Anton Paquin
Josh Wildey
EC 551
4/15/18

Lab 2: Peripheral and Processor Integration with I/O

Abstract:

The goal of this lab is to create a processor capable of executing instructions specified by a user. The processor and instructions being used are from the simplified instruction set used for Lab 1. In addition to the simplified instruction set, we implemented matrix multiplication, and a direct interface to ALU operations. The user specified instructions are input with a keyboard connected to the FPGA via a USB-to-PS2 converter. The user input and results can be viewed in a terminal style screen via UART communication with the FPGA.

Software Architecture:

1. **File Hierarchy**
   Figure 1 shows the file hierarchy of the lab and how they relate to each other.

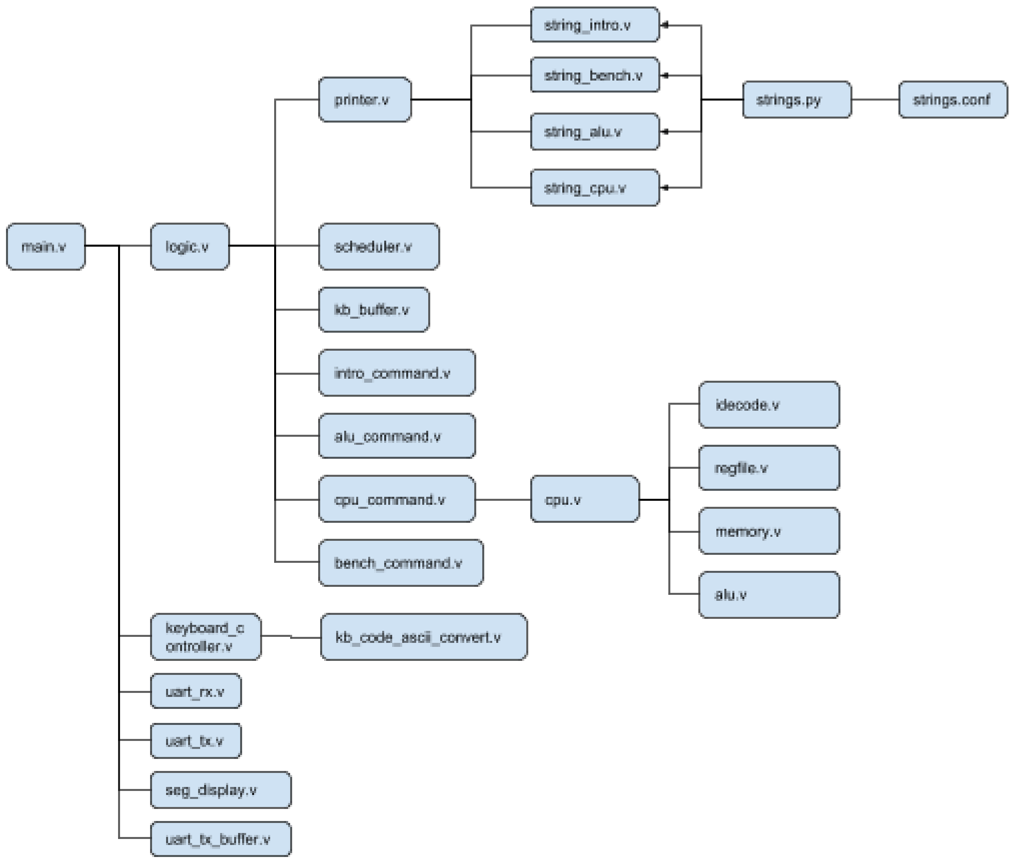

Figure 1: File Hierarchy

2. **Overall Architecture**

   Figure 2 shows the overall module architecture of the lab. There is a top level module basically maps connects modules together and with I/O pins. Details on the sub modules are in the following sections.

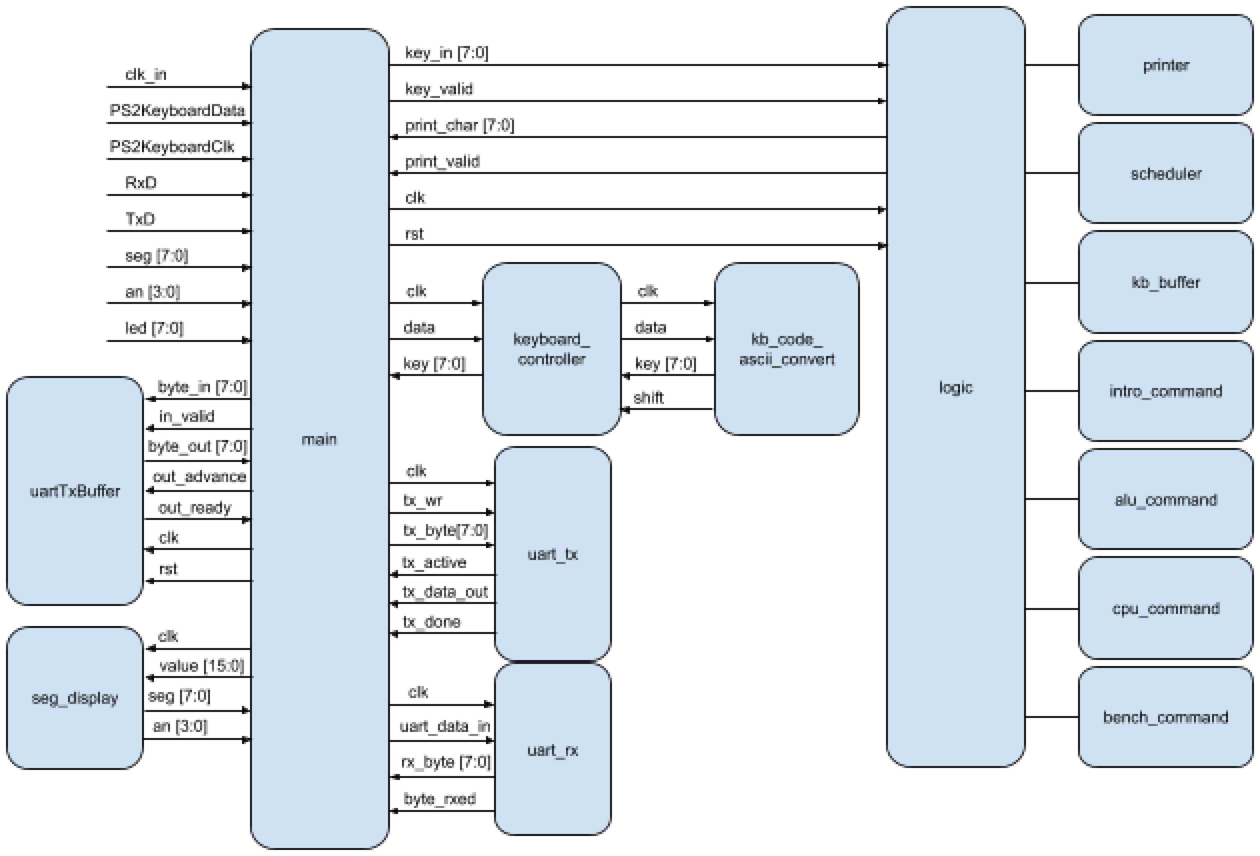

Figure 2: Overall Module Architecture

3. **Printer Sub-Module, UART and 7-Segment**

   Figure 3 shows the printer schematic and relation to the top level module. This module fills up the uartTxBuffer with the what to print to the terminal. What is printed is controlled by the scheduler and the current states of the system. The Logic module forwards the current print character up through main and to the uartTxBuffer which then triggers the uart_tx module to write it to the port. This buffer is a FIFO queue, which allows the computation element to operate at full speed without concern for the UART baud rate. By default, the UART module transmits and receives at 115200 baud but can be configured through a parameter. UART Rx is currently implemented but is unused for the purpose of the lab. The 7-segment display was used for debugging purposes. The left two (2) digits display bytes received from UART and the right two (2) digits display the ASCII code of the key pressed on the keyboard.
   In our print module, we have several preloaded strings -- the “HELLO EC551” string being the longest. These are controlled by a case statement that tracks a string printing counter. These case statements are tedious to write, and there isn’t a good verilog structure that allows them to be condensed, so we’ve split them into their own files (“string_*.v”) and

`included them in the proper position in the printer. These string files are generated by a python script, which reads a config file and produces the proper printing case statement.

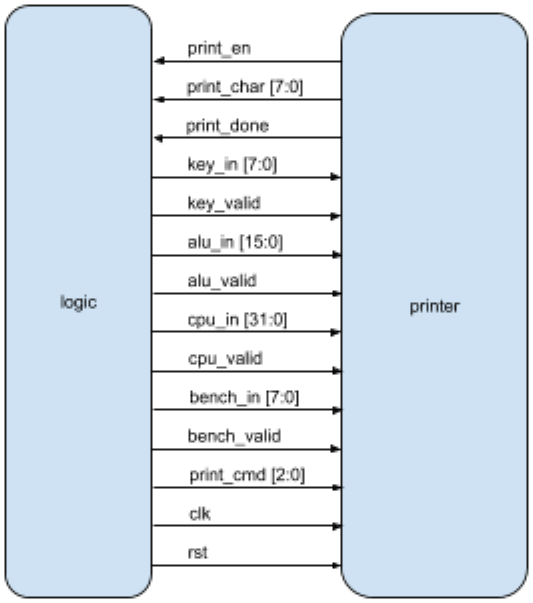

Figure 3: Printer Schematic

4. **Scheduler Sub-Module**

   Figure 4 shows the scheduler schematic and relation to the top level module.

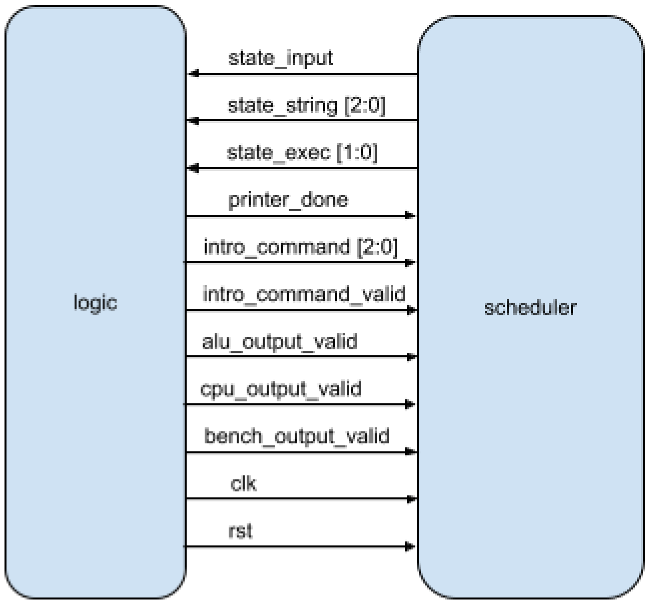

Figure 4: Scheduler Schematic

The scheduler controls the states of the computational component of the FPGA. There are three (3) different states that are controlled here: the “input” state, the “execution” state, and the “output” state.

The input state controls whether or not keyboard input is blocked -- the user should not be able to provide new input while the computational element is executing the last instruction.

The execution state controls which of the three modes the user has selected for operation: CPU mode, ALU mode, or Benchmark mode. In addition, there is an Intro mode which controls execution when the FPGA is first started, before the user selects a mode.
The output mode controls whether the FPGA is currently printing a string, and if it is a constant string or which string it is printing.

5. **Keyboard Buffer Sub-Module**
   Figure 5 shows the scheduler schematic and relation to the top level module. This keyboard buffer is used to queue up single line inputs. Valid key presses from the keyboard controller trigger this buffer to queue up another character. Once a linefeed is received, the buffer is valid and ready for processing. The buffer has room for 16 characters.

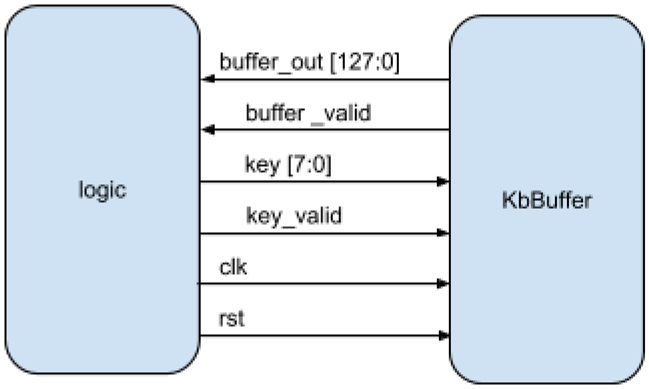

Figure 5: Printer Schematic

For each of the execution modes of the logic stage, there is a “command” module which describes the input and output of that execution mode. Each takes the keyboard line buffer as input, and has a synchronous reset signal. When the module is not active, its reset signal is asserted, effectively shutting down the module. The “exec” state machine in the Scheduler module controls these resets so that only one command module is active at any time.

6. **Intro Command Sub-Module**
   Figure 6 shows the intro command schematic and relation to the top level module. Upon initialization, the scheduler will put the system into an input blocked state so that the system can output the welcome string. It will also set the execution state in the intro mode, which will unblock the input after the welcome string is displayed so the user can specify which operation mode to switch into.

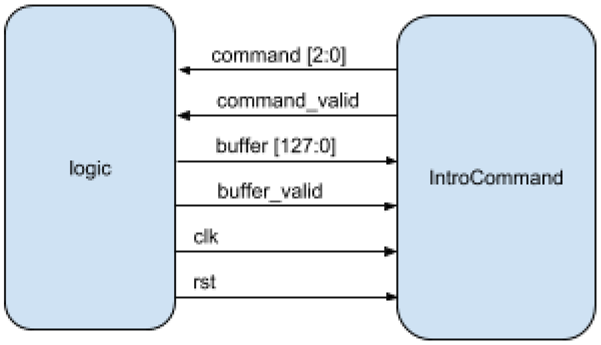

Figure 6: Intro Command Schematic

7. **ALU Command Sub-Module**

Figure 7 shows the intro command schematic and relation to the top level module. This module implements XOR (^), ADD (+), SUB (-) and MUL (#) and is input in the form:

<num1><space><op><space><num2>

The numbers are unsigned 3 bit numbers meaning 0-7. The buffer is formatted with buffer [127:120] for num1, [119:112] for space, [111:104] for operation, [103:96] for space and finally [95:88] for num2.

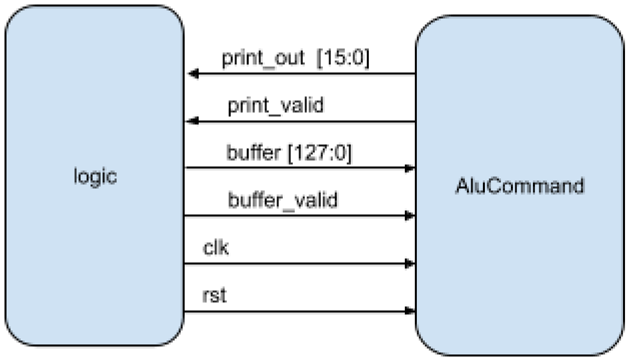

Figure 7: ALU Command Schematic

8. **CPU Command Sub-Module**

Figure 8 shows the CPU command schematic and relation to the top level module.

The CPU command module contains a copy of the CPU built for lab 1, which provides its own synchronous “done” and “rst” signals. Additionally, we’ve inserted the “memhack” write lines, which provide write access to the CPU’s memory. This allows the CPU command module to decode the input buffer into instructions and write them so that they can be executed.

Unfortunately, the block ram modules do not come with their own reset signals. Memory is initialized to zeros, but is not overwritten between program executions. Effectively, this means that when the user inputs a program, they should finish it with a HALT or unexpected results will occur.

Instructions should be input in hexadecimal form, one line per address. Alphabetic characters in the hexadecimal input must be capitalized. Program input and execution start at address 31. When the last instruction has finished executing, the CPU raises “done” and prints the decimal form of the program counter to the output.

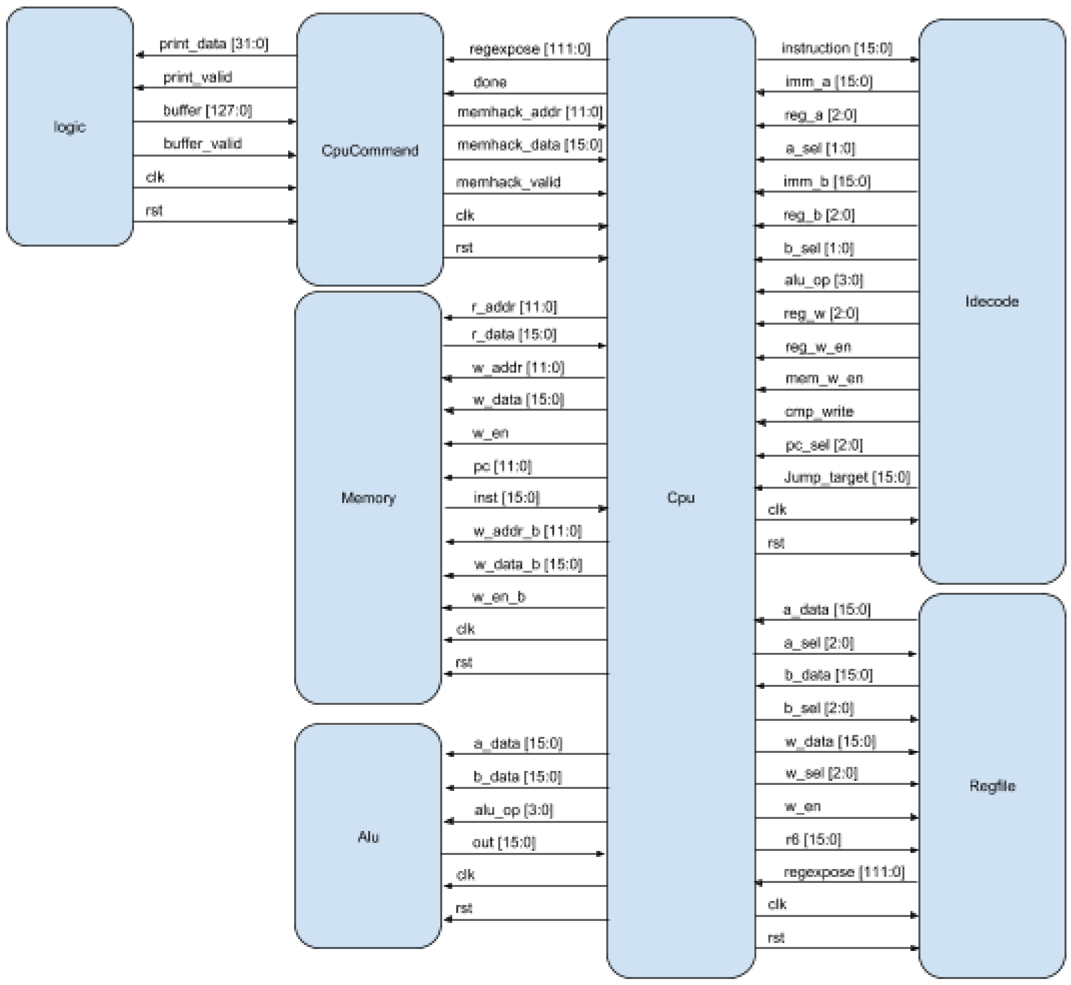

Figure 8: CPU Command Schematic

9. **Bench Command Sub-Module**
   Figure 9 shows the bench command schematic and relation to the top level module.
   The Bench Command module implements 3x3 matrix multiplication. Data is loaded from the buffer in 6 lines, and then printed one character at a time in decimal form to the print_data line. We implemented custom hardware for matrix multiplication in this case. Three 6-bit multipliers are generated, which is not a significant cost. The value of each cell of the result is calculated as it is being printed.

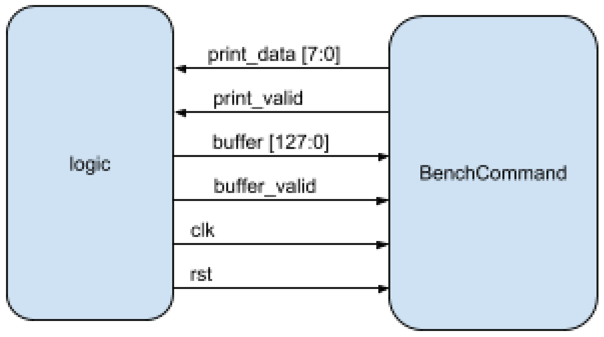

Figure 9: Bench Command Schematic

Modes of Operation

1. Mode I: Enter Instructions

Mode I is entered by typing in ‘I’ and hitting the Enter key. This mode uses the CPU that was developed in Lab 1. As such, when the user enters an instruction and hits enter, each instruction will be stored in memory starting at address 31 until the ‘r’ key is pressed. Each memory location is 16 bits wide and has 4096 address locations. The write to memory is registers are exposed to the cpu_command file to be able to store the instructions from the keyboard input buffer. The 16-bit instruction set used by the CPU can be found in the table below:

Table 1: CPU Simplified Instruction Set

Instruction (16-bit)	OP Code (4-bit)	Operand 1 (6-bit)	Operand 2 (6-bit)
Halt	0000	N/A	
Increment	0001	Reg to inc	N/A
Jump	0010	Address to jump to	
Jump Not Equal	0011	Address to jump to (if cmp not equal)	
Jump Equal	0100	Address to jump to (if cmp equal)	
Add	0101	Rn	Rm
Subtract	0110	Rn	Rm
XOR	0111	Rn	Rm
Compare	1000	Rn	Rm
Move Rn, num	1001	Rn	Number (0-63)
Move Rn, Rm	1010	Rn	Rm
Move [Rn], Rm	1011	Mem contents from addr in Rn	Rm
Move Rn, [Rm]	1100	Rn	Mem contents from addr in Rm
Signed Add	1101	Rn	Rm

Upon entering this mode, the CPU will remain in the reset state while the user inputs data via the keyboard. Once the user enters ‘r’, the CPU will be taken out of reset state and begin running whatever program was entered starting at address 31.

2. **Mode A: Run an ALU Operation**
   The instructions implemented for the ALU Operation mode are XOR (^), ADD (+), SUB (-) and MUL (#) and is input of the form:

   <num1><space><op><space><num2>

   Only unsigned 3-bit numbers are supported for num1 and num2.
   Ex).

   ```
   3 + 4
   7
   5 - 2
   3
   4 # 7
   28
   8 ^ 2
   10
   ```

3. **Mode B: Benchmark Program**
   Matrix multiply was implemented for the benchmark program. It is implemented such that it will multiply two 3x3 matrices. The numbers supported for each location in the input matrices are unsigned 3-bit numbers similarly to the ALU operation. Once in the benchmark operation mode ('B'), 3 numbers need to be entered for each row of each of the two matrices to be multiplied. Once the six rows of the matrices are entered, the module will immediately compute and print the result.
   Ex:

   ```
   1 2 3
   4 5 6
   7 1 2
   7 6 5
   4 3 2
   1 0 1
   018 012 012
   054 039 036
   050 042 036
   ```

4. **Resetting the System:**
   The FPGA has its reset signal tied to the switch closest to the seven segment display. Raising and then lowering this switch will return the user to the "Intro" mode, allowing reselection of an execution mode. CPU memory is not cleared on this reset.