# CS M152A Lab 2

## Sequencer

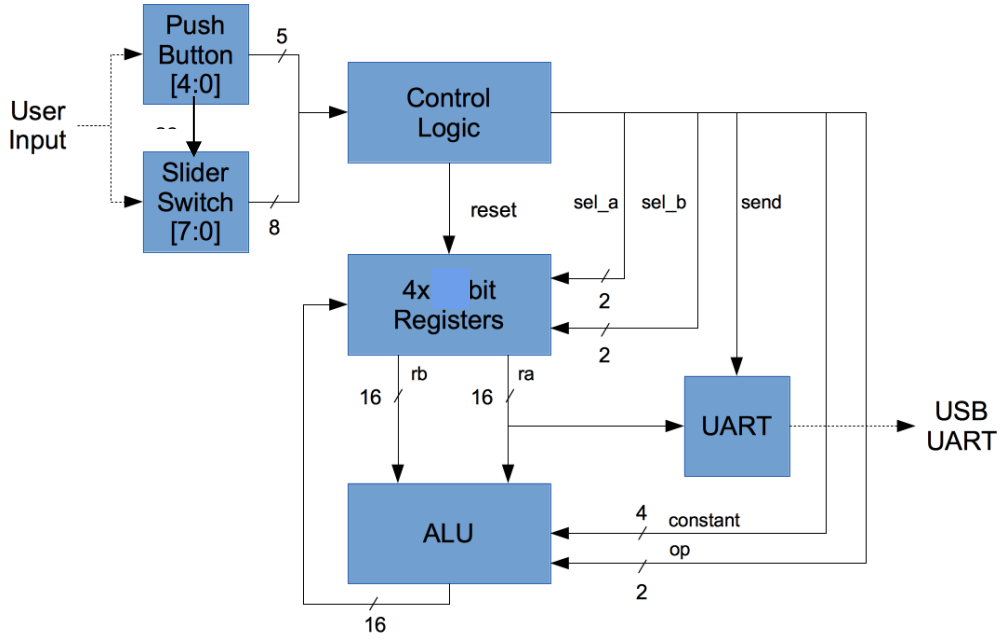
*In this lab, you will study a sequencer as an example project and do exercises based on the project.*

## Introduction

In this lab, you are expected to run a small scale FPGA project, understand it and make modifications of it. The project is a good reference for future design projects, as you can learn the styles, formats, structures of code, etc. from the project. You can also learn useful techniques, like clock dividers and debouncers, from the project. There will be questions based on the code, and there will also be a few tasks that require modifying the original code.

## Sequencer Design

The project is an adder/multiplier sequencer. It has four 16-bit general purpose registers. It can perform add or multiply instructions using registers as operands. The results are stored into register files.




Project Diagram

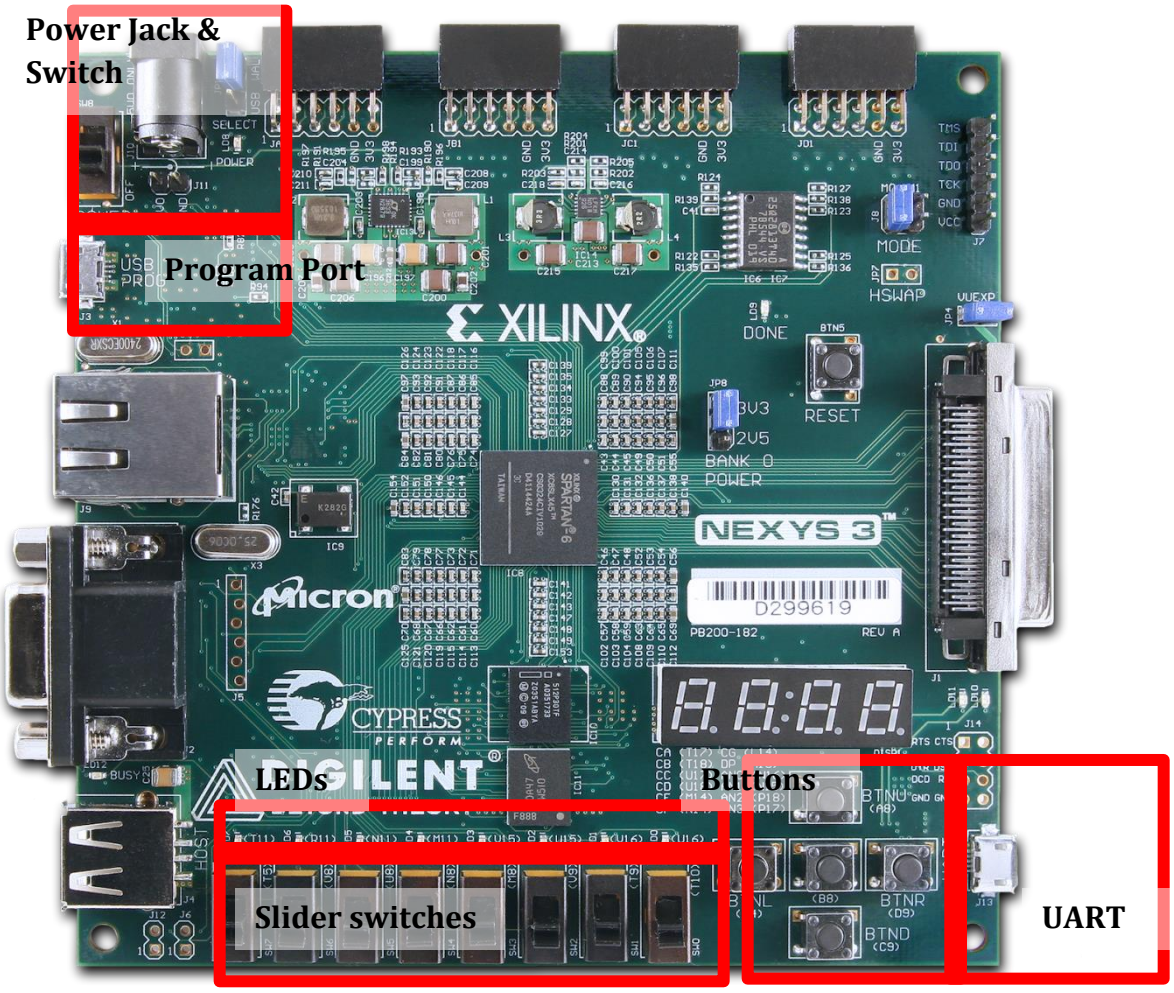
### Sequencer Operation

To operate the sequencer, the user enters instructions using the switches and push buttons:

- 8 switch positions represent a single 8-bit instruction (up: 1, down: 0)
- Push center button to enter and execute the instruction

· The push button BTNR (right button) resets values of all registers to 0

In addition, the 8 LED indicators show the number of instructions executed since the last reset, in binary.

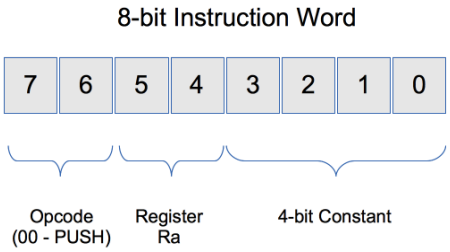



Nexys 3 Board

## Sequencer Instructions

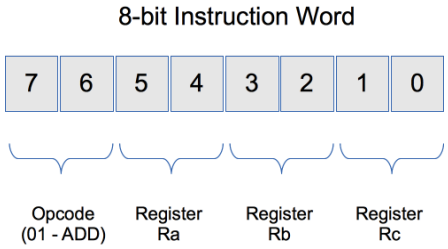
PUSH (00) instruction left-shifts the target register by 4 bits and "pushes" the new constant in. Example:

- R0 starts with value 0x55AA
- PUSH R0 0xB
- Now R0 is 0x5AAB



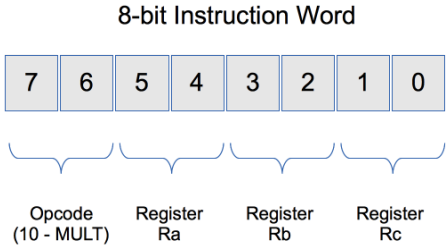

ADD (01) instruction adds Ra and Rb and stores the result to Rc. Addition is performed in unsigned integer arithmetic. Example:

- ADD R0 R1 R3
- R0 + R1 -> R3



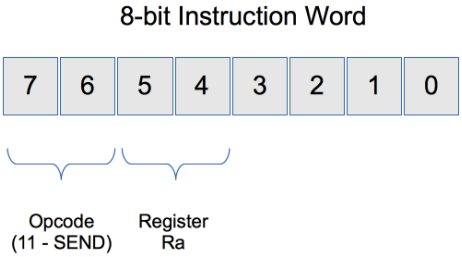

MULT (10) instruction multiplies Ra and Rb and stores the result to Rc. Inputs are assumed to be unsigned integers. Only the lower 16-bit results are retained. Example:

- MULT R3 R0 R2
- R3 * R0 -> R2




SEND (11) instruction sends the content of Ra to the UART for display. The 16-bit value is converted to ASCII-HEX and appended with a newline character. Example:

- R1 has a value of 0x1234
- SEND R1 causes the UART to send "1234\n"




## Tasks

There are 7 tasks in total. Some are related with the [Implementation] portion of the project, i.e. those in the "rtl" folder, while others are related with the [Simulation] portion of the project, i.e. those in the "tb" folder. When you see **Demo Now**, you should demo your current results to your TA.

**[Implementation] Warm Up Task.** Build the project using based on the source files in src.zip, and make sure that it works (don't forget the UCF file!). Please refer to the putty pdf to set up putty and

view uart output. Translate the following "program" into binary instructions. Demo the UART console output to your TA after you finish.

- PUSH R0 0x4
- PUSH R0 0x0
- PUSH R1 0x3
- ADD R0 R2 R2
- ADD R0 R2 R2
- ADD R0 R2 R2
- ADD R2 R0 R3
- SEND R0
- SEND R1
- SEND R2
- SEND R3

**Demo Now**. Demo your putty output to your TA. Arbitrary operations may also be checked.

### [Implementation] Missing Multiply Operation

In the current version of the code, the Multiply operation described in the lab manual is missing. Take a look at how the Add operation is coded, and create the Multiply operation on your own.

### [Implementation] A Separate SEND Button

The SEND instruction is somewhat unintuitive in the current design. Right now SEND is part of the normal instructions, called using *slider switches* and the *execute* button. In this task, we ask you to change the design to use a separate *send* button that is dedicated to the send functionality. For simplicity, the slider switch encoding can remain the same. Modify "nexys3.v", "seq.v" and the UCF file to make it happen.

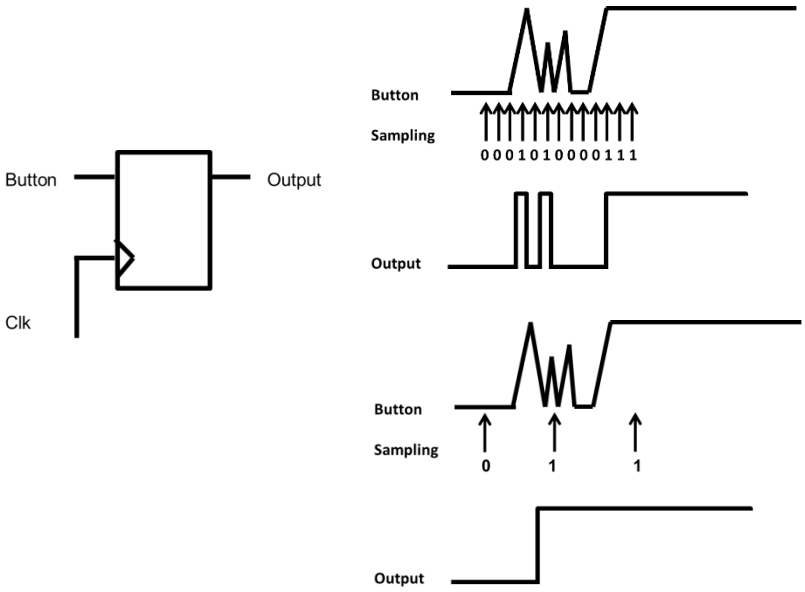
You might be confused by the complicated logic used for obtaining a stable "inst_vld" signal. Here're some explanations:

---

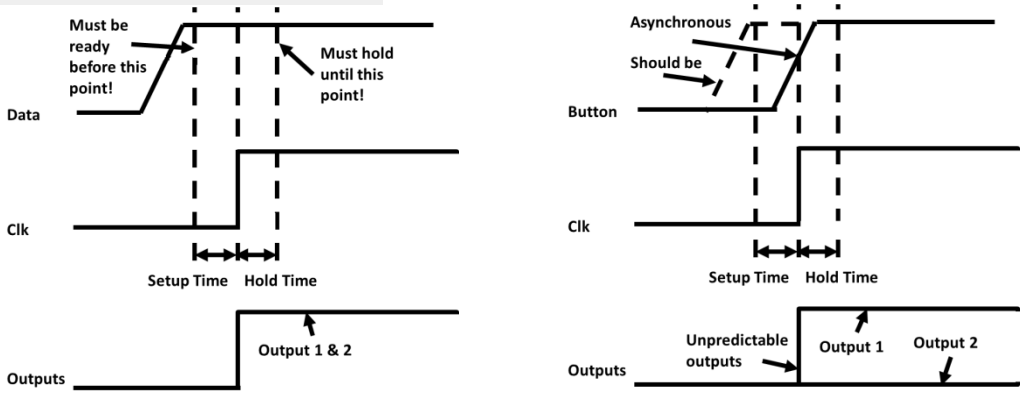
Due to physical contact instability, the buttons are very bouncy: it means that when you press the button once, it generates an oscillation of signals due to poor metal contact, and thus causing considerable **noise** in the input. A simple solution to filter out the noise is sampled at a frequency lower than that of the noise. In that case, the glitches, which may otherwise be considered by the system as a valid button push, will be filtered out (see the figure below).

In the figure, we are showing a flip-flop that samples "Button" at every positive edge of the clk and outputs "Output". If we sample at a very fast frequency (using a high-frequency clk), we may get a sequence such as "0001010000111", where the extra 0s

and 1s caused by the noise. We can simply solve the problem by sampling at a lower frequency




Another problem, **metastability**, exists because of the asynchronous nature of the button and slider switch input. To ensure reliable operation in digital circuits, the input to a register must be stable for a minimum time before the clock edge (register setup time or $t_{SU}$) and for a minimum time after the clock edge (register hold time or $t_H$), which is the signal timing requirements in its simplest form. In synchronous systems, the input signals must always meet the register timing requirements. Yet in the case of the buttons and slider switches, the signal may come at any time, causing unpredictable (and possibly different) outputs to the other modules connected to the signal. A simple solution to the problem is to use a flip-flop after the asynchronous input and regard the output of the flip-flop as the input. With that, the outputs to all other modules will be consistent.



## [Simulation] Nicer UART Output

The existing testbench program contains a uart model that announces each byte that's received by the model. While generic, the output of this model is somewhat hard to read. Modify "model_uart.v" such that the per-byte output is suppressed. Instead, model_uart.v shall output one line at a time after a carriage-return character ('\r') is received. For example, print out "0003\n" instead of "0\n" "0\n" "0\n" and "3\n" for the first line of UART output.

To understand the relevant portion of code in "model_uart.v", you'll need more knowledge on the UART protocol. Wikipedia is a good start.

Note: you are only allowed to use $display in this task (e.g. no $write is allowed). We suggest that read the UART protocol to further understand the test bench before start to work on the task.

**Demo Now**. Demo your current simulation outputs to your TA. It should print out 0040, 0003, 00C0, 0100 as the value for R0, R1, R2, R4 respectively in the console outputs.

## [Implementation] Even Nicer UART Output

Currently the print instruction only prints the *value* of a register, but not the *register number*. It's hard for us to understand what we are printing. Instead, a more intuitive way to print the content of a particular register, R0 for example, is "R0:0003". Modify "uart_top.v" and "nexys3.v" to make it happen.

When you finish, if you did write in your "Nicer UART Output" task, you should see the correct output format from the simulation console.

**Demo Now**. Send an arbitrary register value over the UART and demo the putty output to your TA.

## [Simulation] An Easier Way to Load Sequencer Program

The existing sequencer testbench sends a static sequence of instructions to the UUT (Unit Under Test) after the reset. Now we would like to change the static set of instructions. Instead, we will be loading instructions from a text file. The format of the file is the following:

1. The name of the file is "seq. code"
2. The file is up to 1024 lines long.
3. The first line of the file contains a binary number that indicates how many instructions are included in this file.
4. Each of the remaining (n-1) lines contains a single instruction in binary.

For example, here's the file-equivalent of the simple sequencer instructions currently in use:

---

Line 1: 1001
Line 2: 00000100

Line 3: 00000000
Line 4: 00010011
Line 5: 10000110
Line 6: 01100011
Line 7: 11000000
Line 8: 11010000
Line 9: 11100000
Line 10: 11110000

---

**Modify the testbench such that it loads seq.code into an array, and executes every instruction in the file.**

Hint: for file I/O, you may use the built-in Verilog system task $readmemb (google Verilog quick reference), or the c-like $fopen and $fscanf tasks.

### [Simulation] Fibonacci Numbers

Now that you have an easier way to program the sequencer from simulation, design a sequence of instructions such that the first 10 numbers of the Fibonacci series are printed from the UART.

**Demo Now**. Demo your current simulation outputs to your TA.

## Workshop

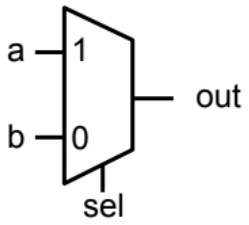
### [Implementation] Clock Dividers

In the nexys3.v file, there is a signal/reg named `clk_en`. The `clk_en` represents a clock that is much slower than the master clock clk. Read the section of code that's relevant to clk_en and try to understand how the clock divider is implemented.

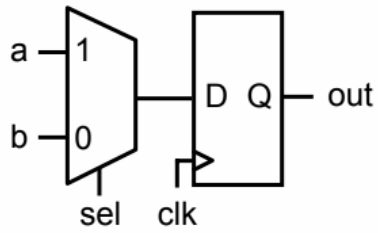
1. Add clk_en to the simulation waveform tab and then run the simulation again. Use the cursor to find the periodicity of this signal (you can select the signal and use arrow keys to reach the exact edges). Capture a waveform picture that shows two occurrences of `clk_en`, and include it in the lab report. Indicate the *exact period* of the signal in the report.
2. A duty cycle is the percentage of one period in which a signal or system is active: $D = \frac{T}{P} \times 100\%$, where D is the duty cycle, T is the interval where the signal is high, and p is the period. What is the *exact duty cycle* of `clk_en` signal?
3. What is the value of `clk_dv` signal during the clock cycle that `clk_en` is high?
4. Draw a simple schematic/diagram of signals `clk_dv`, `clk_en`, and `clk_en_d` signals. It should be a translation of the corresponding Verilog code, such as the following. NOTE: for

other lab reports you don't have to draw diagram with this much detail. A diagram outlining high level module interactions is sufficient.

```
always @ (a or b or sel)
begin
  if (sel) out = a;
  else out = b;
end
endmodule
```




```
always @ (posedge clk)
begin
  if (sel) out <= a;
  else out <= b;
end
endmodule
```




## [Implementation] Debouncing

In this project we input instructions using buttons. We have already discussed concepts of noise and metastability. In this project, the author combined debouncing with edge detection to make sure that every button push is counted precisely as once. That one valid push is represented as the signal `inst_vld` . Read the relevant code and use the simulation as your aid, answer the following questions in your lab report.

1. What would be the impact if we use clk_en instead of `clk_en_d` in the statement at line 102 of nexys3.v?
2. Instead of `clk_en <= clk_dv_inc[17]` (line 76), can we instead have `clk_en <= clk_dv[16]` (which effectively creates a 50% duty cycle clk_en)? Why?
3. Include waveform captures that clearly show the timing relationship between `clk_en`, `step_d[1]`, `step_d[0]`, `btnS`, `clk_en_d`, and `inst_vld`.
4. Draw a simple schematic/diagram of the signals above. It should be a translation of the corresponding Verilog code.

## [Implementation] Register File

The sequencer's register file is located in a file called seq_rf.v. It stores the values of the four registers. Take a look at the source code and see if you can understand how it is implemented. Answer the following questions in the lab report.

1. Find the line of code where a register is written a non-zero value. Is this sequential logic or combinatorial logic?

2. Find the lines of code where the register values are read out from the register file. Is this sequential or combinatorial logic? If you were to manually implement the readout logic, what kind of logic elements would you use?
3. Capture a waveform that shows the first time register 0 is written with a non-zero value.
4. Draw a circuit diagram of the register file block. It should be a translation of the corresponding Verilog code.

## Deliverables

Follow the task instructions closely and show your design to the TA at the **Demo Now** points.

When you finish all the demos, submit the following:

1. Project Code: the Xilinx ISE project folder should be cleaned up (*Project > Cleanup Project Files*), zipped and uploaded in the corresponding assignment page on the course website.
2. Lab Report (Electronic Version): the lab report should be uploaded in the corresponding assignment page. **The lab report should be describing the entire sequencer in its final state as a whole.** You may omit the simulation of parts that remain unchanged throughout the project. Lab Report should contain the answers to the questions listed in "Workshop" as a separate section "Workshop Answers".