is typically stored in a memory somewhere; this memory is often referred to as the *frame buffer*. The data in the frame buffer is of course stored as digital data, but the data is eventually converted to analog data so that it can be magically used to "light up" a pixel on the screen. The notion of turning on a pixel means that you're assigning it a color based on the data stored in the frame buffer. The color of a pixel is obtained by adding various amounts of the three primary colors of red, green, and blue, commonly referred to as RGB. Thus, the data stored in the frame buffer is typically some combination of three digital values representing the subsequent intensities of the three primary colors. The frame buffer must be large enough to hold one "set" of data for each pixel. The data in the frame buffer is thus color data stored in such a way that each data location is associated with a single pixel on the raster. The trick here is that if you want the screen to display a buttload of colors, you're going to have to get a larger memory for the frame buffer.

In the end, the "VGA Driver" consists of two main modules: the frame buffer, and the *other module*. The frame buffer holds digital data representing colors for the raster while the "other module" is responsible for generating the special timing requirements of the VGA monitors. While the memory is just a memory, the "other module" is a bit more complicated. The VGA Driver continually reads from the frame buffer and works with the hardware in the actual monitor to light-up the correct pixels at the correct time to make an image appear on the display.

The CPE 233 VGA Driver

The remainder of this write-up describes the VGA driver used in CPE 233, which is an even more simplified version of a standard VGA monitor. The CPE 233 VGA driver has a lower *resolution* than the standard VGA monitor. What this means is that while a typical VGA display measures 640x480, the dimensions of the VGA we use in CPE 233 is 40x30. We arrive here by growing the size of the typical pixel from just a measly pixel, to a super-pixel, which comprises of a 16x16 block of normal pixels (do the math brothers and sisters). Note that the *aspect ratio*, or the ratio of the number of columns to the number of rows, truly does not change. This is still a VGA monitor, but the provided hardware only supports a 40x30 grid of 16x16 super-pixels.

The general operation of the VGA driver comprises of writing 8-bit color data to the VGA's 40x30 frame buffer. Note that the 8-bit color data provides you with the possibility of 256 colors. We'll shortly fill in some of the lower-level details. For starters, Figure 45 shows the block diagram associated with the provided VGA Driver module. I included this in the document for historical purposes; the diagram has some errors and is generally confusing. The important things to notice about Figure 45 are the following:

- The frame buffer is a 2Kx8 memory. The memory requires 2K of address space in order to hold the 30x40 (=1200) bytes of color data. Recall that colors with this VGA driver are represented by an 8-bit value.
- The diagram lists the actual make-up of the 8-bit color data (more on that later).
- The outputs of the "other module" on the right side of the diagram connect to your actual VGA monitor. These signals are responsible of providing the required driver signals to the VGA monitor's hardware.

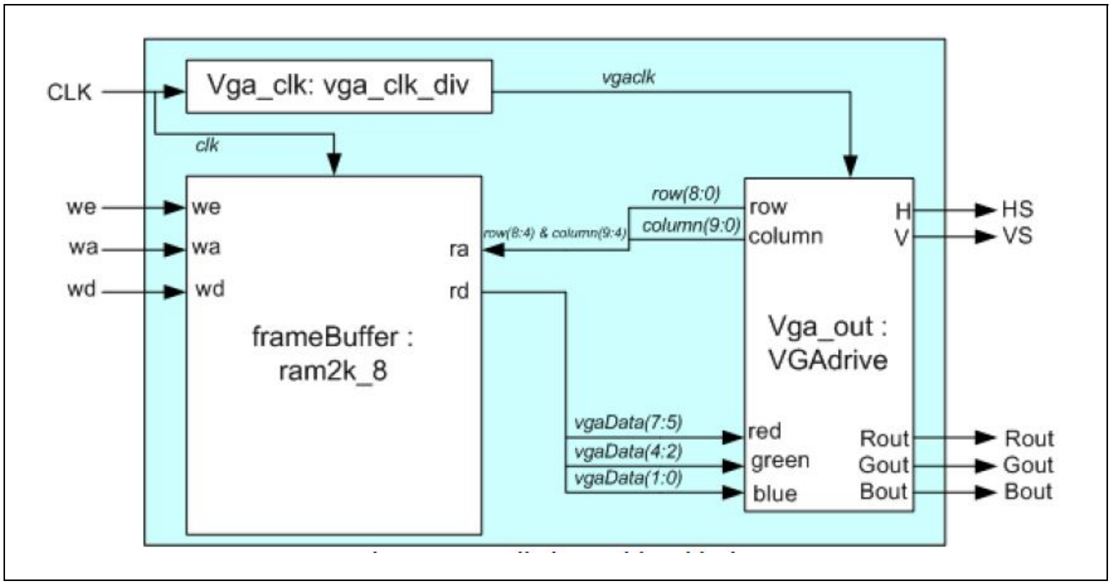


Figure 45: Black box diagram provided for the VGA Driver.

Figure 46 shows the raster associated with CPE 233 VGA driver. The main item of interest here is that the "origin" of the raster, or the (0,0) location, is in the upper-left corner of the raster. Thus, the address of the rows increase in value as you move downwards. Each of the squares in Figure 46 represents a 16x16 super-pixel, so the adjusted resolution is 40 columns by 30 rows. The ugly sets of three dots are masquerading as ellipse so as not to draw all the super-pixels and their associated location numbers.

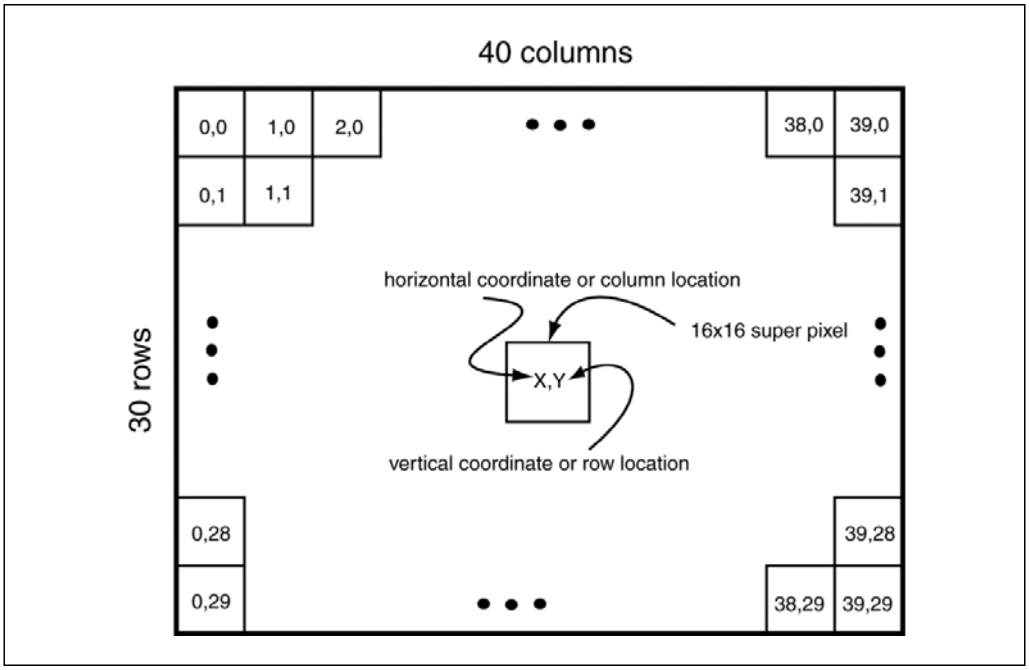


Figure 46: Adjusted resolution raster for the CPE 233 VGA Driver Module.

Figure 47 shows the format of the 8-bit color data for the CPE 233 VGA Driver. Note that this value is an 8-bit value, so you can have a total of 256 unique colors. Each of the individual colors have binary weights associated with the number, with the higher the index, the more of that color will be in the image. In official image processing terms, the color components in 8-bit color data are specified using intensity levels for the components. In this way, a single 8-bit color value contains eight levels of red and green, and two levels of blue. Another way to look at it is that the RED(2) bit can provide twice as much red component as RED(1) bit.

8-bit Color Register

RED(2)	RED(1)	RED(0)	GRN(2)	GRN(1)	GRN(0)	BLU(1)	BLU(0)
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

Figure 47: Color register data showing the red, green, and blue components.

The DrawDot Assembly Language Program

Enough of the lower-level details; let's talk about the high-level details, which is the stuff you'll need to know to do something interesting on the display. This section describes the individual modules of the famous "drawdot" program, which you we provide for you. This description does not include the entire program, but most of the important modules are here for your viewing pleasure.

The drawdot.asm program does four main things. Keep in mind the heart of this program is the drawing of a single "dot", which in this case is a 16x16 super-pixel. Here are the four things the drawdot program does; a description of each main thing follows.

1. Draws the background a given color
2. Draws a single dot
3. Draws a horizontal line
4. Draws a vertical line
5. Does nothing (thus acting like an academic administrator)

Figure 48 shows the main initialization code and main program for the drawdot program. I consider the init code to do all the work; the main code simply does nothing (tired joke deleted). The init code draws a background, draws a single dot, draws a horizontal line, and then draws a vertical line. We'll discuss those subroutines in later figures.

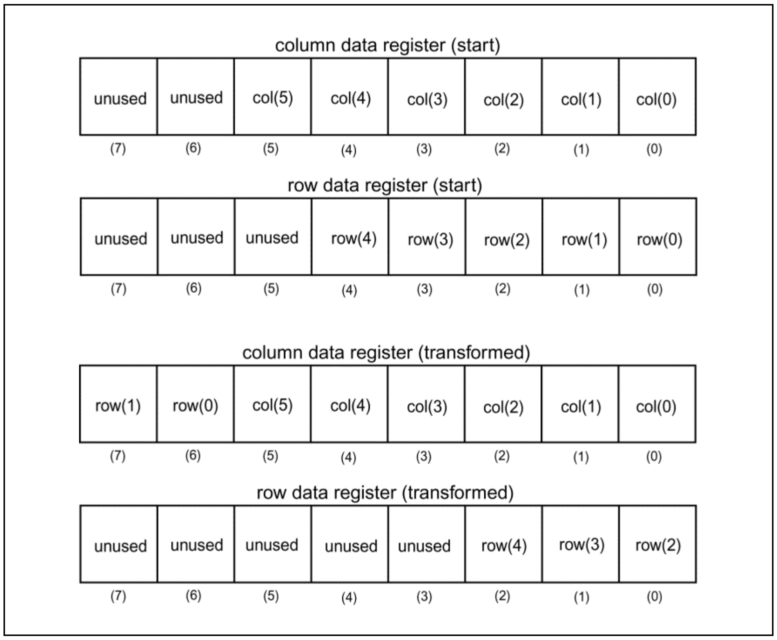
```
;-------------------------------------------------------------------
init:
         CALL   draw_background         ; draw using default color

         MOV    r7, 0x0F                ; generic Y coordinate
         MOV    r8, 0x14                ; generic X coordinate
         MOV    r6, 0xE0                ; color
         CALL   draw_dot                ; draw red square

         MOV    r8,0x01                 ; starting x coordinate
         MOV    r7,0x12                 ; start y coordinate
         MOV    r9,0x26                 ; ending x coordinate
         CALL   draw_horizontal_line

         MOV    r8,0x08                 ; starting x coordinate
         MOV    r7,0x04                 ; start y coordinate
         MOV    r9,0x17                 ; ending x coordinate
         CALL   draw_vertical_line

main:    AND    r0, r0                  ; nop
         BRN    main                    ; continuous loop
;------------------------------------------------------------------
```

Figure 48: The init and main code for the drawdot program. .

Figure 49 show the draw_dot subroutine. The main purpose of this subroutine is to transform pixel location data into a form that the VGA Driver module requires. There are 40 columns, which requires a six bits number to represent, while the 30 rows can be represented using five bits. This subroutine shifts the lower two bits of the row data into the upper two bits (which are unused) of the column data, which is the particular format required by the VGA Driver. Figure 50 shows a visual representation of what the previous sentences was attempting to convey. At end of the subroutine, the adjusted location data (divided into high address and low address) and the color data is then written to their associated registers. Keep in mind that the RAT Wrapper defines the registers being that this subroutine writes to.

```
;---------------------------------------------------------------------
;- Subrountine: draw_dot
;-
;- This subroutine draws a dot on the display the given coordinates:
;-
;- (X,Y) = (r8,r7)  with a color stored in r6
;-
;- Tweaked registers: r4,r5
;---------------------------------------------------------------------
draw_dot:
           MOV   r4,r7         ; copy Y coordinate
           MOV   r5,r8         ; copy X coordinate

           AND   r5,0x3F       ; make sure top 2 bits cleared
           AND   r4,0x1F       ; make sure top 3 bits cleared

           ;--- you need bottom two bits of r4 into top two bits of r5
           LSR   r4            ; shift LSB into carry
           BRCC  bit7          ; no carry, jump to next bit
           OR    r5,0x40       ; there was a carry, set bit
           CLC                 ; freshen bit, do one more left shift

bit7:      LSR   r4            ; shift LSB into carry
           BRCC  dd_out        ; no carry, jump to output
           OR    r5,0x80       ; set bit if needed

dd_out:    OUT   r5,VGA_LADD   ; write low 8 address bits to register
           OUT   r4,VGA_HADD   ; write hi 3 address bits to register
           OUT   r6,VGA_COLOR  ; write data to frame buffer
           RET
; --------------------------------------------------------------------
```

Figure 49: The draw_dot subroutine for the drawdot program.



column data register (start)

unused	unused	col(5)	col(4)	col(3)	col(2)	col(1)	col(0)
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

row data register (start)

unused	unused	unused	row(4)	row(3)	row(2)	row(1)	row(0)
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

column data register (transformed)

row(1)	row(0)	col(5)	col(4)	col(3)	col(2)	col(1)	col(0)
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

row data register (transformed)

unused	unused	unused	unused	unused	row(4)	row(3)	row(2)
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

Figure 50: The draw_dot subroutine for the drawdot program.

Figure 51 shows the draw_vertical_line subroutine. This subroutine comprises of a given number of call to the draw_dot subroutine, which makes sense as a line is nothing more than a series of dots. In particular, this subroutine draws dots at the same column location while adjusting the row location with each subsequent draw_dot subroutine call. One thing worthy of note in this subroutine is that the ending location of the line must be adjusted (it is incremented by one) to account for the particular implementation of the underlying iterative structure.

```
;---------------------------------------------------------------------
;-  Subroutine: draw_vertical_line
;-
;-  Draws a horizontal line from (r8,r7) to (r8,r9) using color in r6.
;-   This subroutine works by consecutive calls to drawdot, meaning
;-   that a vertical line is nothing more than a bunch of dots.
;-
;-  Parameters:
;-   r8  = x-coordinate
;-   r7  = starting y-coordinate
;-   r9  = ending y-coordinate
;-   r6  = color used for line
;-
;- Tweaked registers: r7,r9
;--------------------------------------------------------------------
draw_vertical_line:
         ADD    r9,0x01         ; go from r7 to r9 inclusive

draw_vert1:
         CALL   draw_dot        ; draw tile
         ADD    r7,0x01         ; increment row (y) count
         CMP    r7,R9           ; see if there are more rows
         BRNE   draw_vert1      ; branch if more rows
         RET
;--------------------------------------------------------------------
```

Figure 51: The draw_vertical_line subroutine for the drawdot program.

Figure 52 shows the draw_horizontal_line subroutine. Once again, we accomplish the drawing of a horizontal line by drawing dots across a particular row. This subroutine is simile to the draw_vertical_line subroutine, so there is not much to say here.

```
;--------------------------------------------------------------------
;-  Subroutine: draw_horizontal_line
;-
;-  Draws a horizontal line from (r8,r7) to (r9,r7) using color in r6.
;-   This subroutine works by consecutive calls to drawdot, meaning
;-   that a horizontal line is nothing more than a bunch of dots.
;-
;-  Parameters:
;-   r8  = starting x-coordinate
;-   r7  = y-coordinate
;-   r9  = ending x-coordinate
;-   r6  = color used for line
;-
;- Tweaked registers: r8,r9
;--------------------------------------------------------------------
draw_horizontal_line:
        ADD    r9,0x01          ; go from r8 to r9 inclusive

draw_horiz1:
        CALL   draw_dot         ; draw tile
        ADD    r8,0x01          ; increment column (X) count
        CMP    r8,r9            ; see if there are more columns
        BRNE   draw_horiz1      ; branch if more columns
        RET
;--------------------------------------------------------------------
```

Figure 52: The draw_horizontal_line subroutine for the drawdot program.

Figure 53 shows the draw_background subroutine. This subroutine comprises of 30 calls to the darw_horizontal_line routine, which fills the entire raster with horizontal lines. In this way, the draw_background subroutine writes to every available super-pixel location, which make this a great approach to "erasing" the screen if you ever find yourself with such a need.

```
;---------------------------------------------------------------------
;-  Subroutine: draw_background
;-
;-  Fills the 30x40 grid with one color using successive calls to
;-  draw_horizontal_line subroutine.
;-
;-  Tweaked registers: r13,r7,r8,r9
;----------------------------------------------------------------------
draw_background:
         MOV   r6,BG_COLOR              ; use default color
         MOV   r13,0x00                 ; r13 keeps track of rows
start:   MOV   r7,r13                   ; load current row count
         MOV   r8,0x00                  ; restart x coordinates
         MOV   r9,0x27

         CALL  draw_horizontal_line     ; draw a complete line
         ADD   r13,0x01                 ; increment row count
         CMP   r13,0x1E                 ; see if more rows to draw
         BRNE  start                    ; branch to draw more rows
         RET
;---------------------------------------------------------------------
```

Figure 53: The draw_background subroutine for the drawdot program.

COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

CPE 233 External RAT Module

The Draw_Blit Driver Program

Objectives:

- To describe some of the important guidelines when working with graphics
- To describe the basic functionality of the *draw_blit.asm* program

Somewhat Meaningful Comments:

There are typically many ways to “do things” in the world of digital design. While all of these approaches “get the job done”, there generally preferred ways to do most things, with the preference being the most time efficient or space efficient, etc. The notion here is that you spend most of your time doing things that have already been done before, so you might was well take the “best” approach (or at the least you should be aware of the best approach). Keep in mind that after you graduate and are making the big bucks, you’ll be faced with a deadline to get your company’s product out the door. As the deadline nears, you’ll probably discard all the programming and documentation skills you learned in school in an effort to actually complete the project at hand.

There are right and wrong ways to do graphics also. The graphics you may be faced with are writing to the 40x30 VGA screen for your CPE 233 project. This document does not attempt to provide you with everything anyone would want to know about graphics programming, but it hopefully provides you with some clues on how to improve the quality of your CPE 233 project.

Brief VGA Overview:

There is another document in this series that describes the VGA driver; you should already be familiar with that document. In case you’re not, this section provides a somewhat brief version of the VGA driver as it applies to the draw_blit program.

The CPE 233 version of the VGA driver comprises of two main sub-modules; the frame buffer and the VGA driver. The VGA driver is in a world of its own, so refer to the Nexys2 reference manual for a better description of the VGA driver’s responsibilities. In other words, the VGA driver module is something you don’t need to worry about. The frame buffer is what we’re interested in, as that is what programmers interact with to produce something interesting on the display. The frame buffer is a chunk of memory that holds the color values for each of the pixels in the VGA raster (grid). The CPE 233 frame buffer contains one byte of color data for each pixel in the raster, which allows you to write one of 256 colors to the display.

The VGA driver module continually reads from the frame buffer and drives the VGA display. When you write a new value to the frame buffer, it immediate appears on the display (well, not really immediately, but it’s faster than your eye and brain can detect. Your mission for the CPE 233 VGA module is to interface with the frame buffer. The document describing the frame buffer has more details in case you’re interested.

Simulating Movement on the VGA Display:

In real life, things just sort of move around. In VGA-land, you the programmer, are in charge of making things *appear* as if they are moving on the display. Making something appear as if it is moving on the display is relatively simple in theory (though actual practice is another issue). Moving something on the display is a three-step process:

1. Figure out what you're going to move and where you're going to move it to. This step involves either internal program control of external inputs to direct the "what" and "where".

2. Remove, or *erase* the object you've decided to move. Erasing means that you'll write over the object with the "background color", which effectively makes the object disappear on the display.

3. Draw the object in a new location on the display.

While this approach seems rather choppy, you're brain is too slow to see the choppiness. So in this case, choppiness is not a bad thing due to characteristics of your human visual system. Though the above description is choppy, you certainly did not notice anything choppy about the multiplexed displays (the retinal persistence thang). In truth, if the object you're moving does not move too far or move too often, you can certainly create the illusion of smooth movement.

The Right and "Less Right" Approach to Simulating Movement:

When all is said and done, you're still programming a computer. In CPE 233, your RAT CPU is not usually too busy to do the stuff you tell it to do, but in real life, you're generally going to push your CPU to its operating limits. The reason that your CPU is going to be pushed to its limits is because if it was not being pushed to it limits with your given application, you may as well use a "cheaper" device. The general notion here is that slower, "less capable" devices are cheaper so there is incentive to keep the CPU performing meaningful stuff as much as possible[1].

The approach you should strive to take in CPE 233 is to do it right. Or at the very least, you should know how it can be done right. In truth, if you drive your graphics in the "less right" way, it's not a big deal. The right way to drive your display in the context of things that need to move on the display is to only move the things that need moving. The goodness in this approach that you'll be saving processing time by not changing things that don't require changing. The alternative, or the "less right" approach, is to redraw the entire screen each time anything on the display moves. The problem with this approach is that is makes no sense to redraw something that does not change.

The truth is that it is relatively easy to take the "right" approach if you only have a few objects that require moving. But once you have more than a few objects that need moving, it requires less computational effort to draw the entire screen even if only one things makes one small movement. Part of the computational effort thing relates to the fact that most people are new to assembly language programming; another part of the problem is that assembly language is generally not the best language to do the bulk of your graphics programming.

The Draw-Blit Program:

You've probably been wondering what is a "blit". I'm not sure where I got the term from; maybe it's not even the right term. But for CPE 233, a blit is an graphics object you can draw on the display. In that you can draw it on the display, you can generally make it appear to move by following the movement approach previously outlined in this document.

[1] However, in academia, the slower, less capable people become administrators. Worse yet, they come at a high cost in that their salaries are not commensurate with their actual throughput.

The purpose of the draw_blit program is to show you the “right” approach to graphics. The program consists of drawing a blit, which in this case is a 3x3 square, and allowing the user to move the blit around the display by entering letter commands from a keyboard. The draw_blit program provides a framework for you to do similar actions with your own graphics program for you CPE 233 project. The only slightly new thing in the draw_blit program is the notion that the program prevents the user from moving the blit off the edge of the display.

The draw_blit program really does not provide much in the way of new stuff for you. The program itself is primarily a combination of the draw_dot program and the keyboard driver module, both of which are documented elsewhere in this series of documents. The only new modules for are the *draw_blit*, the *erase_blit*, and the bounds checking subroutines. We’ll talk about these later in this document.

Hardware Considerations for the Draw-Blit Program:

The only two additional modules required by the draw_blit program on the VGA Drive module and the keyboard drive module. Both of these modules are described in other documents, so we’ll spare you the details here. For your comfort, Figure 54 shows a schematic of the associated keyboard driver.

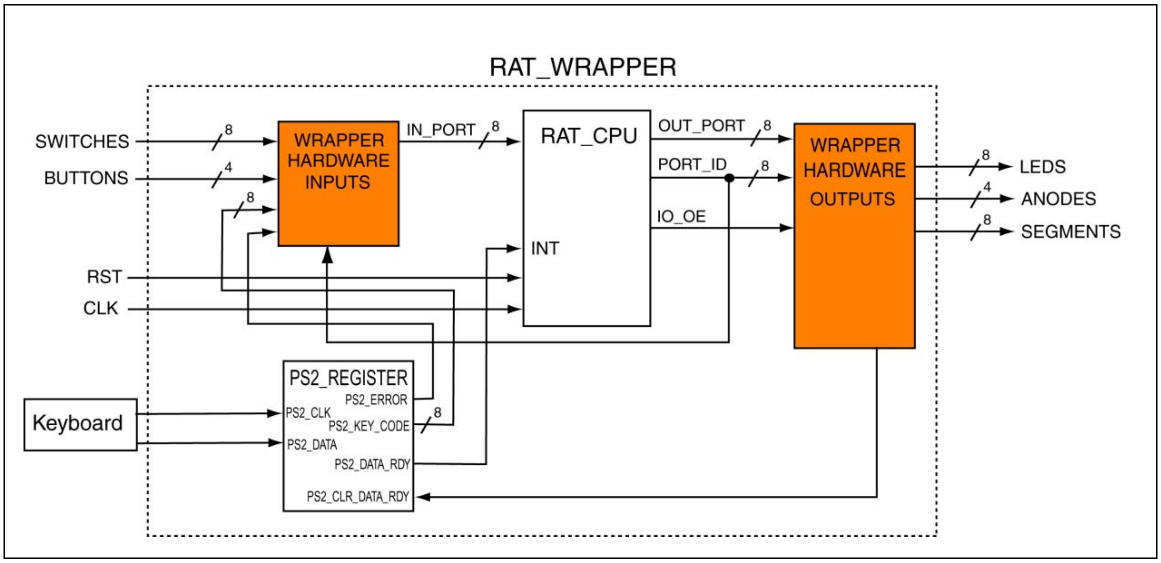


Figure 54: Black box of the RAT Wrapper showing the interface of the PS2_REGISTER.

Important Sections of the Draw-Blit Program:

The draw_blit program is once again an interrupt driven program. The program starts by drawing a background color on the display and then draws a blit in the center of the display. For the draw_blit program, the blit is a 3x3 square with a different color than the background, and it has a pixel of a different color in the middle of the 3x3 square. Not that this blit is nothing exciting, but it does provide you with a template to change the blit into something you find more interesting.

Figure 55 shows the foreground task of the draw_blit program. This fragment of code consists of some initialization code followed by a main task. The main task does nothing; in this case, doing nothing means that the program is essentially waiting for the user to hit a key on the keyboard, yet wasting clock cycles the entire time. Not too exciting folks.

```
;-------------------------------------------------------------------
; Foreground Task
;-------------------------------------------------------------------
Init:    MOV    r7, 0x0F        ;- starting y value (middle row)
         MOV    r8, 0x14        ;- starting x value (middle column)
         MOV    r15,0x00        ;- clear interrupt flag register

         MOV    r6, BG_COLOR    ;- bluish color
         CALL   draw_background ;- draw using default color
         CALL   draw_blit       ;- plop down initial shape
         SEI                    ;- allow interrupts

main:    CMP    r0, 0x00        ;- do nothing (nop)
         BRN    main            ;- dumb poll waiting for interrupts
;---------------------------------------------------------------------
```

Figure 55: The initialization and main code for the draw_blit program.

Figure 56 and Figure 57 shows the *draw_blit* and *erase_blit* subroutines, respectively. These subroutines are quite similar and probably should have been combined into one subroutine[2]. Don't allow these subroutines to intimidate you, they are relatively simple. The main responsibility of the draw_blit code is to draw a blit (what a surprise). The program calling the subroutines passes the location in which to draw the blit via registers r7 and r8. The subroutine saves r7 and r8 upon entering the subroutine and restores them upon exiting[3].

The bulk of the draw_blit subroutine comprises of writing each of the nine pixels in the 3x3 object. The subroutine does this via nine (3x3 does in fact equal nine) calls to the draw_dot subroutine after first adjusting the coordinates in order to draw the correct location in the 3x3 kernel. That's about it.

The erase_blit routine is strikingly similar in that it that it does all the same stuff as the draw_blit subroutine, except that it draws all of the pixels in the 3x3 kernel with the background color. Once again, you effectively make the blit go away (you erase it) by replacing the pixels associated with the blit with the background color.

[2] The notion here is that anytime you see repeated code, you should probably work to combine that code into one piece of code. This often requires a more personal programming effort, but the rewards of saving code space generally make it well worth your while.

[3] The code uses MOV instructions to save these registers because the program has the registers it is not using in the program. If you did not have these registers available, you would PUSH the r7 and r8 registers upon entering the subroutine and POP them upon exiting.

```
;---------------------------------------------------
;- Subroutine: Draw_blit
;-
;- The subroutine draws a 3x3 square centered at the
;- values in (r8,r7) <==> (x,y). The center of this
;- 3x3 is green; the outside edges are blue.
;---------------------------------------------------
Draw_blit:
         MOV   r25,r7       ; save current y location
         MOV   r26,r8       ; save current x location

         MOV   r6,GREEN     ; change color
         CALL  draw_dot     ; draw center

         MOV   r6,BLUE      ; change color

         SUB   r7,0x01      ; adjust coordinates
         SUB   r8,0x01
         CALL  draw_dot     ; NW pixel
         ADD   r8,0x01      ; adjust coordinates
         CALL  draw_dot     ; N pixel
         ADD   r8,0x01      ; adjust coordinates
         CALL  draw_dot     ; NE pixel

         ADD   r7,0x01      ; adjust coordinates
         CALL  draw_dot     ; E pixel
         SUB   r8,0x02      ; adjust coordinates
         CALL  draw_dot     ; W pixel

         ADD   r7,0x01      ; adjust coordinates
         CALL  draw_dot     ; SW pixel
         ADD   r8,0x01      ; adjust coordinates
         CALL  draw_dot     ; S pixel
         ADD   r8,0x01      ; adjust coordinates
         CALL  draw_dot     ; SE pixel

         MOV   r7,r25       ; restore current y location
         MOV   r8,r26       ; restore current x location
         RET                ; later dude
;-------------------------------------------------------------
```

Figure 56: The draw_blit subroutine.

```
;------------------------------------------------------------
;- Subroutine: Erase_blit
;-
;- The subroutine erases 3x3 square centered at the
;- values in (r8,r7) <==> (x,y). Ereasing is done by
;- writing the background color at the given coordinates.
;------------------------------------------------------------
Erase_blit:
         MOV   r25,r7       ; save current y location
         MOV   r26,r8       ; save current x location

         MOV   r6,BG_COLOR  ; load background color
         CALL  draw_dot     ; draw center

         SUB   r7,0x01      ; adjust coordinate
         SUB   r8,0x01      ; adjust coordinate
         CALL  draw_dot     ; NW
         ADD   r8,0x01      ; adjust coordinate
         CALL  draw_dot     ; N
         ADD   r8,0x01      ; adjust coordinate
         CALL  draw_dot     ; NE

         ADD   r7,0x01      ; adjust coordinate
         CALL  draw_dot     ; E
         SUB   r8,0x02      ; adjust coordinate
         CALL  draw_dot     ; W

         ADD   r7,0x01      ; adjust coordinate
         CALL  draw_dot     ; SW
         ADD   r8,0x01      ; adjust coordinate
         CALL  draw_dot     ; W
         ADD   r8,0x01      ; adjust coordinate
         CALL  draw_dot     ; W

         MOV   r7,r25       ; restore current y location
         MOV   r8,r26       ; restore current x location
         RET
;---------------------------------------------------------
```

Figure 57: The erase_blit subroutine.

Figure 49 shows four subroutines, which prevent the user from driving the blit off the side of the display. The basic notion of these four subroutines is to check the given bounds, which in this case are the edges of the display, if the user requests that the blit off the side of the display, the subroutines effectively ignored the request, and control returns to the calling program. Nothing too exciting here either.

```
;------------------------------------------------------------
;- These subroutines add and/or subtract '1' from the given
;- X or Y value, depending on the direction the blit was
;- told to go. The trick here is to not go off the screen
;- so the blit is moved only if there is room to move the
;- blit without going off the screen.
;------------------------------------------------------------
sub_x:   CMP   r8,LO_X    ; see if you can move
         BREQ  done1
         SUB   r8,0x01    ; move if you can
done1:   RET

sub_y:   CMP   r7,LO_Y    ; see if you can move
         BREQ  done2
         SUB   r7,0x01    ; move if you can
done2:   RET

add_x:   CMP   r8,HI_X    ; see if you can move
         BREQ  done3
         ADD   r8,0x01    ; move if you can
done3:   RET

add_y:   CMP   r7,HI_Y    ; see if you can move
         BREQ  done4
         ADD   r7,0x01    ; move if you can
done4:   RET
;---------------------------------------------------------
```

Figure 58: The four subroutines that perform bounds checking associated with movement requests.

Figure 59 shows the interrupt service routine portion of the draw_blit program. The two main responsibilities of the ISR are to handle the strangeness associated with a key press on a PS/2 keyboard (once again, see the document in this series describing the keyboard driver for more details) and the decoding of a valid key press. When the ISR detects a key press associated with a valid direction, it first erases the blit at its current location by a call to the erase_blit subroutine. After the program erases the blit, it uses the direction to adjust the coordinates of the blit according the directional move request by a call the coordinate adjusting subroutines. Once the program adjusts the coordinates blit coordinates, the code draws the blit at its new location with a call to the draw_blit subroutine. Once the subroutine completes all of the important work, the subroutine writes to the PS2_REGISTER in order to allow more interrupts sponsored by the keyboard and its associated human.

One final comment. It's not always a good idea to do all the work in the ISR. The main issue here is that while you're in the ISR, the interrupts are generally disabled. For your purposes, this does not really matter much. But in real life, there may be some important devices connected to the interrupts that require immediate service upon request. In this case, it's not going to get immediate service if the associated interrupts are disabled. You may think that you can simply unmask the interrupts once you enter the ISR, but… the notion of nested interrupts brings even the most experienced embedded systems programmers to their knees[4].

[4] A familiar position for all upward-bound academic administrators.

```
;--------------------------------------------------------------
; Interrupt Service Routine - Handles Interrupts from keyboard
;--------------------------------------------------------------
; Sample ISR that looks for various key presses. When a useful
; key press is found, the program does something useful. The
; code also handles the key-up code and subsequent re-sending
; of the associated scan-code.
;
; Tweaked Registers; r2,r3,r15
;--------------------------------------------------------------
ISR:      CMP   r15, int_flag        ; check key-up flag
          BRNE  continue
          MOV   r15, 0x00            ; clean key-up flag
          BRN   reset_ps2_register

continue: IN    r2, PS2_KEY_CODE     ; get keycode data

move_up:  CMP   r2, UP               ; decode keypress value
          BRNE  move_down
          CALL  Erase_blit
          CALL  sub_y                ; verify move is possible
          CALL  Draw_blit            ; draw object
          BRN   reset_ps2_register

move_down:
          CMP   r2, DOWN
          BRNE  move_left
          CALL  Erase_blit
          CALL  add_y                ; verify move
          CALL  Draw_blit            ; draw object
          BRN   reset_ps2_register

move_left:
          CMP   r2, LEFT
          BRNE  move_right
          CALL  Erase_blit
          CALL  sub_x                ; verify move
          CALL  Draw_blit            ; draw object
          BRN   reset_ps2_register

move_right:
          CMP   r2, RIGHT
          BRNE  key_up_check
          CALL  Erase_blit
          CALL  add_x                ; verify move
          CALL  Draw_blit            ; draw object
          BRN   reset_ps2_register


key_up_check:
          CMP   r2,KEY_UP            ; look for key-up code
          BRNE  reset_ps2_register   ; branch if not found

set_skip_flag:
          ADD   r15, 0x01            ; indicate key-up found

reset_ps2_register:                  ; reset PS2 register
          MOV    r3, 0x01
          OUT    r3, PS2_CONTROL
          MOV    r3, 0x00
          OUT    r3, PS2_CONTROL
          RETIE                      ; aloha Mr. Subroutine
;-------------------------------------------------------------------
```

Figure 59: The part of the interrupt service routine that pulses a bit in the PS/2 control register.

COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

CPE 233 External RAT Module

The PS/2 Keyboard Interface Module a Simple Driver Program

Objectives:

- To provide a basic operational overview of a generic PS2-type keyboard driver

Somewhat Meaningful Comments:

Long before USB interfaces started taking over the world, there were many other serial interfaces out there; PS/2 was one of them. The PS/2 was commonly used to interface typical personal computer-related items such as keyboards and mouse(s). There is a lot to say about the PS/2 interface, but this document provides only a quick overview that should help you apply this interface to your project. The PS/2 interface provides a way for two devices to communicate with each other; the CPE 233 PS/2 driver is currently only a one-way interface because it does not allow the RAT MCU to communicate directly with the keyboard. For more information, check various online resources. One particularly quick and good reference is in the reference manual for the Nexys2 development board.

The primary thing we're interested in for CPE 233 is the ability of using a keyboard to control the operation of the RAT MCU. The keyboard is able to do many things and some of these things can be rather complicated. But, what we're interested in is having the RAT MCU react to the simple pressing of keys. In this way, we can forget about more advanced items such as caps locks, shifted key presses, alt key presses, zombie killing, etc.

When you press a key on the keyboard, the keyboard sends a "scan code" associated with that key. A simple key press and relatively quick release causes the keyboard to do three things. First, when the key is pressed, the keyboard sends the 8-bit scan code associated with that key across the PS/2 interface. Second, when the user releases the key, the keyboard sends a "key-up" code (0xF0). Third, after the keyboard sends the key-up code, the keyboard resends the scan code associated with the original key press. In this way, every valid key press causes a minimum of three data transmissions. The other related scenario is when the user presses a key and keeps that key pressed. In this scenario, the keyboard resends the original scan code at about a 100ms rate until the user releases the key, at which time the keyboard sends the key-up code followed by a resending of the original scan code one final time.

The two scenarios above are what the current CPE 233 PS/2 driver and associated assembly language driver program nicely handles. If you need functionality beyond that, you'll need study the assembly code further and/or modify it to your needs.

The CPE 233 PS/2 Driver

The PS/2 driver for CPE 233 is a serial-to-parallel converter. This means that it gathers up the serial data sent by the keyboard and puts it into an 11-bit register; the data in the register is parallel. The data sent from the keyboard is associated with a particular key press. Each of the keys on a keyboard is assigned a scan code; anytime you press one of these keys, one or more pieces of code is sent from the keyboard to whatever device is monitoring the keyboard, which in your case is going to be the RAT MCU. The data associated with a key press is only eight bits, but the keyboard attaches three extra bits, including a start bit, a parity bit, and a stop bit. Asynchronous interfaces such as PS/2 typically require “framing” bits such as the start and stop bits, while the parity bit provides a small amount of error detection capabilities. Figure 61 shows the PS2_REGISTER data format, which we’ll describe in more detail later in this description. Additionally, the sending device is also responsible for providing a clock signal to the receiving device. The CPE 233 PS/2 driver defines this signal as bi-directional, but for the case of CPE 233, you won’t have a need to send data to the keyboard.

Figure 45 shows the black box diagram for the hardware component of the PS/2 drive which we refer to as the “PS2_REGISTER”. Table 10 provides a brief description of the signals associated with the PS2_REGISTER. You can gather further details by looking at the associated VHDL model for the PS2_REGISTER and the associated RAT Wrapper. Further explanation appears later in this document.

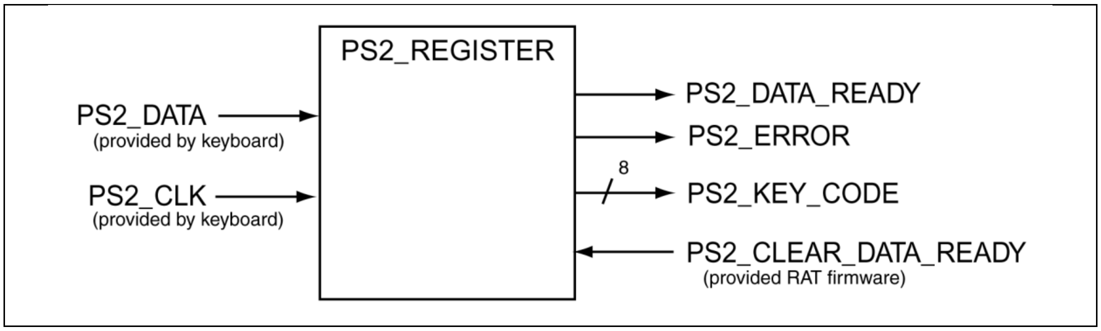
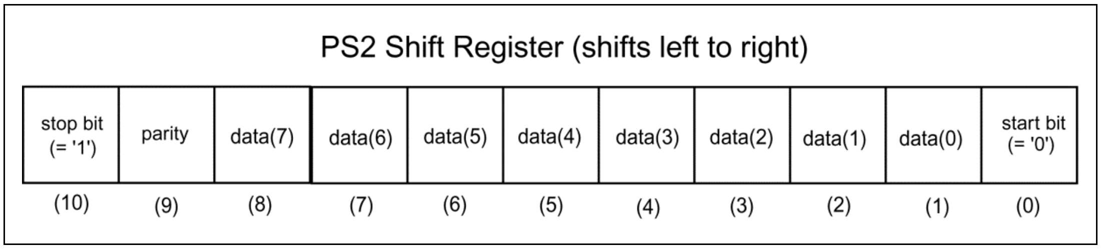


Figure 60: Black box diagram for the keyboard Driver.

The CPE 233 PS/2 driver is effectively a shift register with some special control functions added. Figure 61 shows the shift register contents after a successful data transmission. The PS2_REGISTER module primarily contains a shift register that shifts in the data sent from the keyboard in the left-to-right direction. Because of the directionality of the shift, the first bit sent by the keyboard is the start bit, which is always set to ‘0’. When the data transmission is complete, the complement of the right-most bit of the shift register, which is the start bit, officially triggers an interrupt on the RAT MCU. The keyboard includes a parity bit based on the all the transmitted bit except for the stop bit. We refer to the data transmission as being “framed” by the combination of the start and stop bits. The data bits in Figure 61 represent the 8-bit scan code sent from the keyboard resulting from a key press.


PS2 Shift Register (shifts left to right)

stop bit (= '1')	parity	data(7)	data(6)	data(5)	data(4)	data(3)	data(2)	data(1)	data(0)	start bit (= '0')
(10)	(9)	(8)	(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

Figure 61: The PS2_REGISTER contents after a successful data transmission.

Table 10 provides a list and description of the signals associated with the PS/2 interface. We’ll describe some of these signals later in this document, so plan on referring back to Table 10as you read onwards.

Signal Name	Dir	Description
PS2_CLK	bi-dir	The clock used to synchronize data transmissions. The keyboard provides this clock in the CPE 233 PS/2 driver.
PS2_DATA	in	The data sent by the keyboard associated with a key press. This data is serial; the PS2_REGISTER module parallelizes the data and makes it available on the module's outputs.
PS2_CLR_DAT_RDY	in	This bit needs to be set to '1', then set to '0' to enable the PS2_REGISTER to accept data. After every data transmission, this input needs to be pulsed to allow more data to arrive. The RAT MCU is responsible for pulsing this bit under program control.
PS2_DATA_READY	out	This signal is actually the first bit sent on the PS2_DATA line as it is the start bit of the data transmission. This bit is always a '0' and ends up being the LSB of the 11 bits sent across the PS2_DATA line. When the shifting is complete, the bit triggers an interrupt on the RAT MCU as it is connected to the RAT MCU's interrupt input pin.
PS2_ERROR	out	This bit indicates there was a parity error with a data transmission.
PS2_KEY_CODE	out	This is the parallelized byte of data associated with the key press and is input to the RAT MCU under program control.

Table 10: Name, direction, and brief description of the PS2_REGISTER signals.

Figure 54 shows a diagram of the RAT Wrapper including the proper interface with the PS2_REGISTER module. You should use Figure 54 in conjunction with Table 10 to convince yourself that the operation of the PS2_REGISTER and its interface to the RAT MCU is relatively straight-forward. Here are a few items relating to Figure 54 that are of great significance.

- The two outputs from the keyboard are the data and clock inputs. Recall that the sending device in a PS/2 interface provides the clocking signal, which the PS2_REGISTER uses to synchronize the shifting of data into the PS2_REGISTER's internal shift register.

- The "Wrapper Hardware Inputs" device is essentially a MUX that is RAT MCU controls via the RAT's PORT_ID output. In this way, the RAT MCU can input under program control either the associated scan code (PS2_KEY_CODE) or an error code (PS2_ERROR) associated with the PS2_REGISTER module. Note that other inputs to this MUX include the button and switch inputs, so this should seem somewhat familiar to you.

- The PS2_DATA_READY signal ties directly to the RAT's interrupt input. The PS2_DATA_READY signal is the start bit of the data transmission; this signal is complemented and connected to the RAT's interrupt input.

- The "Wrapper Hardware Outputs" provide both a selection mechanism and a register for various outputs. Note that the Wrapper contains external outputs for the LEDS, ANODES, and SEGMENTS. Also note that the PS2_CLEAR_DATA_READY is effectively in internal output from the Wrapper Hardware Module. The RAT MCU must send a pulse to the PS2_REGISTER under program control in order to unassert the PS2_DATA_READY signal, which is the signal that originally caused the interrupt. This "clearing" action is why the signal has the name "PS2_CLEAR_DATA_READY", as the PS2_REGISTER uses a pulse on this line to unassert the interrupt and prepare the PS2_REGISTER to receive more data from the keyboard.

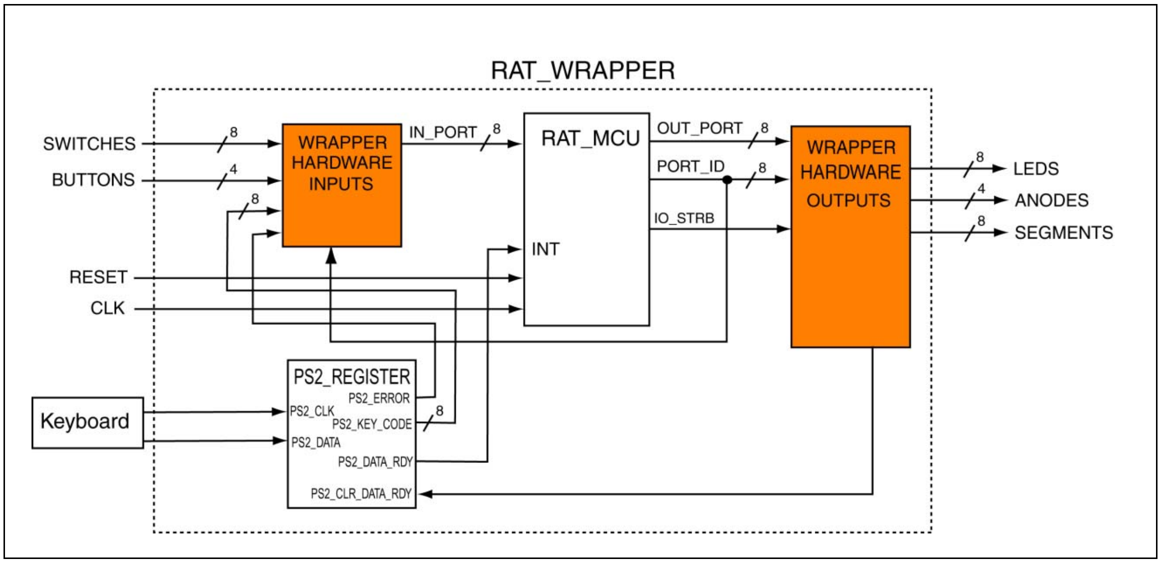


Figure 62: Black box of the RAT Wrapper showing the interface of the PS2_REGISTER.

Yet More Important Stuff

Here is some more information that will help you get this experiment up and running. You have a choice of the level of understanding you can go with regarding the keyboard interface, but the items listed below are stuff you need to be aware of.

- The Nexys2 board does indeed contain a six-pin DIN connector that is common in later devices using a PS/2 interface. This is the connector on the lower-left hand side of the Nexys2 board. Back in the old days, the PS/2 connectors were the standard DB9 connectors, which is the connector above the USB connector on the Nexys2 board.

- The PS/2 interface requires that the two interfaced devices to be using the same power. Most of the later PS/2 devices require 3.3V, while the older devices required 5.0V. For CPE 233, the Nexys2 board provides power to the keyboard through the PS/2 interface. Most of the keyboards floating around in the 20-100 & 20-132 labs require 3.3V. If your interface does not seem to be working, you may want to verify which voltage level the keyboard uses for power. There's a jumper on the lower-left side of the Nexys2 board that allows you to choose either 3.3V or 5.0V; be sure to check the Nexys2 reference manual for full details.

- There are two external connections that your RAT Wrapper needs to handle: the PS/2 data and clock lines. Both of these signals are declared as bidirectional for generic interface purposes, but you'll only be having the keyboard provide both the data and clock (thus making them uni-directional). Additionally, you'll need to modify your constraints file according to Figure 63.

```
NET "PS2C" CLOCK_DEDICATED_ROUTE = FALSE;

NET "PS2C" LOC = R12;
NET "PS2D" LOC = P11;
```

Figure 63: The additions to the constraints file as required by the CPE 233 Keyboard Driver.

The Provided Assembly Language Driver Program

There are only a few sections to the driver program. The driver program performs as follows. When you press either a ‘j’ or ‘k’ key, the corresponding key code appears on the development board’s LEDs. The scan code only stays on the LEDs while you continue pressing the button; the LEDs turn off when you release the button. If you press any key other than ‘j’ or ‘k’, nothing appears to happen on the development board. This idea behind this relatively simple program is to understand this program enough for you to do something useful when a button of your choosing is pressed. The other thing to keep in mind about his program is that the pressing of a button generates an interrupt, thus this program is interrupt driven.

Figure 55 shows the main code associated with the driver program. This code consists of an initialization of r15, followed by an enabling of the interrupts. The main code does nothing useful as it awaits interrupts caused by keyboard key presses. The driver program uses the r15 register as a flag, which we’ll describe later in this document.

```
;--------------------------------------------------------------------
init:    MOV   r15,0x00      ; clear key-up flag
         SEI                 ; enable interrupts

main:    AND   r21,r21       ; nop
         BRN   main          ; do relatively nothing
;--------------------------------------------------------------------
```

Figure 64: The initialization and main code for the keyboard driver program.

Figure 48 shows the interrupt service routine for the keyboard driver program. This code does two main things. First, it looks for the key presses of interest, which are arbitrarily ‘j’ and ‘k’ for this program. When the program detects a press of one of these letters, the ISR calls a subroutine that does something useful, which is light up LEDs in the case of this program. You’ll probably have the subroutines do something more useful that what this program does. Second, this code handles the issue of the key-up code by doing nothing except setting the r15 flag when it receives the key-up code, then skipping the next data transmission from the keyboard. When the code skips a data transmission because of a scan code following the key-up code, the program clears the r15 flag.

```
;---------------------------------------------------------------
; Interrupt Service Routine - Handles Interrupts from keyboard
;---------------------------------------------------------------
; Sample ISR that looks for two key presses only: 'j' and 'k'.
; If 'j' or 'k' are found, the keycode data is displayed on
; the LEDs; otherwise, the LEDs are turned off.
;
; Tweaked Registers; r3,r3,r15
;---------------------------------------------------------------
ISR:      CMP   r15, int_flag        ; check key-up flag
          BRNE  continue
          MOV   r15, 0x00            ; clearn key-up flag
          BRN   reset_ps2_register

continue: IN    r2, PS2_KEY_CODE     ; get keycode data

check_1:  CMP   r2,JJ                ; was 'j' pressed?
          BRNE  check_2
          CALL  disp_j
          BRN   reset_ps2_register

check_2:  CMP   r2,KK                ; was 'k' pressed?
          BRNE  key_up_check
          CALL  disp_k
          BRN   reset_ps2_register

key_up_check:
          MOV   r3,0x00              ; turn off LEDs
          OUT   r3,LEDS

          CMP   r2,KEY_UP            ; look for key-up code
          BREQ  set_skip_flag        ; branch if found

          BRN   reset_ps2_register


set_skip_flag:
          ADD   r15, 0x01            ; indicate key-up found
          BRN   reset_ps2_register
;--------------------------------------------------------------------
```

Figure 65: The main portion of the interrupt service routine.

Figure 66 shows two subroutines that "do something" when the key of interest is pressed. Once again, you'll most likely want to do something more interesting than send the associated scan code to the LEDs. These two subroutines are "hooks" (a standard programming term) that you're expected to modify in your own program so that they do something that your program requires.

```
;--------------------------------------------------------------------
; do something meaningful when particular keys are pressed
;--------------------------------------------------------------------
disp_j:   OUT   r2, LEDS
          RET

disp_k:   OUT   r2, LEDS
          RET
;--------------------------------------------------------------------
```

Figure 66: Two generic subroutines associated the pressing of a "key of interest".

Lastly, Figure 49 shows the code that sends out a pulse to the PS2_CLEAR_DATA_READY signal on the PS2_REGISTER. Sending a pulse remove the asserted interrupt from the RAT MCU and allows the keyboard to send more data to the PS2_REGISTER device.

```
;--------------------------------------------------------------------
; reset PS2 register which allow it to send more interrupts
;--------------------------------------------------------------------
reset_ps2_register:
          MOV    r3, 0x01
          OUT    r3, PS2_CONTROL
          MOV    r3, 0x00
          OUT    r3, PS2_CONTROL
          RETIE
;--------------------------------------------------------------------
```

Figure 67: The part of the interrupt service routine that pulses a bit in the PS/2 control register.

COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

CPE 233 External RAT Module

The Etch-A-Sketch Driver Program

Objectives:

- To provide a basic overview of a graphics program that emulates "Etch-a-Sketch" functionality
- To provide a module that allows for the testing of background colors

Somewhat Meaningful Comments:

In case you were wondering, an "Etch-a-Sketch" was a drawing toy that was popular in the 1960's, long before kids had cell phones they could toy with all day long. The main attributes of the Etch-a-Sketch are a screen to draw on, and two knobs to control drawing on the display. The unique characteristic of the Etch-a-Sketch was the notion that you could not "lift your pen" when you were drawing something on the display. The Etch-a-Sketch was simply a line drawing device, which used two knobs to control the horizontal and vertical coordinates of the point the user was drawing. The ramification of these characteristics is that if you wanted to create a line bifurcating from the middle of an existing line, you would need to backtrack through the existing line to the starting point of your new line. And of you did not like the image you had drawn, you simply faced the display towards the ground and wildly shook the Etch-a-Sketch device. The Etch-a-Sketch was an overall interesting and engaging low-tech toy from another era.

The characteristics of the Etch-a-Sketch are quite instructive for CPE 233 students new to graphics programming. The VGA computer display replaces the Etch-a-Sketch display while four direction keys on a computer keyboard replace the two drawing control knobs on the Etch-a-Sketch. The Etch-a-Sketch driver program borrows heavily from the draw_dot program and the keyboard driver program, both of which are described in other documents in this series.

The document and supporting files provide a simple program that you can use to learn about several useful aspects of working with graphics and interfacing with an external keyboard. The associated assembly language program provides a working program that interfaces with a keyboard. We intend that you use this package to develop an understanding of the topic such that you can modify the existing materials in such a way as to support your CPE 233 projects.

The Underlying Hardware:

The keyboard interface associated with the Etch-a-Sketch package uses a PS/2 driver. The PS/2 protocol is a generic serial interface standard that many computer peripherals use for their communication needs. A separate document in this series describes the important details of the PS/2 driver; we won't regurgitate them again here. The main things you need to know about the PS/2 driver are that it is basically a serial-to-parallel converter: serial data sent from the keyboard is parallelized and made available to the RAT CPU. Figure 54 shows a high-level diagram for the overall system using a keyboard. Once again, a different document provides more pertinent keyboard driver details.

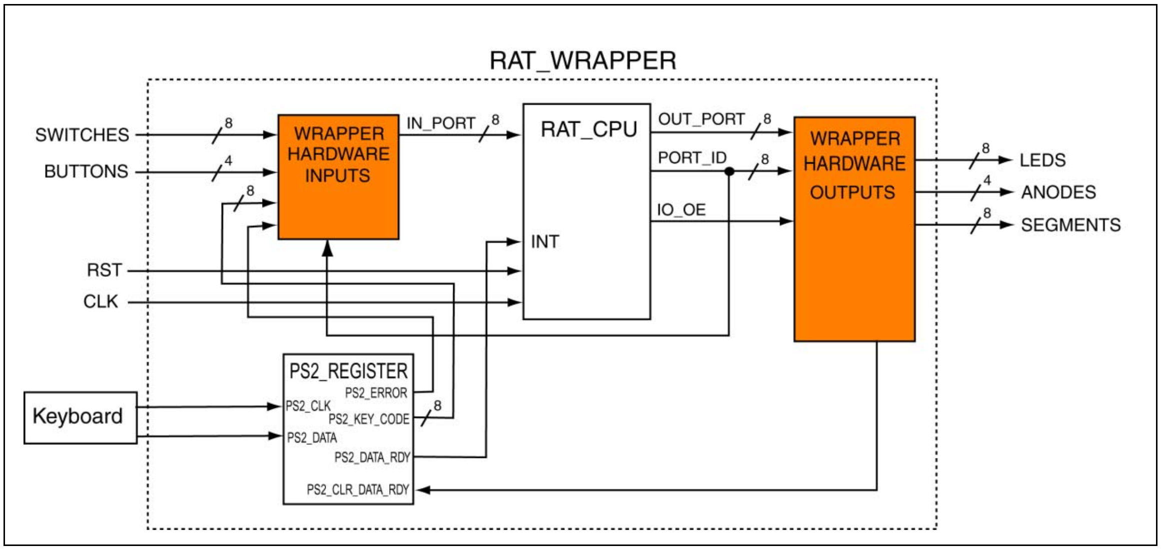


Figure 68: Black box of the RAT Wrapper showing the interface of the PS2_REGISTER.

The Etch-a-Sketch Program Details:

The Etch-a-Sketch program is relatively simple. The program is interrupt driven; the main responsibilities of the program are to wait for input from the keyboard, and then react to that input in a way that imitates the basic operation of an Etch-a-Sketch toy. The program also provides an added feature of using the eight switches on the Nexys2 board to interactively change the background color on the display. This feature shows how you can poll switch values in a relatively useful way as well as providing you with a way to select pixel colors for your particular CPE 233 project without having to "guess" how a color will appear based on the 8-bit color data associated with the CPE 233 VGA driver.

The final notion to keep in mind regarding the Etch-a-Sketch program is that it borrows heavily from the original draw_dot program and the PS/2 driver module. Other available documents describe the basic aspects of that particular functionality so we won't repeat it here. The following verbage describes the new and important aspects of the Etch-a-Sketch program, so we don't describe all of the code in the Etch-a-Sketch program as much of the code appears in the draw_dot and PS/2 driver programs.

Figure 55 shows the main code associated with the driver program. This code consists of both an initialization sequence followed by the main loop of the code. The initialization portion of the code includes initializing registers for both the display and keyboard control aspects of the program. The program initializes registers r7 & r8 to act as ongoing holders of the current dot location. Similarly, the keyboard driver requires the clearing of the r15 register to indicate no keyboard keys have been pressed. The other portion of the init sequence establishes an initial value for the switches, which the program uses to draw the background color.

The main loop of the foreground task comprises of polling the development board's switches. This portion of the code is constantly comparing the current switch value to the previous switch value in an effort to determine if the user has changed the switch values. When the code detects a change, several thing occur. First, the program redraws the display background with the color associated with the new value on the switches. Second, the program changes the default color of the line drawing color. And finally, the drawing portion of the program restarts with a dot of the new color in the middle of the display that contains the new background color.

```
;-----------------------------------------------------------------
; Foreground Task
;-----------------------------------------------------------------
init:
         MOV    r7, 0x0F         ;- starting y value (middle row)
         MOV    r8, 0x14         ;- starting x value (middle column)
         MOV    r15,0x00         ;- clear interrupt flag register

         MOV    r6, BG_COLOR     ;- bluish color
         CALL   draw_background  ;- draw using default color

         MOV    r6, BLUE         ;- bluish color
         CALL   draw_dot         ;- plop down intial shape
         IN     r20,SWITCHES     ;- store current switch settings
         SEI                     ;- allow interrupts


main:    IN     r21,SWITCHES     ;- poll the switches
         CMP    r20,r21          ;- see if a new value is requested
         BREQ   main             ;- keep polling if no new value

         MOV    r20,r21          ;- save new color
         MOV    r6,r21           ;- set new color as dot color
         CALL   draw_background

         EXOR   r6,0xFF          ;- copy new color to dot color
         MOV    r7, 0x0F         ;- restore starting y value
         MOV    r8, 0x14         ;- restore starting x value
         CALL   draw_dot         ;- draw original dot
         BRN    main             ;- dumb poll waiting for interrupts
;-------------------------------------------------------------------
```

Figure 69: The initialization and main code for the keyboard driver program.

Figure 48 shows the interrupt service routine for the Etch-a-Sketch driver program. This code does two main things. First, it looks for the key presses of interest, which is either an up, down, left, or right direction associated with constants defined in as pre-assembler directives which are described in a portion of the driver program not described in this document. The ISR either reacts to a valid key press, or ignores any other key press. Each valid key press first determines if that particular key press will move the drawing line off the associated display. When the program detects a valid key press, a the program moves the adjusts the current drawing location and draws a dot at that location using the default dot drawing color stored in register r6.

Lastly, the bottom portion of the ISR code is responsible for basic ISR administrative tasks are they relate to the keyboard driver. The code at the bottom of the ISR sends out a pulse to the PS/2 module, which instructs the PS/2 register to remove the asserted interrupt from the RAT CPU and allows the keyboard to send more data to the RAT CPU.

```
;--------------------------------------------------------------
; Interrupt Service Routine - Handles Interrupts from keyboard
;--------------------------------------------------------------
; Sample ISR that looks for various key presses. When a useful
; key press is found, the program does something useful. The
; code also handles the key-up code and subsequent re-sending
; of the associated scan-code.
;
; Tweaked Registers; r2,r3,r15
;--------------------------------------------------------------
ISR:      CMP   r15, int_flag        ; check key-up flag
          BRNE  continue
          MOV   r15, 0x00            ; clean key-up flag
          BRN   reset_ps2_register

continue: IN    r2, PS2_KEY_CODE     ; get keycode data

move_up:  CMP   r2, UP               ; decode keypress value
          BRNE  move_down
          CALL  sub_y                ; verify move is possible
          CALL  draw_dot             ; draw object
          BRN   reset_ps2_register

move_down:
          CMP   r2, DOWN
          BRNE  move_left
          CALL  add_y                ; verify move
          CALL  draw_dot             ; draw object
          BRN   reset_ps2_register

move_left:
          CMP   r2, LEFT
          BRNE  move_right
          CALL  sub_x                ; verify move
          CALL  draw_dot             ; draw object
          BRN   reset_ps2_register

move_right:
          CMP   r2, RIGHT
          BRNE  key_up_check
          CALL  add_x                ; verify move
          CALL  draw_dot             ; draw object
          BRN   reset_ps2_register


key_up_check:
          CMP   r2,KEY_UP            ; look for key-up code
          BRNE  reset_ps2_register   ; branch if not found

set_skip_flag:
          ADD   r15, 0x01            ; indicate key-up found


reset_ps2_register:                  ; reset PS2 register
          MOV    r3, 0x01
          OUT    r3, PS2_CONTROL
          MOV    r3, 0x00
          OUT    r3, PS2_CONTROL
          RETIE
;-------------------------------------------------------------------
```

Figure 70: The interrupt service routine.

Figure 66 shows four subroutines which prevent the user from driving the line drawn by the Etch-a-Sketch program off the side of the display. The basic notion of these four subroutines is to check the given bounds, which in this case are the edges of the display, if the user requests that the blit off the side of the display, the subroutines effectively ignored the request, and control returns to the calling program.

```
;-----------------------------------------------------------
;- These subroutines add and/or subtract '1' from the given
;- X or Y value, depending on the direction the blit was
;- told to go. The trick here is to not go off the screen
;- so the blit is moved only if there is room to move the
;- blit without going off the screen.
;-
;- Tweaked Registers: possibly r7; possibly r8
;-----------------------------------------------------------
sub_x:   CMP   r8,LO_X    ; see if you can move
         BREQ  done1
         SUB   r8,0x01    ; move if you can
done1:   RET

sub_y:   CMP   r7,LO_Y    ; see if you can move
         BREQ  done2
         SUB   r7,0x01    ; move if you can
done2:   RET

add_x:   CMP   r8,HI_X    ; see if you can move
         BREQ  done3
         ADD   r8,0x01    ; move if you can
done3:   RET

add_y:   CMP   r7,HI_Y    ; see if you can move
         BREQ  done4
         ADD   r7,0x01    ; move if you can
done4:   RET
;--------------------------------------------------------
```

Figure 71: Two generic subroutines associated the pressing of a “key of interest”.

COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

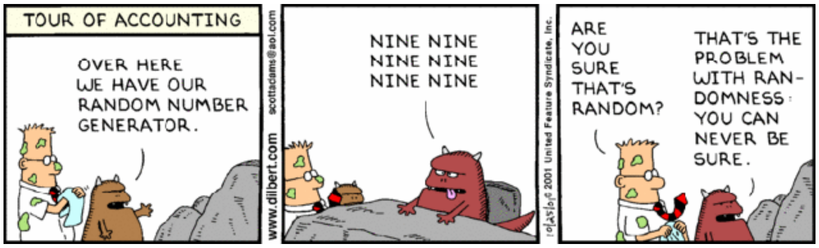
CPE 233 External RAT Module

The Pseudo-Random Number Generator Module

Objectives:

- To provide a basic operational over view of the Pseudo-Random Number Generator

Somewhat Meaningful Comments:


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com
scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

Jokes aside, one of the holy grails in technical land is the random number. Though there are many ways to generate a random number, pretty much none of them can generate a true random number. That is why most people never say they have a “random number generator”, they say they have a “pseudo-random number generator”. The ability to generate true random numbers in digital-land is important in many fields, particularly cryptography.

This document describes a pseudo-random number generator (PRNG) that you can use in your CPE 233 projects. This PRNG is synthesized from a VHDL model, which you can find among the online course materials. The current VHDL model for the PRNG is based on a well-known digital construct referred to as a linear feedback shift register (LFSR). The LFSR is quite useful for many things in digital-land, including cyclic redundancy checkers (CRCs). The basic LFSR looks like a standard shift register, but some of the internal bits are EXORed together in an effort to create a random affect.

Figure 72 shows the black box diagram for the PRNG. Keep in mind that this device is truly a register; in particular, this PRNG is an 8-bit register. Figure 73 shows the underlying registers both before and after the clock edge. Figure 73 does not show the gate-level logic in the PRNG, but it does indicate the resulting equations. There are two noteworthy items in Figure 73. The new value in the register is formed by 1) creative shifting, 2) an exclusive ORing of the two MSBs of the registers pre-clock edge value, and a complementing of bit(2) from the pre-clock edge register value.

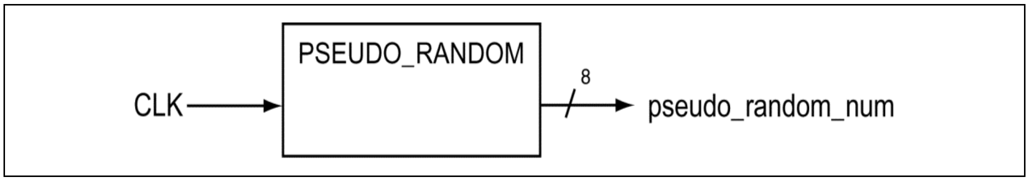


Figure 72: The black box diagram for the PRNG.

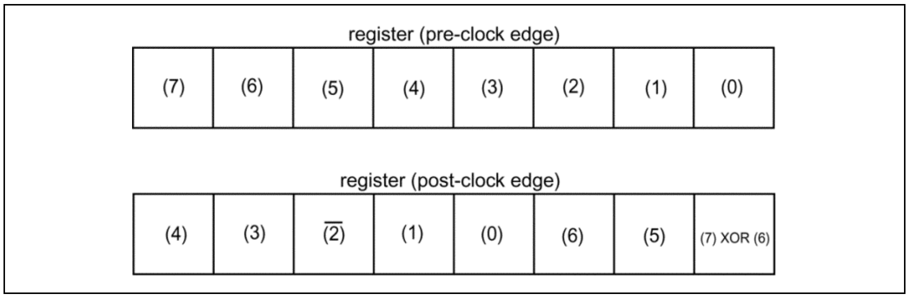


Figure 73: The PRNG register before and after the clock edge.

Using the Pseudo-Random Number Generator Module

There are two things you need to do in order to user the PRNG in your CPE 233 project. First, you must include the output of the register into the input section of the current RAT Wrapper. You need to assign a port ID for the PRNG, and then use that port ID when you access the PRNG under RAT assembly language program control. Recall that the port ID you assign the PRNG controls the MUX whose output connects to the single 8-bit data input of the RAT CPU. Once you modified the hardware, you access the PRNG with a RAT IN instruction using the port ID you assigned in the RAT Wrapper.

The other issue involved with using the PRNG in a RAT assembly language program is the notion that you typically need to tweak the pseudo-random number in order to make it usable for your needs. Here are the typical usage scenarios:

- If your program in fact needs an 8-bit random number than, you don’t need to do anything; you can simply use the number as input from the PRNG.

- If your program needs random number with a range that fall within the bounds of binary numbers, than you’ll need to clear the unused bits in the 8-bit number provided by the PRNG. For example, if you need a number in the range [0,63], you would clear the two MSBs of the 8-bit number output from the PRNG to provide you with a random number in that range.

- If you have a special range, something that does not follow the above two bullets, then you have to either do special things or compromise the randomness of your number. Here are a few strategies based on the notion that you’re hoping for a PWRN in the range [0,39].

    1. Input a PRN; clear the top two bits. If the resulting number is greater than 39, toss out the number and restart this process.

2. Input a PRN; clear the most significant three bits to put the number in the range [0,31]. Input another PRN, clear the most significant five bits to put the number in the range [0,7]. Add these two numbers. This provides a range of [0,38].

COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

CPE 233 External RAT Module

Timer-Counter Module Description

Objectives:

- To describe the functionality and use of a timer-counter module

Somewhat Meaningful Comments:

Similar to most microcontrollers, the RAT CPU has a simple interrupt architecture that allows the user to run both a foreground and background task. However, unlike most microcontrollers, the RAT CPU lacks many basic internal peripheral devices that allow the CPU to be more functional when working with a real application. These internal peripheral devices allow more flexibility in "doing things" with the microcontroller by allowing "tasks" to occur in the background and thus freeing up processor resources by allows the program to concentrate on required foreground task processing.

The timer-counters are probably the most common internal peripheral in microcontrollers. These devices are relatively simple but provide useful functionality. This document describes a timer-counter that is as basic as possible in order to facilitate your understanding and use of the device. The timer-counter is implemented as a VHDL model; the timer-counter itself is instantiated at the RAT Wrapper level. The timer-counter is an external peripheral device that you can use in conjunction with the RAT CPU.

High-level Timer-Counter Description

Figure 24 shows the black-box diagram for the timer-counter module. The timer-counter is an up counter automatically counts up when the device is enabled. The user can preset a "terminal count" for the timer-counter; the timer-counter reaches that terminal counter, the timer-counter outputs a two-period-wide positive output pulse that you can as an input to the RAT's interrupt input. In this way, the timer-counter can automatically generate interrupts in at a frequency specified by the programmer by writing to the TCCNT registers, which hold the terminal counts. The timer counter also has a clock-prescaling feature to allow programmers to further reduce the frequency of generated interrupts.

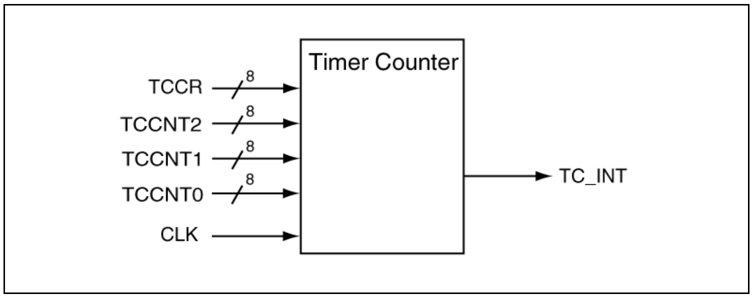
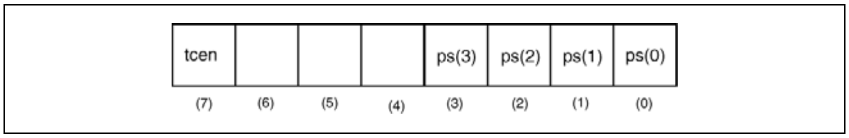


Figure 74: Black box diagram for the timer-counter peripheral.

Low-level Timer-Counter Description

The timer-counter module consists to two main registers: the timer-counter control register (TCCR) and the count register; the count register is a 24-bit register implemented as three 8-bit registers. Figure 25 shows the format of the TCCR.



tcen				ps(3)	ps(2)	ps(1)	ps(0)
(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)

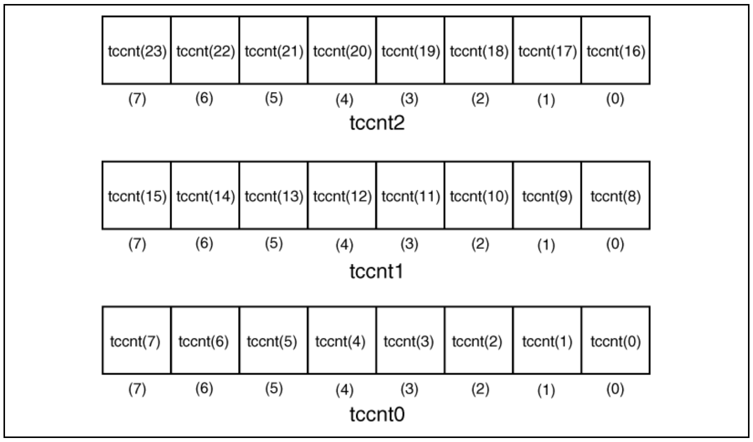
Figure 75: Timer-counter control register TCCR.

Signal	Description
tcen	This signal enables or disables the timer-counter. Writing to this bit turns on (‘1’) or turns off (‘0’) the timer-counter module. When the timer-counter is turned off, the count associated with the timer-counter resets to zero.
ps(3:0)	This four-bit signal represents a number that is used to “prescale” the clock input to the timer-counter. In this way, the programmer can effectively reduce the input clock frequency before the signal is “counted” by the timer-counter. Scaling the inputs clock signal allows for longer count periods beyond the value held in the timer-counter terminal counter registers. These pre-scale bits can be written at any time in a count sequence and will take affect when the timer-counter reaches the current prescale value.

Table 11: Timer-Counter Control Register (TCCR) description.

Figure 51 shows the timer-counter count registers (TCCNTx) comprised of three 8-bit registers that hold the terminal count for the timer counter. The timer-counter operates by counting upwards starting from zero; when the count in the timer-counter is equivalent to the count in the TCCNTx registers, the timer-counter outputs a pulse. Note that the duration of the output pulse is two system clock periods, and not two pre-scaled clock periods. When the timer counter reaches the terminal count, the timer-counter count resets to zero and continues counting upwards. Figure 78 shows an example driver program for this module.

Figure 77 provides a timing diagram that highlights the important operational characteristics of the timer-counter module. In Figure 77, the terminal count of the timer was pre-set to 0x400000 and no prescaler is applied (no applied prescaler means the prescaler was set to “0000”). Two things occur when the timer-counter reaches the terminal count. First, the counter is automatically reset to 0x000000, and second, the timer-counter module outputs a two clock cycle pulse. The clock signal to the timer-counter module is the system clock. Thus, the two clock cycle pulse is intended to connect to the RAT’s interrupt input, where two clock cycles guarantees the pulse will be seen by the RAT CPU.



(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)
tccnt(23)	tccnt(22)	tccnt(21)	tccnt(20)	tccnt(19)	tccnt(18)	tccnt(17)	tccnt(16)

tccnt2

(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)
tccnt(15)	tccnt(14)	tccnt(13)	tccnt(12)	tccnt(11)	tccnt(10)	tccnt(9)	tccnt(8)

tccnt1

(7)	(6)	(5)	(4)	(3)	(2)	(1)	(0)
tccnt(7)	tccnt(6)	tccnt(5)	tccnt(4)	tccnt(3)	tccnt(2)	tccnt(1)	tccnt(0)

tccnt0

Figure 51: Timer-counter 24-bit count registers: TCCNT2, TCCNT1, TCCNT0.

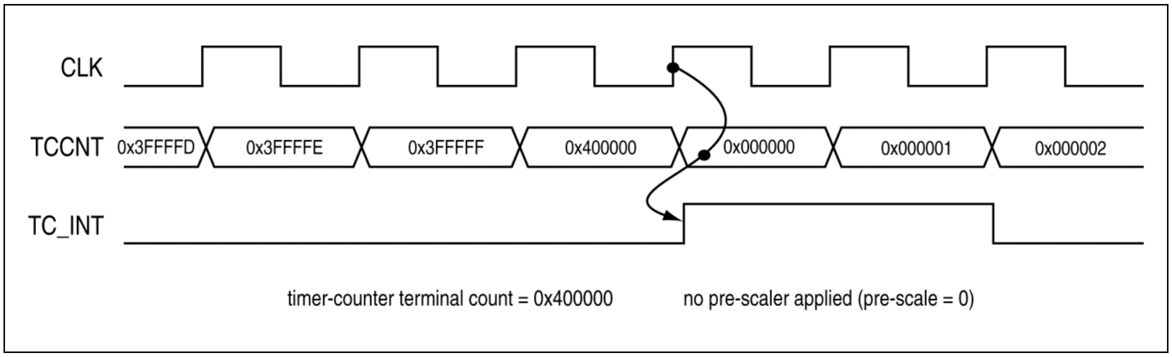

CLK
TCCNT
TC_INT
0x3FFFFD
0x3FFFFE
0x3FFFFF
0x400000
0x000000
0x000001
0x000002
timer-counter terminal count = 0x400000
no pre-scaler applied (pre-scale = 0)

Figure 77: Timing diagram showing the inner workings of the timer-counter module.

```
; --------------------------------------------------------------------
;- Engineer: James Ratner
;- Company: Barefoot Engineering
;-
;- The program tests a background timer counter modules that can be
;- used to provide background t(hardware) timing for RAT applications.
;- This program blinks the left-most led; as a sanity check, the
;- the first four switches are used to turn on the right-four LEDs.
;--------------------------------------------------------------------
;--------------------------------------------------------------------
;-- Assembler Directives
;- --------------------------------------------------------------------
.EQU   SWITCHES   = 0x20      ; in: switches address

.EQU   SSEG_DISP  = 0x82      ; out: 7-segment display address
.EQU   SSEG_EN    = 0x83      ; out: display enable address
.EQU   LEDS       = 0x40      ; out: LED address
                              ;
.EQU   TCCR       = 0xB5      ; timer-Counter control register
.EQU   TCCNT0     = 0xB0      ; timer-Counter count reg (7:0)
.EQU   TCCNT1     = 0xB1      ; timer-Counter count reg (15:8)
.EQU   TCCNT2     = 0xB2      ; timer-Counter count reg (23:0)
;--------------------------------------------------------------------
.CSEG
.ORG    0x10

init1:      MOV    r0, 0xFF          ; turn off sseg display
            OUT    r0, SSEG_EN       ;

            OUT    r0, TCCNT0        ; set timer counter to max value
            OUT    r0, TCCNT1
            OUT    r0, TCCNT2

            MOV    r0, 0x81          ; turn timer on with max pre-scale
            OUT    r0, TCCR          ;

            MOV    r31, 0x80         ; init LED Blink
            OUT    r31, LEDS
            SEI                      ; enable interrupts


;--------------------------------------------------------------------------
; Relatively meaningless foreground task; this task allows you to use the
;  switches to verify that the program is actually working
;--------------------------------------------------------------------------
main:       IN     r4, SWITCHES      ; input switch values
            AND    r4, 0x0F          ; clear the upper four bits
            OR     r4, r31           ; OR with blinking bit
            OUT    r4, LEDS          ; write to LEDs
            BRN    main              ; continue with foreground task


;--------------------------------------------------------------------------
;- ISR: toggles bit(7) of r31
;--------------------------------------------------------------------------
isr:        EXOR    r31, 0x80        ; toggle LED output
            RETIE
;--------------------------------------------------------------------------

.CSEG
.ORG   0x3FF
            BRN  isr                 ; My baby does the ISR thang
```

Figure 78: Source code listing for timer_test_prog.asm .

Troubleshooter’s Guide to Xilinx

Xilinx Golden Rules

- Make sure there are no spaces in the path name to your project. If there are spaces, any one of a number of problems may arise that will prevent you from completing your project.

- Make sure you do not save project in the folder that contains the Xilinx software. You should always create a special folder for yourself away from system software areas.

- If the Xilinx application is acting strange of running slow, don’t deal with it: reboot your computer.

Known Quirks and Common Errors

- Entity and architecture names must not start with a numeric character (VHDL)

- Entity and architecture names must not contain hyphens (VHDL)

- Entity and architecture names much not contain spaces (VHDL)

- All dealings with Xilinx should be done on the local drive. Completed projects should be saved to your own personal storage device such as a USB drive. Leaving files on a lab computer is extremely problematic if you ever hope to see your work again. In addition, working from local drive increases the speed at which you can traverse the Xilinx Design Methodology.

- Working off a USB drive is doable without problems but it is much slower than working from the host computer’s hard drive.

Xilinx Catastrophic Error Fix Flow

If you’ve worked with Xilinx software before, you know that sometimes your project simply will not compile even though everything is in order on your project. In these cases, you may want to attempt one of the following set of steps (in the order listed):

- 1) Save all the files in your current project, 2) kill then relaunch the Xilinx application, 3) create a new project, 4) Recompile your project.

- 1) Save all the files in your current project, 2) turn off your computer, 3) restart your computer, 4) relaunch the Xilinx application, 3) create a new project, 4) Recompile your project.

Constraints File (.xdc) Quirks and Problems

Problem	Possible Causes/Solutions
You get strange error messages	Verify the case of the signals match in the higher-level entity and your .xdc file. You have a type in the .xdc file

General Synthesis Problems

Problem	Possible Causes/Solutions
I made a bunch of changes to my circuit and reprogrammed the FPGA but they changes do not seem to be implemented in the final circuit.	Verify that you downloaded the correct programming file to the board. It is easy to have your project in one directory and have software use a configuration file from another directory.
The Xilinx software is running really slow, that is, much slower than usual.	Do all circuit development on the local drive. If you work from your USB drive, data transfers can be painfully slow. If all else fails, exit the Xilinx ISE software and re-launch. If that fails, restart the computer before you re-launch the software.
The error message points to a line in the code that cannot possible have an error.	You are most likely viewing the wrong file. Verify that you are viewing the correct file by comparing the file specified by the error message to the file in the active window.
General strange stuff is happening, such that you made a change that does not seem to take effect.	Vivado handles files very strangely. If often has a link to an external file that you thinking you're modifying locally.

Lab Grading Acronyms

√	**Check Mark** – This means that minimum expectations were met.
ANN	**Annotations** – This means that a timing diagram was missing or did not have proper annotations.
BE	**Better English**: Indicates that the right information is present, but you should strive to write better English.
BV	**Bad Signal/Variable Labels**: Be sure to use self-commenting labels for all signal and variable names. This includes both VHDL and assembly language programs.
CAP	**Caption** – A figure diagram or table did not have a proper caption or title. Each figure should have a brief stand-alone description.
G	**Good** – This indicates something was done better than minimum requirements.
IND	**Indentation Problems**: All source code (VHDL, assembly language) should be properly indented.
MC	**Missing Comments** – All source code (VHDL, assembly language) should be well commented. This comment means comments are either missing or have some other problem.
MH	**Missing Header** – All source code (VHDL, assembly language) should have a file header with information describing what is found in the file. This comment indicates that the header is missing or does not contain the proper information.
MP	**Missing Point** – Generally associated with conclusion but also could be associated with objectives. This comment generally means that the conclusion did not state if experiment's objectives were met or the conclusion simply missed the point of the experiment.
MQ	**Museum Quality** – When source code goes beyond all the rules of neatness and clarity, you get assigned this acronym. You know you've done a good job.
MS	**More Space** – Don't try to cram stuff in. Use more space between sections, figures, etc. to give the report an overall neat appearance. This also includes the insertion of white space in VHDL and assembly language programs.
MW	**Missing Work** – There was a calculation or derivation that you should have shown more explicitly in your report. All non-trivial calculations and derivations should be included somewhere in your report.
NC	**Not Clear** – This can refer to anything but most often refers to descriptions of things or answers to questions.
NR	**Not Readable** – This can refer to anything but most often refers to hand-written stuff (try to be neat) or cut-and-pasted diagrams from other sources.
RG	**Re-read Lab Report Guidelines** – If your report has too many problems, your best bet will be to go back and read the lab report guidelines.
REF	**Missing References** – All diagrams/figures in report should be referenced from the body of the report.
SIG	**Signal Consistency** – Signal names on all models (timing diagrams, VHDL models, circuit diagrams) should match.
TT	**Title** – Experiment has problems with title or information in title such as course, section number, names of people in group, etc.
TSH	**Too Short** – This can apply to anything including conclusions, descriptions, objectives, testing procedures, etc.
TSM	**Too Small** – This can refer to anything. Make everything, particularly diagrams, large enough to be read by the average human.
WC	**Weak Conclusion** – This refers to the overall quality of the conclusion. The conclusion can be weak for many reasons; refer to the Lab Report document for an overview of what is expected in the conclusion.

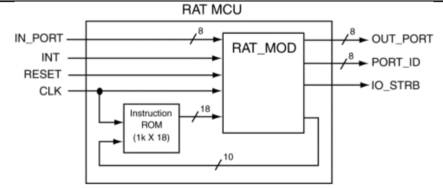
VHDL Style File

The main goal of your VHDL source code is to model a digital circuit. This means that your only mission is to satisfy the VHDL synthesizer. But in reality, you and other humans are going to need to read and understand your VHDL source code. The single most important factor in creating readable and understandable VHDL source code is to make sure your VHDL source code follows certain appearance guidelines. The file listed below shows some more of the more important attributes and rules that all VHDL source code should adhere to. The overriding factor with your VHDL source code is to make it neat, organized, and readable; any specific items not listed in this style files should adhere to these principles.

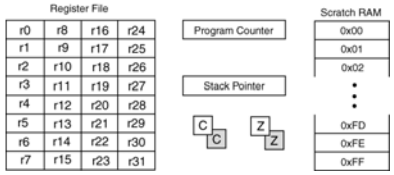
```
----------------------------------------------------------------------------------
-- Company:
-- Engineer: Manuel Laybor, Ima Goodstud
--
-- Create Date: 09/10/2007
-- Design Name: VHDL Style File
-- Module Name: comp2
-- Project Name: style file example
-- Target Devices: Digilent Nexys Board
-- Tool versions:
-- Description: Describe the purpose of this file. This should be a high-level
--     description but it should be complete. The low-level implementation
--     details should appear in the body of the VHDL code in the form of comments.
--
-- Dependencies: If the synthesis of this file depends on other file, be
--     sure to list them here.
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments: Say stuff that may be helpful to others who may be
--     wanting to use this file for their own projects.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- delete meaningless comments (such as this one)

-- entity declaration should appear neat; much of the declaration
--  is aligned to increase readability.
entity comp2 is
    Port ( A,B : in  STD_LOGIC_VECTOR (9 downto 0);
            EQ : out STD_LOGIC);
end comp2;

-- code is indented;
--  comments align with code indentations
architecture my_comp2 of comp2 is
begin
   -- all items in architecture body are indented

   -- label is included with process statements
   my_comp: process(A,B)
   begin
      if (A = B) then
         EQ <= '1';
      else
         EQ <= '0';
      end if;
   end process my_comp;

end my_comp2;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity comp3
    Port ( A,B,C : in  STD_LOGIC_VECTOR (9 downto 0);
             EQ3 : out STD_LOGIC);
end comp3;

architecture my_comp3 of comp3 is
   -- all items in declarative region of architecture are indented

   -- component declaration
   component comp2
       Port ( A,B : in  STD_LOGIC_VECTOR (9 downto 0);
               EQ : out STD_LOGIC);
   end component;

   -- intermediate signals: note proper prefixing
   signal s_eq1, s_eq2 : std_logic;

begin
   -- blank lines are used to delineate instantiations
   --  and signal assignment statements

   -- component instantiations:
   --  label and entity name one first line
   --  port mappings on subsequent lines
   --
   --  port mappings are neat and aligned
   eq1: comp2
   port map (A => A,
             B => B,
             EQ => s_eq1);

   -- component instantiation
   --  another form of alignment used
   eq2: comp2
   port map ( A => A,
              B => C,
             EQ => s_eq2);

   -- basic logic for final output
   EQ3 <= s_eq1 AND s_eq2;

end;
```

RAT MCU Architecture and Assembly Language Cheat Sheet



schematic diagram



programming model

RAT Instruction Set:

Program Control			Interrupt	Input/Output
`BREQ label`	`BRN label`	`CLC`	`RETID`	`IN rX,pp`
`BRNE label`		`SEC`	`RETIE`	`OUT rX,pp`
`BRCS label`	`CALL label`		`SEI`	
`BRCC label`	`RET`	`WSP rX`	`CLI`	

Logical		Arithmetic		Shift & Rotate	Storage	
`AND rX,kk`	`AND rX,rY`	`ADD rX,kk`	`ADD rX,rY`	`LSL rX`	`ST rX,imm`	`PUSH rX`
`OR rX,kk`	`OR rX,rY`	`ADDC rX,kk`	`ADDC rX,rY`	`LSR rX`	`ST rX,(rY)`	`POP rX`
`EXOR rX,kk`	`EXOR rX,rY`	`SUB rX,kk`	`SUB rX,rY`	`ROL rX`	`LD rX,imm`	
`TEST rX,kk`	`TEST rX,rY`	`SUBC rX,kk`	`SUBC rX,rY`	`ROR rX`	`LD rX,(rY)`	`MOV rX,rY`
		`CMP rX,kk`	`CMP rX,rY`	`ASR rX`		`MOV rX,imm`

Fun Facts:

- Max program size: 1024 instructions (18-bit/instr)
- 32 8-bit general purpose registers (GPRs)
- Stack: implemented in scratch RAM

I/O:

- Port Mapped device (8-bit port addresses)
- 8-bit GPR-based Input and Output

Interrupt Architecture:

- Interrupt on: interrupt input high voltage & interrupt enabled
- One maskable external interrupt (see interrupt group)
- Vector interrupt: vector address 0x3FF
- Context saving: C & Z flags saved on interrupt
- Interrupt automatically disabled on interrupt
- Context restoration: C & Z flags restored on RETID or RETIE instructions

Bit Masking:

bit setting: OR with ‘1’	bit clearing: AND with ‘0’	bit toggling: XOR with ‘1’
`OR r3,0x01 ;set bit 0`	`AND r3,0xFE ;clear bit 0:`	`EXOR r3,0x01 ;toggle bit 0`

Conditional Constructs:

if/else construct

input byte; if byte is 0x00, then r1=0x0A; otherwise, r1=0x0B output r1

```
              IN       r0,IN_PORT     ; grab data
              CMP      r0,0x00        ; test to see if byte is 0x00
              BRNE     not_zero       ; jump if r0 is not 0x00
              MOV      r1,0x0A        ; place 0x0A in r1
              BRN      out_val        ; jump to output instruction
                                      ;
not_zero:     MOV      r1,0x0B        ; place 0x0B in r1
out_val:      OUT      r1,OUT_PORT    ; output some data
```

iterative construct
(known iterative value)

place iterative value in r3; do something; decrement count; check to see if zero; repeat if not zero

```
              MOV      r3,0x08        ; load iterative count value
loop:         ;                       ; do something meaningful
              ;
              SUB      r3,0x01        ; decrement iteration variable
              BRNE     loop           ; do it again if count non-zero
              ;
loop_done:    ; do something else…
```

conditional construct
(unknown iterative value)

Input value; increment it; do it again if carry flag is not set

```
              MOV      r0,0x00        ; clear register
loop:         IN       r1,IN_PORT     ; grab some data
              ADD      r0,r1          ; add some value to r0
              BRCC     loop           ; repeat if no carry

loop_done:    ; do something else…
```

Equality/Inequality:

- C and Z flag are used to establish relationship between registers:

Operation:	`SUB rA, rB`
	`CMP rA, rB`	

C	**Z**	**Comment**
0	0	rA > rB
1	0	rA < rB
-	1	rA = rB

RAT MCU Assembly Language Style File

The following file shows some of the more important issues regarding generating neat and readable RAT MCU assembly source code. No style file can show you everything and they rarely make such an attempt. The underlying factor in writing any source code is to be neat and consistent. Using proper indentation, white space and commenting helps you attain the goals of being neat and consistent. The code in the following block is a modified fragment from another working program; if it were to actually assemble, it would not do anything useful. We present this program primarily for appearance purposes.

```
;--------------------------------------------------------------------
;- Programmer: Pat Wankaholic
;- Date: 09-29-10
;- Experiment #??
;-
;- This program does something really cool. Here's the description...
;--------------------------------------------------------------------

;--------------------------------------------------------------------
;- I/O Constants
;--------------------------------------------------------------------
.EQU SWITCH_PORT = 0x30          ; port for switches ----- INPUT
.EQU LED_PORT    = 0x0C          ; port for LED output --- OUTOUT
.EQU BTN_PORT    = 0x10          ; port for button input - INPUT
;--------------------------------------------------------------------

;--------------------------------------------------------------------
;- Misc Constants
;--------------------------------------------------------------------
.EQU BTN2_MASK = 0x08            ; mask all but BTN5
.EQU B0_MASK   = 0x01            ; mask all but bit0
;--------------------------------------------------------------------

;--------------------------------------------------------------------
;- Memory Designation Constants
;--------------------------------------------------------------------
.DSEG
.ORG     0x00

COW:   .DB 0x09,0x07,0x06,0x05
;--------------------------------------------------------------------
;
.CSEG
.ORG         0x01

init:        SEI                       ; enable interrupts

main_loop:   IN      r0, BTN_PORT      ; input status of buttons
             AND     r0, BTN2_MASK     ; clear all but BTN2
             BRN     bit_wank          ; jumps when BTN2 is pressed

             ;----------------------------------------------------------
             ;- nibble wank portion of code
             ;----------------------------------------------------------
wank:        ROL     r1                ; rotate 2 times - msb-->lsb
bit3:        BRN     fin_out           ; jump unconditionally to led output
             ;----------------------------------------------------------

             ;----------------------------------------------------------
             ; bit-wank algo: do something Blah, blah, blah ...
             ;----------------------------------------------------------
```

```
bit_wank:    LD      r0,0x00           ; clear r0 for working register

             OR      r0, B1_MASK       ; set bit1
bit2:        LSR     r1                ; shift msb into carry bit
             BRCS    bit3              ; jump if carry not set
             ;-------------------------------------------------------------

             CALL    My_sub           ; subroutine call
fin_out:     OUT     r0,LED_PORT      ; output data to LEDs
             BRN     main_loop        ; endless loop
             ;-------------------------------------------------------------


             ;-------------------------------------------------------------
             ; Subroutine: My_sub:
             ;
             ; This routines does something useful. It expects to find
             ; some special data in registers r0, r1, and r2. It changes
             ; the contents of registers blah, blah, blah...
             ;
             ; Tweaked registers: r1
             ;--------------------------------------------------------------
My_sub:      LSR     r1               ; shift msb into carry bit
             BRCS    bit3             ; jump if carry not set
             RET
```

VHDL Cheat-Sheet

Concurrent Statements		Sequential Statements
Concurrent Signal Assignment **(dataflow model)**	⇔	**Signal Assignment**
target <= *expression*;		*target* <= *expression*;

A <= B **AND** C;
DAT <= (D **AND** E) **OR** (F **AND** G);
		A <= B **AND** C;
		DAT <= (D **AND** E) OR (F **AND** G);

Conditional Signal Assignment **(dataflow model)**	⇔	***if*** **statements**

target <= *expressn* **when** *condition* **else**
          expressn **when** *condition* **else**
          expressn;
		if (*condition*) **then**
		   { sequence of statements }
		elsif (*condition*) **then**
		   { sequence of statements }
		else --(the else is optional)
		   { sequence of statements }
		end if;

F3 <= ‘1’ **when** (L=‘0’ **AND** M=‘0’) **else**
      ‘1’ **when** (L=‘1’ **AND** M=‘1’) **else**
      ‘0’;
		if (SEL = “111”) **then** F_CTRL <= D(7);
		elsif (SEL = “110”) **then** F_CTRL <= D(6);
		elsif (SEL = “101”) **then** F_CTRL <= D(1);
		elsif (SEL = “000”) **then** F_CTRL <= D(0);
		else F_CTRL <= ‘0’;
		end if;

Selective Signal Assignment **(dataflow model)**	⇔	***case*** **statements**

with *chooser_expression* **select**
   target <= *expression* **when** *choices*,
             expression **when** *choices;*
		case (*expression*) **is**
		   when *choices* =>
		       {sequential statements}
		   when *choices* =>
		       {sequential statements}
		   when others => -- (optional)
		       {sequential statements}
		end case;

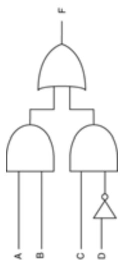
with SEL **select**
MX_OUT <= D3 **when** “11”,
          D2 **when** “10”,
          D1 **when** “01”,
          D0 **when** “00”,
          ‘0’ **when others**;
		case ABC **is**
		    when “100” **=>** F_OUT **<=** ‘1’;
		    when “011” **=>** F_OUT **<=** ‘1’;
		    when “111” **=>** F_OUT **<=** ‘1’;
		    when others => F_OUT **<=** ‘0’;
		end case;

Process **(behavioral model)**		

opt_label: **process**(*sensitivity_list*)
begin
   {*sequential_statements*}
end process *opt_label*;		

proc1: **process**(A,B,C)
begin
   if (A = ‘1’ **and** B = ‘0’**) then**
      F_OUT **<=** ‘1’;
   elsif (B = ‘1’ **and** C = ‘1’) **then**
      F_OUT **<=** ‘1’;
   else
      F_OUT **<=** ‘0’;
   end if;
end process proc1;		

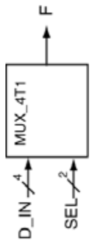
<table>
<tr><th>Description</th><th>CKT Diagram</th><th>VHDL Model</th></tr>
<tr><td>Typical logic circuit</td><td>



</td><td>

```
entity my_ckt is
   Port ( A,B,C,D : in std_logic;
                F : out std_logic);
end my_ckt;

architecture ckt1 of my_ckt is
begin
   F <= (A AND B) OR (C AND (NOT D));
end ckt1;

architecture ckt2 of my_ckt is
begin
   F <= ‘1’ when (A = ‘1’ AND B = ‘1’) else
        ‘1’ when (C = ‘1’ AND D = ‘0’) else
        ‘0’;
end ckt2;
```

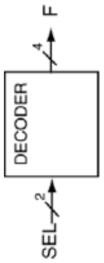
</td></tr>
<tr><td>4:1 Multiplexor</td><td>



</td><td>

```
entity MUX_4T1 is
   Port (  SEL : in std_logic_vector(1 downto 0);
          D_IN : in std_logic_vector(3 downto 0);
             F : out std_logic);
end MUX_4T1;

architecture my_mux of MUX_4T1 is
begin
   F <= D_IN(0) when (SEL = "00") else
        D_IN(1) when (SEL = "01") else
        D_IN(2) when (SEL = "10") else
        D_IN(3) when (SEL = "11") else
        '0';
end my_mux;
```

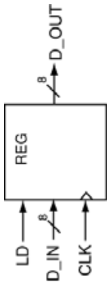
</td></tr>
<tr><td>2:4 Decoder</td><td>



</td><td>

```
entity DECODER is
   Port ( SEL : in std_logic_vector(1 downto 0);
            F : out std_logic_vector(3 downto 0));
end DECODER;

architecture my_dec of DECODER is
begin
   with SEL select
   F <= "0001" when "00",
        "0010" when "01",
        "0100" when "10",
        "1000" when "11",
        "0000" when others;
end my_dec;
```

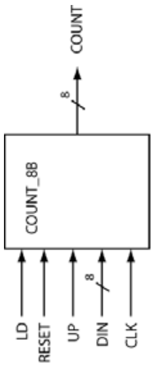
</td></tr>
<tr><td>8-bit register with load enable</td><td>



</td><td>

```
entity REG is
   port ( LD,CLK : in std_logic;
            D_IN : in std_logic_vector (7 downto 0);
           D_OUT : out std_logic_vector (7 downto 0));
end REG;

architecture my_reg of REG is
begin
   process (CLK,LD)
   begin
      if (LD = '1' and rising_edge(CLK)) then
         D_OUT <= D_IN;
      end if;
   end process;
end my_reg;
```

</td></tr>
<tr><td>8-bit up/down counter with asynchronous reset</td><td>



</td><td>

```
entity COUNT_8B is
   port ( RESET,CLK,LD,UP : in std_logic;
                      DIN : in std_logic_vector (7 downto 0);
                    COUNT : out std_logic_vector (7 downto 0));
end COUNT_8B;

architecture my_count of COUNT_8B is
   signal  t_cnt : std_logic_vector(7 downto 0);
begin
   process (CLK, RESET)
   begin
      if (RESET = '1') then
         t_cnt <= (others => ‘0’); -- clear
      elsif (rising_edge(CLK)) then
         if (LD = '1') then     t_cnt <= DIN;  -- load
         else
            if (UP = '1') then  t_cnt <= t_cnt + 1; -- incr
            else                t_cnt <= t_cnt - 1; -- decr
            end if;
         end if;
      end if;
   end process;
   COUNT <= t_cnt;
end my_count;
```

</td></tr>
</table>

Finite State Machine Modeling using VHDL Behavioral Models

```
entity fsm is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                 Z1,Z2 : out std_logic);
end fsm;
```

Description of FSM's interface (I/O)

```
architecture my_fsm of fsm is
   type state_type is (ST0,ST1,ST2,ST3);
   signal PS,NS : state_type;
begin
```

Declaration of VHDL type used to represent the states of the FSM.

```
sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '1') then PS <= ST0;
     elsif (rising_edge(CLK)) then PS <= NS;
     end if;
   end process sync_proc;
```

The "synchronous" process to control FSM parameters associated with the state variable representation (storage elements).

The combinatorial process to handles output decoding and next-state assignments.

```
comb_proc: process(PS,X)
   begin
      Z1 <= '0'; Z2 <= '0';
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST1; Z2 <= '0';
            else  NS <= ST0; Z2 <= '1';
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST2; Z2 <= '0';
            else  NS <= ST1; Z2 <= '1';
            end if;
         when ST2 =>    -- items regarding state ST2
            Z1 <= '0';  -- Moore output
            if (X = '0') then NS <= ST3; Z2 <= '0';
            else  NS <= ST2; Z2 <= '1';
            end if;
         when ST3 =>    -- items regarding state ST3
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST3;  Z2 <= '1';
            end if;
         when others => -- the catch all default case
            NS <= ST0; Z1 <= '0'; Z2 <= '0';
      end case;
   end process comb_proc;
```

All outputs are assigned initial values at start of process.

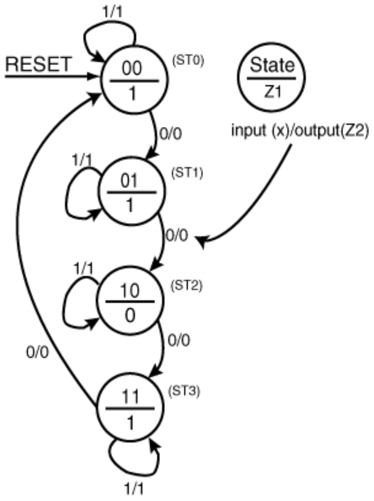
The "cases": one case for each FSM state (and a default case too).

Moore output assigments are outside conditional statement

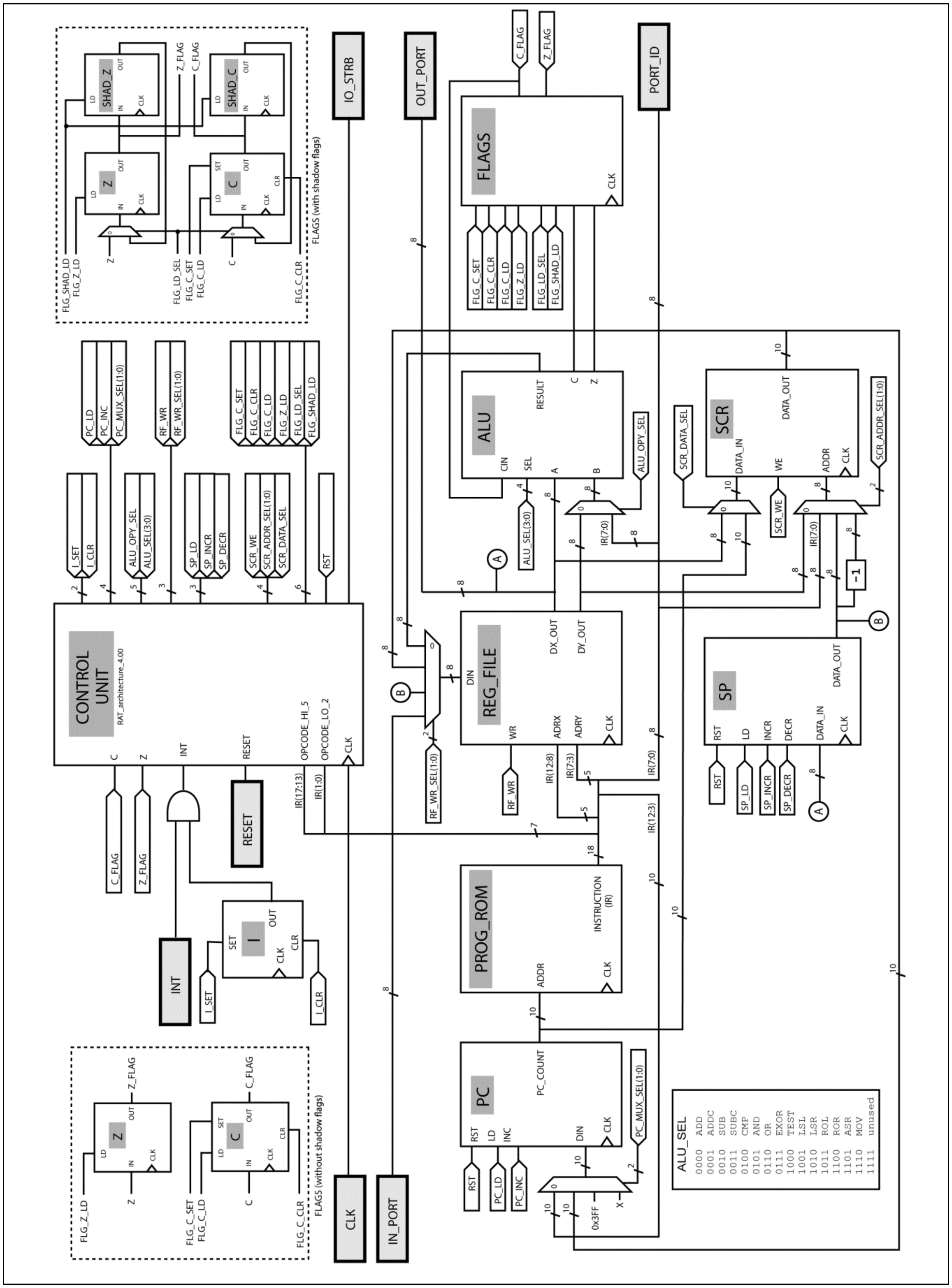
Mealy output assignments and next state assigments are inside the conditional statement.

```
with PS select
   Y <= "00" when ST0,
        "01" when ST1,
        "10" when ST2,
        "11" when ST3,
        "00" when others;
end my_fsm;
```

Concurrent statement to provide desired output assignments based on FSM states; required if the state variables are used as FSM outputs.



RAT MCU Architecture Diagram



VHDL TestBenches: A Quick Overview

Learning Objectives

1. VHDL modeling
   - To learn the basic of writing VHDL test benches

2. The Simulator
   - To learn to simulate your circuits using the simulator
   - To get a feel for some of the power and features of the simulator

Introduction and Overview

Testbenches are an important part of VHDL modeling. Up to this point in your VHDL career, you've probably have never directly written your own test bench when you simulated your circuits. This is because you've probably up to this point used nothing but the Xilinx ISE software. The approach taken by the ISE package is to make the design methodology as comfortable as possible for everyone using the software. One of those forms of comfort was the *Testbencher* software embedded in the ISE package. This software allowed you to visually select waveforms to act as test vectors for the circuit you need to test. For better or worse, the Testbencher software is no longer included in the Xilinx ISE distribution, which means you must understand some of the fundamental concepts of the official VHDL testbench.

VHDL Test Benches

Most of your VHDL career up to this point has been focused on designing circuits that were intended to be synthesized which is of course one of the powerful points of VHDL. But keep in mind that one of the other powerful characteristics of VHDL is the ability to act as a simulator. In other words, VHDL is so versatile that it can also act as a test stimulus language. Once again, out here in academic-land, the main focus of courses that use VHDL is in the design and generation of circuits; the testing portion of circuit design is unfortunately highly attenuated due primarily to time constraints.

In the real world, the up-front verification of circuit operation is critical to the success of any project. As you know, the earlier you catch errors, the easier they are to fix and they'll have less tendency to generate more errors and bad design decisions along the way. This is massively important in the case of ASIC design in that obtaining the actual design on custom silicon is going cost you about a million bucks. In the end, if you play your cards right, you'll be using simulation as your primary design tool. Keep in mind that VHDL was originally designed as a tool to allow you to generate test vectors and simulate designs.

VHDL test benches can be anywhere from quite simple to massively complex depending on the intended purpose of the design. The have the ability to read from files and write results to files. Often times, the test bench can become more complicated than the actual circuit you are testing. A general model of a test bench is shown in Figure 82. The test bench comprises of two main components: the *stimulus driver* and the *design under test*. These two black boxes are typically referred to by a bunch of different names but the functions are still the same. The design under test, or *DUT* as the cool VHDL people refer to it, is the VHDL model you've probably designed and now intend to test. The stimulus driver is a VHDL model that communicates with the DUT. The main idea here is that the stimulus driver provides test vectors to the DUT and examines results. In actuality, one thing that is not shown in Figure 82 is the fact that the stimulus driver can easily interact with the external environment. This allows for reading test vectors from files and writing various data and status notes to files.

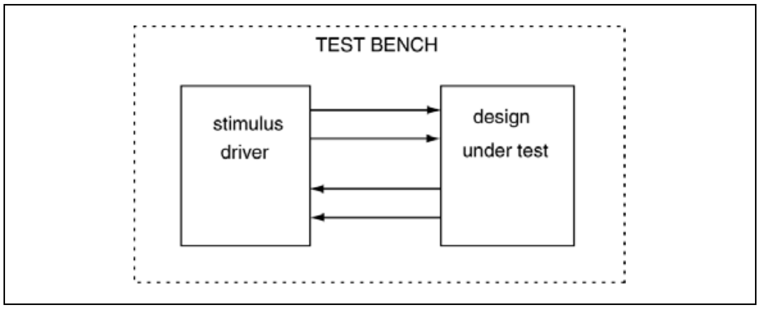


Figure 79: The general model of a VHDL test bench.

The stimulus driver is really nothing special in terms of a VHDL model. The main difference here is that instead of dealing with signals which interface with the outside world (such a switches and LEDs), we're now dealing with signals that are driving the unit we intend to test. Note that in Figure 82 there are no signals touching the dotted line; therefore, you can probably guess right away that the entity declaration of the test bench is going to be fairly simple. And one last note before we go on… this is just a brief introduction to the concept of writing test benches.

A Simple Test Bench

Let's write a simple test bench and fill in the blanks as we go along. First thing to note is that the test bench is necessarily going to be structural model as the diagram in Figure 82 implies. One approach would be to make it purely structural and which would require the instantiation of both the stimulus driver as well as the device under test. A better approach, especially considering that our test benches won't be overly complex, is to instantiate only the device under test and to model the stimulus using concurrent statements in the body of the test bench architecture. This is what is shown in Figure 83.

```
entity my_nand2 is
        port(   a,b : in std_logic;
                f : out std_logic);
end my_nand2;

architecture my_nand2 of my_nand2 is
begin
   f <= a NAND b;
end my_nand2;


entity testnand is
end testnand;

architecture stimulus of testnand is

   component my_nand2
        port(   a,b : in std_logic;
                f : out std_logic);
   end component;

   signal A,B,Y : std_logic;

begin
   -- instantiate the NAND gate
   n1: my_nand2
   port map (a,b,y);

   process
      constant PERIOD: time := 40 ns;  -- let’s be generic
   begin
      A <= '1';  B <= '1';
      wait for PERIOD;

      A <= '1';  B <= '0';
      wait for PERIOD;

      A <= '0';  B <= '1';
      wait for PERIOD;

      A <= '0';  B <= '0';
      wait for PERIOD;
      wait;
   end process;
end stimulus;
```

Figure 80: A test bench for a 2-input NAND gate.

Here are some interesting things to note about the test bench provided in Figure 83:

- The first portion of the test bench is a 2-input NAND gate. This simple gate is considered the device under test for this example. There is nothing else worth noting in this model.

- The entity declaration for the test bench is missing a port clause. This is because the test bench module essentially has no contact with the outside world. In other words, the port clause is used to describe the model’s interface; the test bench model in this example has no interface with the outside world.

- The NAND gate is declared and instantiated as you would with any circuit. The only thing new here is that there is nothing new here: massive similarities with other models you’ve created.

- Some intermediate signals have been declared. These act as the inputs to the device under test. Note there is no output from the device under test in this particular model.

- The architecture body has one process. This one process provides the required stimulus to the device under test. The process contains no sensitivity list which requires that it rely on wait statements instead. The process uses the *wait for* form of wait statements which happens to be the one type we did not discuss in the previous section of these notes. Be sure to note that the coding of the signals in the process statement under control of the wait statements has a hard-coded feel. Although this approach somewhat lacks genericity, it's adequate for what we need to do.

- Also worthy of note is the use of a VHDL constant. This helps make the test bench more generic and brings general happiness to the VHDL Gods.

- And here is how this test bench operates.
  - the simulation starts
  - the process is executed
  - the values of '1' are assigned to both the A and B signal
  - the process suspends (but only remains suspended for the stated period)
  - when the process resumes, the value of the A and B signals are reassigned before another wait statement is encountered; this happens a few times.
  - A solitary *wait* statement is encountered

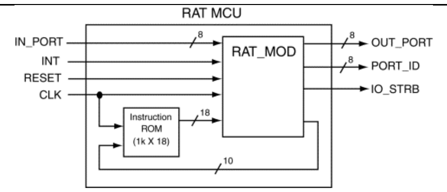
- The final wait statement has no parameters. This means that the process is essentially dead because there is nothing to make it resume. The simulation has therefore ended.

Figure 84 shows a cleaner example of a test bench. The main difference between this test bench and the previous one is that the both the processes have a way to officially terminate, never to be resumed again. Keep in mind that you are free to stop the simulation anytime you want but… it just seems more complete to have all the processes end before the simulation stops. The dual termination mechanism relies on the ability for the two processes to communicate with one another. So once the process that is generating the A and B test signals is "done", it asserts the done signal which in turns terminates the process that is generating the clock.

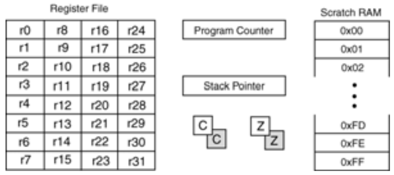
```
entity testnand3 is end testnand3;

architecture stimulus of testnand3 is

   component my_nand2
        port(   a,b : in std_logic;
                f : out std_logic);
   end component;

   signal A,B,Y : std_logic;
   signal CLK : std_logic := '0';
   constant PERIOD: time := 40 ns;
   signal done : std_logic := '0';

begin
   n1: my_nand2
   port map (a,b,y);

   my_clk: process
   begin
      while (done = '0') loop
          wait for PERIOD/2;
          CLK <= not CLK;
      end loop;
      wait;
   end process;

   process
   begin
      A <= '1';  B <= '1';
      wait for PERIOD;
      A <= '1';  B <= '0';
      wait for PERIOD;
      A <= '0';  B <= '1';
      wait for PERIOD;
      A <= '0';  B <= '0';
      wait for PERIOD;
      done <= '1';
      wait;
   end process;
end stimulus;
```

Figure 81: A test bench for a 2-input NAND gate with two processes that terminate.

RAT MCU Architecture and Assembly Language Cheat Sheet



schematic diagram



programming model

RAT Instruction Set:

Program Control			Interrupt	Input/Output
`BREQ label`	`BRN label`	`CLC`	`RETID`	`IN rX,pp`
`BRNE label`		`SEC`	`RETIE`	`OUT rX,pp`
`BRCS label`	`CALL label`		`SEI`	
`BRCC label`	`RET`	`WSP rX`	`CLI`	

Logical		Arithmetic		Shift & Rotate	Storage	
`AND rX,kk`	`AND rX,rY`	`ADD rX,kk`	`ADD rX,rY`	`LSL rX`	`ST rX,imm`	`PUSH rX`
`OR rX,kk`	`OR rX,rY`	`ADDC rX,kk`	`ADDC rX,rY`	`LSR rX`	`ST rX,(rY)`	`POP rX`
`EXOR rX,kk`	`EXOR rX,rY`	`SUB rX,kk`	`SUB rX,rY`	`ROL rX`	`LD rX,imm`	
`TEST rX,kk`	`TEST rX,rY`	`SUBC rX,kk`	`SUBC rX,rY`	`ROR rX`	`LD rX,(rY)`	`MOV rX,rY`
		`CMP rX,kk`	`CMP rX,rY`	`ASR rX`		`MOV rX,imm`

Fun Facts:

- Max program size: 1024 instructions (18-bit/instr)
- 32 8-bit general purpose registers (GPRs)
- Stack: implemented in scratch RAM

I/O:

- Port Mapped device (8-bit port addresses)
- 8-bit GPR-based Input and Output

Interrupt Architecture:

- Interrupt on: interrupt input high voltage & interrupt enabled
- One maskable external interrupt (see interrupt group)
- Vector interrupt: vector address 0x3FF
- Context saving: C & Z flags saved on interrupt
- Interrupt automatically disabled on interrupt
- Context restoration: C & Z flags restored on RETID or RETIE instructions

Bit Masking:

bit setting: OR with ‘1’	bit clearing: AND with ‘0’	bit toggling: XOR with ‘1’
`OR r3,0x01 ;set bit 0`	`AND r3,0xFE ;clear bit 0:`	`EXOR r3,0x01 ;toggle bit 0`

Conditional Constructs:

if/else construct

input byte; if byte is 0x00, then r1=0x0A; otherwise, r1=0x0B output r1

```
            IN       r0,IN_PORT     ; grab data
            CMP      r0,0x00        ; test to see if byte is 0x00
            BRNE     not_zero       ; jump if r0 is not 0x00
            MOV      r1,0x0A        ; place 0x0A in r1
            BRN      out_val        ; jump to output instruction
                                    ;
not_zero:   MOV      r1,0x0B        ; place 0x0B in r1
out_val:    OUT      r1,OUT_PORT    ; output some data
```

iterative construct
(known iterative value)

place iterative value in r3; do something; decrement count; check to see if zero; repeat if not zero

```
            MOV      r3,0x08        ; load iterative count value
loop:       ;                       ; do something meaningful
            ;
            SUB      r3,0x01        ; decrement iteration variable
            BRNE     loop           ; do it again if count non-zero
            ;
loop_done:  ; do something else…
```

conditional construct
(unknown iterative value)

Input value; increment it; do it again if carry flag is not set

```
            MOV      r0,0x00        ; clear register
loop:       IN       r1,IN_PORT     ; grab some data
            ADD      r0,r1          ; add some value to r0
            BRCC     loop           ; repeat if no carry

loop_done:  ; do something else…
```

Equality/Inequality:

- C and Z flag are used to establish relationship between registers:

Operation:	`SUB rA, rB`
	`CMP rA, rB`	

C	**Z**	**Comment**
0	0	rA > rB
1	0	rA < rB
-	1	rA = rB

RAT MCU Assembly Language Style File

The following file shows some of the more important issues regarding generating neat and readable RAT MCU assembly source code. No style file can show you everything and they rarely make such an attempt. The underlying factor in writing any source code is to be neat and consistent. Using proper indentation, white space and commenting helps you attain the goals of being neat and consistent. The code in the following block is a modified fragment from another working program; if it were to actually assemble, it would not do anything useful. We present this program primarily for appearance purposes.

```
;--------------------------------------------------------------------
;- Programmer: Pat Wankaholic
;- Date: 09-29-10
;- Experiment #??
;-
;- This program does something really cool. Here's the description...
;--------------------------------------------------------------------

;--------------------------------------------------------------------
;- I/O Constants
;--------------------------------------------------------------------
.EQU SWITCH_PORT = 0x30          ; port for switches ----- INPUT
.EQU LED_PORT    = 0x0C          ; port for LED output --- OUTOUT
.EQU BTN_PORT    = 0x10          ; port for button input - INPUT
;--------------------------------------------------------------------

;--------------------------------------------------------------------
;- Misc Constants
;--------------------------------------------------------------------
.EQU BTN2_MASK = 0x08            ; mask all but BTN5
.EQU B0_MASK   = 0x01            ; mask all but bit0
;--------------------------------------------------------------------

;--------------------------------------------------------------------
;- Memory Designation Constants
;--------------------------------------------------------------------
.DSEG
.ORG     0x00

COW:   .DB 0x09,0x07,0x06,0x05
;--------------------------------------------------------------------
;
.CSEG
.ORG         0x01

init:        SEI                       ; enable interrupts

main_loop:   IN      r0, BTN_PORT      ; input status of buttons
             AND     r0, BTN2_MASK     ; clear all but BTN2
             BRN     bit_wank          ; jumps when BTN2 is pressed

             ;----------------------------------------------------------
             ;- nibble wank portion of code
             ;----------------------------------------------------------
wank:        ROL     r1                ; rotate 2 times - msb-->lsb
bit3:        BRN     fin_out           ; jump unconditionally to led output
             ;----------------------------------------------------------

             ;----------------------------------------------------------
             ; bit-wank algo: do something Blah, blah, blah ...
             ;----------------------------------------------------------
```

```
bit_wank:    LD      r0,0x00           ; clear r0 for working register

             OR      r0, B1_MASK       ; set bit1
bit2:        LSR     r1                ; shift msb into carry bit
             BRCS    bit3              ; jump if carry not set
             ;-------------------------------------------------------------

             CALL    My_sub           ; subroutine call
fin_out:     OUT     r0,LED_PORT      ; output data to LEDs
             BRN     main_loop        ; endless loop
             ;-------------------------------------------------------------


             ;-------------------------------------------------------------
             ; Subroutine: My_sub:
             ;
             ; This routines does something useful. It expects to find
             ; some special data in registers r0, r1, and r2. It changes
             ; the contents of registers blah, blah, blah...
             ;
             ; Tweaked registers: r1
             ;--------------------------------------------------------------
My_sub:      LSR     r1               ; shift msb into carry bit
             BRCS    bit3             ; jump if carry not set
             RET
```

VHDL Cheat-Sheet

Concurrent Statements		**Sequential Statements**
Concurrent Signal Assignment **(dataflow model)**	⇔	**Signal Assignment**
target <= *expression*;		*target* <= *expression*;

A <= B **AND** C;
DAT <= (D **AND** E) **OR** (F **AND** G);
		A <= B **AND** C;
		DAT <= (D **AND** E) OR (F **AND** G);

Conditional Signal Assignment **(dataflow model)**	⇔	***if* statements**

target <= *expressn* **when** *condition* **else**
          expressn **when** *condition* **else**
          expressn;
		if (*condition*) **then**
		   { sequence of statements }
		elsif (*condition*) **then**
		   { sequence of statements }
		else --(the else is optional)
		   { sequence of statements }
		end if;

F3 <= ‘1’ **when** (L=‘0’ **AND** M=‘0’) **else**
      ‘1’ **when** (L=‘1’ **AND** M=‘1’) **else**
      ‘0’;
		if (SEL = “111”) **then** F_CTRL <= D(7);
		elsif (SEL = “110”) **then** F_CTRL <= D(6);
		elsif (SEL = “101”) **then** F_CTRL <= D(1);
		elsif (SEL = “000”) **then** F_CTRL <= D(0);
		else F_CTRL <= ‘0’;
		end if;

Selective Signal Assignment **(dataflow model)**	⇔	***case* statements**

with *chooser_expression* **select**
   target <= *expression* **when** *choices*,
             expression **when** *choices;*
		case (*expression*) **is**
		   when *choices* =>
		       {sequential statements}
		   when *choices* =>
		       {sequential statements}
		   when others => -- (optional)
		       {sequential statements}
		end case;

with SEL **select**
MX_OUT <= D3 **when** “11”,
          D2 **when** “10”,
          D1 **when** “01”,
          D0 **when** “00”,
          ‘0’ **when others**;
		case ABC **is**
		    when “100” **=>** F_OUT **<=** ‘1’;
		    when “011” **=>** F_OUT **<=** ‘1’;
		    when “111” **=>** F_OUT **<=** ‘1’;
		    when others => F_OUT **<=** ‘0’;
		end case;

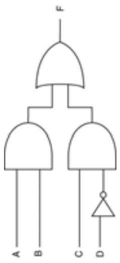
Process **(behavioral model)**		

opt_label: **process**(*sensitivity_list*)
begin
   {*sequential_statements*}
end process *opt_label*;		

proc1: **process**(A,B,C)
begin
   if (A = ‘1’ **and** B = ‘0’**) then**
      F_OUT **<=** ‘1’;
   elsif (B = ‘1’ **and** C = ‘1’) **then**
      F_OUT **<=** ‘1’;
   else
      F_OUT **<=** ‘0’;
   end if;
end process proc1;		

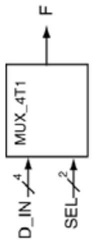
Description	CKT Diagram	VHDL Model

Typical logic circuit



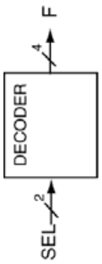
```
entity my_ckt is
   Port ( A,B,C,D : in std_logic;
                F : out std_logic);
end my_ckt;

architecture ckt1 of my_ckt is
begin
   F <= (A AND B) OR (C AND (NOT D));
end ckt1;

architecture ckt2 of my_ckt is
begin
   F <= ‘1’ when (A = ‘1’ AND B = ‘1’) else
        ‘1’ when (C = ‘1’ AND D = ‘0’) else
        ‘0’;
end ckt2;
```

4:1 Multiplexor



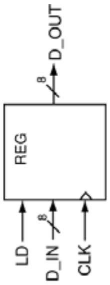
```
entity MUX_4T1 is
   Port (  SEL : in std_logic_vector(1 downto 0);
          D_IN : in std_logic_vector(3 downto 0);
             F : out std_logic);
end MUX_4T1;

architecture my_mux of MUX_4T1 is
begin
   F <= D_IN(0) when (SEL = "00") else
        D_IN(1) when (SEL = "01") else
        D_IN(2) when (SEL = "10") else
        D_IN(3) when (SEL = "11") else
        '0';
end my_mux;
```

2:4 Decoder



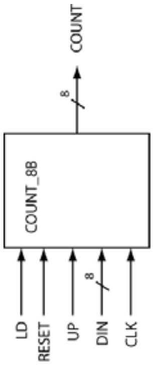
```
entity DECODER is
   Port ( SEL : in std_logic_vector(1 downto 0);
            F : out std_logic_vector(3 downto 0));
end DECODER;

architecture my_dec of DECODER is
begin
   with SEL select
   F <= "0001" when "00",
        "0010" when "01",
        "0100" when "10",
        "1000" when "11",
        "0000" when others;
end my_dec;
```

8-bit register with load enable



```
entity REG is
   port ( LD,CLK : in std_logic;
            D_IN : in std_logic_vector (7 downto 0);
           D_OUT : out std_logic_vector (7 downto 0));
end REG;

architecture my_reg of REG is
begin
   process (CLK,LD)
   begin
      if (LD = '1' and rising_edge(CLK)) then
         D_OUT <= D_IN;
      end if;
   end process;
end my_reg;
```

8-bit up/down counter with asynchronous reset



```
entity COUNT_8B is
   port ( RESET,CLK,LD,UP : in std_logic;
                      DIN : in std_logic_vector (7 downto 0);
                    COUNT : out std_logic_vector (7 downto 0));
end COUNT_8B;

architecture my_count of COUNT_8B is
   signal  t_cnt : std_logic_vector(7 downto 0);
begin
   process (CLK, RESET)
   begin
      if (RESET = '1') then
         t_cnt <= (others => ‘0’); -- clear
      elsif (rising_edge(CLK)) then
         if (LD = '1') then     t_cnt <= DIN;  -- load
         else
            if (UP = '1') then  t_cnt <= t_cnt + 1; -- incr
            else                t_cnt <= t_cnt - 1; -- decr
            end if;
         end if;
      end if;
   end process;
   COUNT <= t_cnt;
end my_count;
```

Finite State Machine Modeling using VHDL Behavioral Models

```
entity fsm is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                 Z1,Z2 : out std_logic);
end fsm;

architecture my_fsm of fsm is
   type state_type is (ST0,ST1,ST2,ST3);
   signal PS,NS : state_type;
begin

sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '1') then PS <= ST0;
     elsif (rising_edge(CLK)) then PS <= NS;
     end if;
   end process sync_proc;


comb_proc: process(PS,X)
   begin
      Z1 <= '0'; Z2 <= '0';
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST1; Z2 <= '0';
            else  NS <= ST0; Z2 <= '1';
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST2; Z2 <= '0';
            else  NS <= ST1; Z2 <= '1';
            end if;
         when ST2 =>    -- items regarding state ST2
            Z1 <= '0';  -- Moore output
            if (X = '0') then NS <= ST3; Z2 <= '0';
            else  NS <= ST2; Z2 <= '1';
            end if;
         when ST3 =>    -- items regarding state ST3
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST3;  Z2 <= '1';
            end if;
         when others => -- the catch all default case
            NS <= ST0; Z1 <= '0'; Z2 <= '0';
      end case;
   end process comb_proc;

with PS select
   Y <= "00" when ST0,
        "01" when ST1,
        "10" when ST2,
        "11" when ST3,
        "00" when others;
end my_fsm;
```

Description of FSM's interface (I/O)

Declaration of VHDL type used to represent the states of the FSM.

The "synchronous" process to control FSM parameters associated with the state variable representation (storage elements).

The combinatorial process to handles output decoding and next-state assignments.

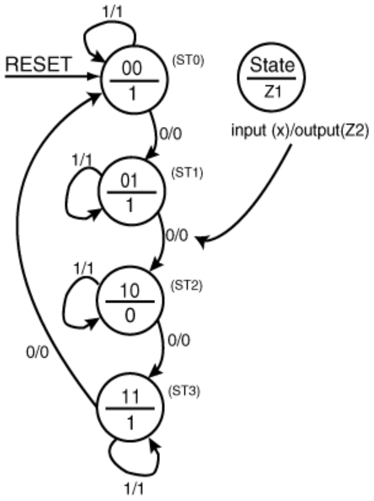
All outputs are assigned initial values at start of process.

The "cases": one case for each FSM state (and a default case too).

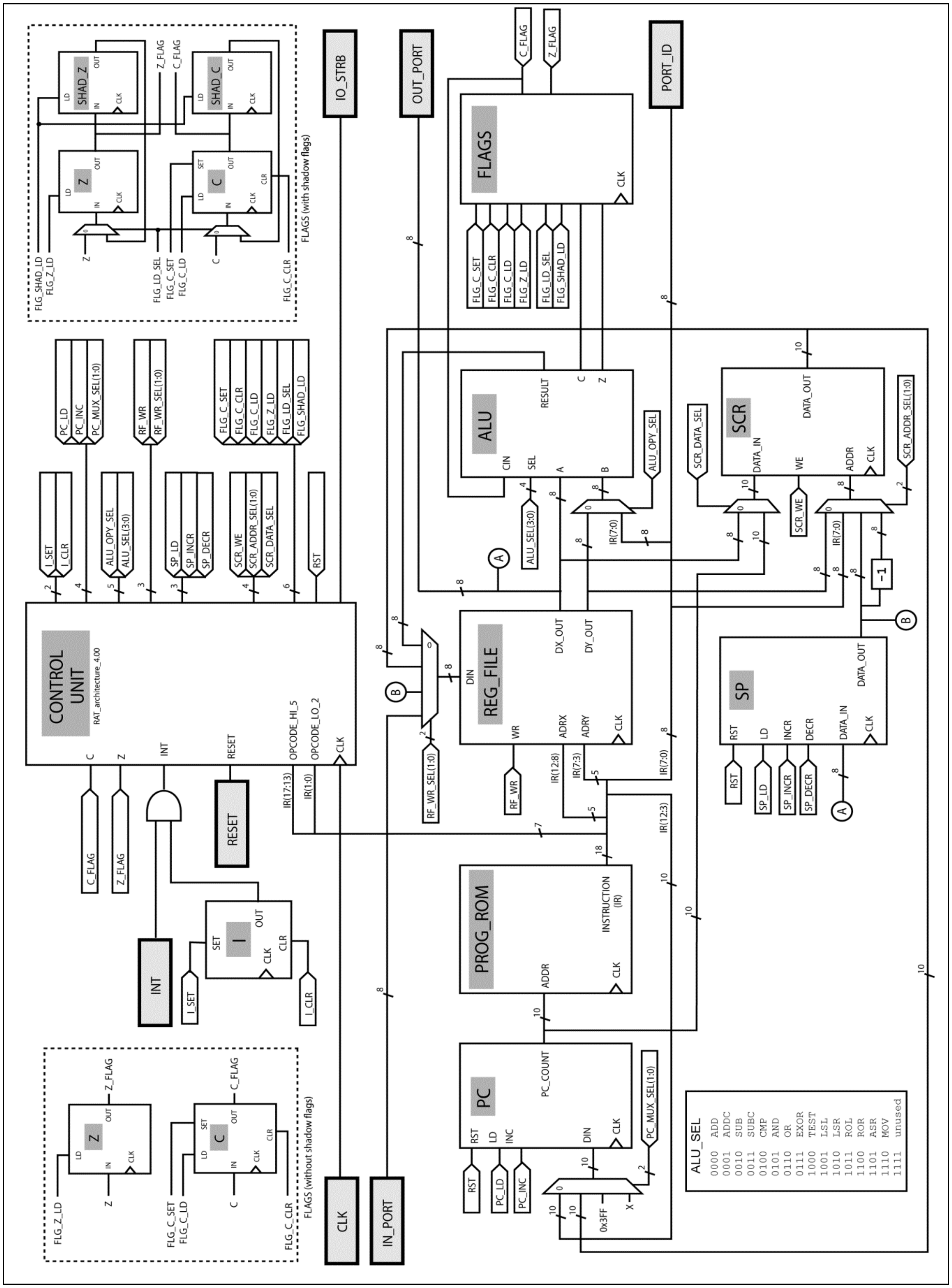
Moore output assigments are outside conditional statement

Mealy output assignments and next state assigments are inside the conditional statement.

Concurrent statement to provide desired output assignments based on FSM states; required if the state variables are used as FSM outputs.



RAT MCU Architecture Diagram



VHDL TestBenches: A Quick Overview

Learning Objectives

3. VHDL modeling
    - To learn the basic of writing VHDL test benches

4. The Simulator
    - To learn to simulate your circuits using the simulator
    - To get a feel for some of the power and features of the simulator

Introduction and Overview

Testbenches are an important part of VHDL modeling. Up to this point in your VHDL career, you've probably have never directly written your own test bench when you simulated your circuits. This is because you've probably up to this point used nothing but the Xilinx ISE software. The approach taken by the ISE package is to make the design methodology as comfortable as possible for everyone using the software. One of those forms of comfort was the *Testbencher* software embedded in the ISE package. This software allowed you to visually select waveforms to act as test vectors for the circuit you need to test. For better or worse, the Testbencher software is no longer included in the Xilinx ISE distribution, which means you must understand some of the fundamental concepts of the official VHDL testbench.

VHDL Test Benches

Most of your VHDL career up to this point has been focused on designing circuits that were intended to be synthesized which is of course one of the powerful points of VHDL. But keep in mind that one of the other powerful characteristics of VHDL is the ability to act as a simulator. In other words, VHDL is so versatile that it can also act as a test stimulus language. Once again, out here in academic-land, the main focus of courses that use VHDL is in the design and generation of circuits; the testing portion of circuit design is unfortunately highly attenuated due primarily to time constraints.

In the real world, the up-front verification of circuit operation is critical to the success of any project. As you know, the earlier you catch errors, the easier they are to fix and they'll have less tendency to generate more errors and bad design decisions along the way. This is massively important in the case of ASIC design in that obtaining the actual design on custom silicon is going cost you about a million bucks. In the end, if you play your cards right, you'll be using simulation as your primary design tool. Keep in mind that VHDL was originally designed as a tool to allow you to generate test vectors and simulate designs.

VHDL test benches can be anywhere from quite simple to massively complex depending on the intended purpose of the design. The have the ability to read from files and write results to files. Often times, the test bench can become more complicated than the actual circuit you are testing. A general model of a test bench is shown in Figure 82. The test bench comprises of two main components: the *stimulus driver* and the *design under test*. These two black boxes are typically referred to by a bunch of different names but the functions are still the same. The design under test, or *DUT* as the cool VHDL people refer to it, is the VHDL model you've probably designed and now intend to test. The stimulus driver is a VHDL model that communicates with the DUT. The main idea here is that the stimulus driver provides test vectors to the DUT and examines results. In actuality, one thing that is not shown in Figure 82 is the fact that the stimulus driver can easily interact with the external environment. This allows for reading test vectors from files and writing various data and status notes to files.

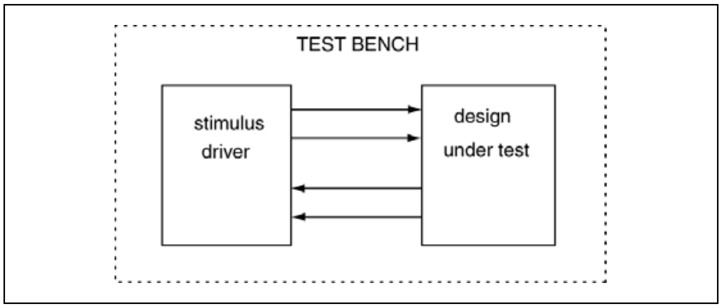


Figure 82: The general model of a VHDL test bench.

The stimulus driver is really nothing special in terms of a VHDL model. The main difference here is that instead of dealing with signals which interface with the outside world (such a switches and LEDs), we're now dealing with signals that are driving the unit we intend to test. Note that in Figure 82 there are no signals touching the dotted line; therefore, you can probably guess right away that the entity declaration of the test bench is going to be fairly simple. And one last note before we go on… this is just a brief introduction to the concept of writing test benches.

A Simple Test Bench

Let's write a simple test bench and fill in the blanks as we go along. First thing to note is that the test bench is necessarily going to be structural model as the diagram in Figure 82 implies. One approach would be to make it purely structural and which would require the instantiation of both the stimulus driver as well as the device under test. A better approach, especially considering that our test benches won't be overly complex, is to instantiate only the device under test and to model the stimulus using concurrent statements in the body of the test bench architecture. This is what is shown in Figure 83.

```
entity my_nand2 is
       port(   a,b : in std_logic;
               f : out std_logic);
end my_nand2;

architecture my_nand2 of my_nand2 is
begin
   f <= a NAND b;
end my_nand2;


entity testnand is
end testnand;

architecture stimulus of testnand is

   component my_nand2
       port(   a,b : in std_logic;
               f : out std_logic);
   end component;

   signal A,B,Y : std_logic;

begin
   -- instantiate the NAND gate
   n1: my_nand2
   port map (a,b,y);

   process
      constant PERIOD: time := 40 ns;  -- let’s be generic
   begin
      A <= '1';  B <= '1';
      wait for PERIOD;

      A <= '1';  B <= '0';
      wait for PERIOD;

      A <= '0';  B <= '1';
      wait for PERIOD;

      A <= '0';  B <= '0';
      wait for PERIOD;
      wait;
   end process;
end stimulus;
```

Figure 83: A test bench for a 2-input NAND gate.

Here are some interesting things to note about the test bench provided in Figure 83:

- The first portion of the test bench is a 2-input NAND gate. This simple gate is considered the device under test for this example. There is nothing else worth noting in this model.

- The entity declaration for the test bench is missing a port clause. This is because the test bench module essentially has no contact with the outside world. In other words, the port clause is used to describe the model’s interface; the test bench model in this example has no interface with the outside world.

- The NAND gate is declared and instantiated as you would with any circuit. The only thing new here is that there is nothing new here: massive similarities with other models you’ve created.

- Some intermediate signals have been declared. These act as the inputs to the device under test. Note there is no output from the device under test in this particular model.

- The architecture body has one process. This one process provides the required stimulus to the device under test. The process contains no sensitivity list which requires that it rely on wait statements instead. The process uses the *wait for* form of wait statements which happens to be the one type we did not discuss in the previous section of these notes. Be sure to note that the coding of the signals in the process statement under control of the wait statements has a hard-coded feel. Although this approach somewhat lacks genericity, it's adequate for what we need to do.

- Also worthy of note is the use of a VHDL constant. This helps make the test bench more generic and brings general happiness to the VHDL Gods.

- And here is how this test bench operates.
    - the simulation starts
    - the process is executed
    - the values of '1' are assigned to both the A and B signal
    - the process suspends (but only remains suspended for the stated period)
    - when the process resumes, the value of the A and B signals are reassigned before another wait statement is encountered; this happens a few times.
    - A solitary *wait* statement is encountered

- The final wait statement has no parameters. This means that the process is essentially dead because there is nothing to make it resume. The simulation has therefore ended.

Figure 84 shows a cleaner example of a test bench. The main difference between this test bench and the previous one is that the both the processes have a way to officially terminate, never to be resumed again. Keep in mind that you are free to stop the simulation anytime you want but… it just seems more complete to have all the processes end before the simulation stops. The dual termination mechanism relies on the ability for the two processes to communicate with one another. So once the process that is generating the A and B test signals is "done", it asserts the done signal which in turns terminates the process that is generating the clock.

```
entity testnand3 is end testnand3;

architecture stimulus of testnand3 is

   component my_nand2
        port(   a,b : in std_logic;
                f : out std_logic);
   end component;

   signal A,B,Y : std_logic;
   signal CLK : std_logic := '0';
   constant PERIOD: time := 40 ns;
   signal done : std_logic := '0';

begin
   n1: my_nand2
   port map (a,b,y);

   my_clk: process
   begin
      while (done = '0') loop
          wait for PERIOD/2;
          CLK <= not CLK;
      end loop;
      wait;
   end process;

   process
   begin
      A <= '1';  B <= '1';
      wait for PERIOD;
      A <= '1';  B <= '0';
      wait for PERIOD;
      A <= '0';  B <= '1';
      wait for PERIOD;
      A <= '0';  B <= '0';
      wait for PERIOD;
      done <= '1';
      wait;
   end process;
end stimulus;
```

Figure 84: A test bench for a 2-input NAND gate with two processes that terminate.