Snake

Team Alpha Wolf Squadron
(also known as Team Tron/Snake)
Beverly Yee
Michelle Simmons
Samuel Farnsworth
Dirk Lamb

***Abstract* - A functional central processing unit (CPU) can be created by programming a field-programmable gate-array (FPGA) using Verilog. The team demonstrated just that by building (in Verilog) the components of a CPU, then show-cased its functionality by recreating Snake. To ensure that the game could be played with the CPU, additional features were added to the unit so that it can take in input from two super Nintendo entertainment system (SNES) controllers and output the corresponding command to a monitor via video graphics array (VGA). The overall CPU operated as a black box referred into this report as Datapath, which takes the outputs of an arithmetic logic unit (ALU), register file, SNES control file, VGA control file, and an on-chip instantiated random-access memory (RAM). It worked in conjunction with a programmer created assembler that also operated as the software of the game.**

I. Introduction

The objective of the project is to create a CPU from scratch in Verilog and create a game or program that could run on that CPU, in addition to adding one or two peripherals. In this case, the team decided to recreate the game Snake, add the ability to have two players play using two SNES controllers, and display the game on a monitor using VGA. Time permitting, extras added to the game could include sound, a start screen, and more than two players.

In its final state, Snake is a two-player game. Both players begin as one segment - one green, one blue - until they eat food. Once a player's snake manages to eat food, of which appears randomly on the screen, their snake grows by one segment. The snake's movements is controlled by a SNES controller and can move in all directions with the exception of backwards. The game ends when a player's snake: a) runs into themselves, b) runs into a wall, or c) runs into their opponent. Upon any one of these conditions, the other player, the player who did not run into an object, is the winner. Then, either player can press START on their controller to reset the game, then either player can press START to begin a new game.

All of these movements, the inputs of players, the appearance of food and snakes - all these components are monitored and taken control of in the assembler, of which is created in Python. In a sense, everything seen on the screen is taken control of by the assembler - almost as though it's the master program with little helpers, of which are: ALU, Datapath, FSM, Memory, Regfile, SNES_Control, VGA, snake memory file, and glyph table memory file. The roles of each individual module will be explained in the following section.

II. Basic Hardware

At the very bottom of the basic CPU organization is the ALU, of which computes all of the CPU's operation. It takes in four inputs and gives out two outputs. Of the four inputs, two of them are 16-bit registers, another is a 16-bit operation code (opcode for short), and the last is a 5-bit flag module.

The two 16-bit registers are responsible of carrying in the data to be operated on. The first register is the destination register while the second register is the source register. The 16-bit opcode is responsible for referring to which register is the destination, which one is source, and what operation is to be operated on them. Encoded within the opcode itself, the first 4 bits are the HIGH opcode values. The next 4 bits are the pointers to which destination register to use. From bits [7:4] are the LOW opcode values and [3:0] refers to the source register. Note that not all instructions are register-to-register - there are instructions that deals with immediate values and other instructions that shifts the bits of the value in register. Table 1 below shows the list of opcodes the team used in their ALU.

	Opcode HIGH	RDEST	Opcode LOW/ImmHi	RSRC/ImmLo
ADDI	0101	RDEST	ImmHi	ImmLo
ADDUI	0110	RDEST	ImmHi	ImmLo
ADDCI	0111	RDEST	ImmHi	ImmLo
MULI	1110	RDEST	ImmHi	ImmLo
SUBI	1001	RDEST	ImmHi	ImmLo
SUBCI	1010	RDEST	ImmHi	ImmLo
CMPI	1011	RDEST	ImmHi	ImmLo
ANDI	0001	RDEST	ImmHi	ImmLo
ORI	0010	RDEST	ImmHi	ImmLo
XORI	0011	RDEST	ImmHi	ImmLo
MOVI	1101	RDEST	ImmHi	ImmLo
SHIFT	1000	RDEST	ImmHi	ImmLo
LUI	1111	RDEST	ImmHi	ImmLo
ADD	0000	RDEST	0101	RSRC
ADDU	0000	RDEST	0110	RSRC
ADDC	0000	RDEST	0111	RSRC
MUL	0000	RDEST	1110	RSRC
SUB	0000	RDEST	1001	RSRC
SUBC	0000	RDEST	1010	RSRC

CMP	0000	RDEST	1011	RSRC
AND	0000	RDEST	0001	RSRC
OR	0000	RDEST	0010	RSRC
XOR	0000	RDEST	0011	RSRC
MOVE	0000	RDEST	1101	RSRC
LSH	1000	RDEST	0100	RAMOUNT
LLSHI	1000	RDEST	0000	RAMOUNT
LRSHI	1000	RDEST	0001	RAMOUNT
ASH	1000	RDEST	0110	RAMOUNT
ALSHI	1000	RDEST	0010	RAMOUNT
ARSHI	1000	RDEST	0011	RAMOUNT

Table 1 - List of opcodes used by the team. This list is based of the CR-16 ISA architecture, but modified a little to suit the team's needs better.

The design of the ALU is paired with the design of a register bank, of which is a module wrapper consisting of 16 16-bit registers. Depending on the register enable signal and the register number, that specified register will be read from/ written into.

Following the ALU and RegBank designs came the design and instantiation of Memory. Using a template provided by the Quartus software, the memory used is a True-Dual Port Memory RAM, which had two inputs and outputs to read and write out to simultaneously, if needed (Figure 1 & 2). To read from memory is simple - a text file with either binary or hex commands is all that is needed along with the Verilog command $readmemh("filepath.txt", ram) or $readmemb("filepath.txt", ram).

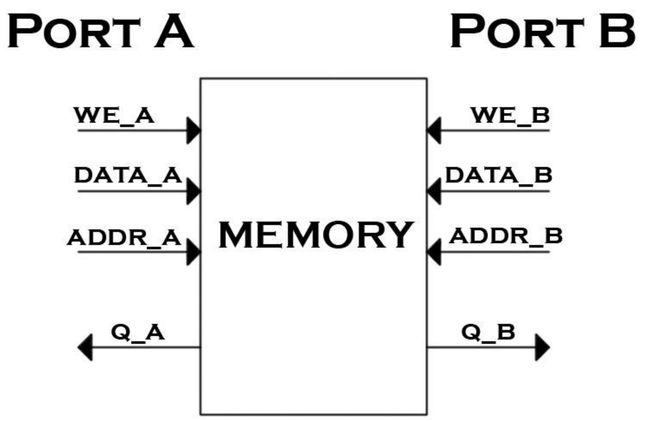

Figure 1 - A simple diagram showing the memory the team used.

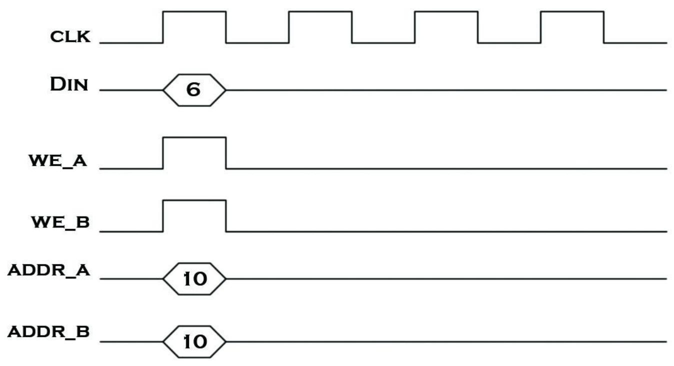

Figure 2 - A timing diagram demonstrating the condition where both write enables have been turned on and the addresses that the memory wants to write to are the same from both ports.

With this, the basic of basic components of the CPU are ready. To put them all together, two additional modules, FSM and Datapath, were created. To start, the FSM is a 17-state finite-state machine that determines what part of the instruction cycle the CPU is in. The individual states are as follows:

1. RESET - sets everything to 0 when RESET = 1. Otherwise, head to FETCH_1
2. FETCH_1 - flips pc enable (pc_en) to 1, then switches to FETCH_2
3. FETCH_2 - sets pc_en = 0, and figures out what kind of instruction to do
4. R_TYPE - if the instruction is a register-to-register type of instruction, utilize the ALU and perform the corresponding operation. Go back to FETCH_1 after this state.
5. STORE_1 - if the instruction is a store-type instruction, turn off pc_sel (PC select), which determines whether the output of the program counter (pc_sel = 1) or the output of the registers (pc_sel = 0) is needed. Then turn on Memory to write into Port A, which is the port given priority. Afterwards, head to STORE_2.
6. STORE_2 - turn on pc_sel, turn off writing to Port A, go back to FETCH_1.
7. LOAD_1 - if the instruction is a load-type instruction, turn off pc_sel, load the address, and get the value out. Go to LOAD_2.
8. LOAD_2 - turn off alu_sel, the MUX that determines whether the bus from the ALU is read (alu_sel = 1) or the data from Port A in memory (alu_sel = 0). Go back to FETCH_1.
9. JUMP_1 - if the instruction is a jump-type instruction, depending on what the parameters are in the instruction set, do one of the following operations, load it into pc_ld, then jump to JUMP_2.
    a. EQUAL
    b. NOT_EQ
    c. GREAT_EQ
    d. CARRY_SET
    e. CARRY_CL
    f. HIGHER
    g. LOW_SAME

    h. LOWER
    i. HIGH_SAME
    j. GREATER
    k. LESS_EQ
    l. FLAG_SET
    m. FLAG_CL
    n. LESS
    o. UNCOND
    p. NO_JUMP

10. JUMP_2 - sets pc_ld (PC load) and pc_en (PC enable) to 0 and go back to FETCH_1.
11. JAL_1 - if the instruction is a jump-and-link-type instruction, turn pc_ld and pc_en on, save the old instruction, save the new instructions, and go to JAL_2.
12. JAL_2 - turn off pc_ld and pc_en and set the opcode and registers to the new instruction. Go to JAL_3.
13. JAL_3 - load the old instruction, go back to R_TYPE to execute the new instruction.
14. SNES_1 - if the instruction is a SNES-type instruction, set the instruction based on original instruction data and the SNES data. Go to SNES_2.
15. SNES_2 - sets the opcode and registers to the instructions. Go to SNES_3.
16. SNES_3 - go to R_TYPE to execute the instruction set in the previous state.
17. STOP - stops the state at the very end. Similar to RESET.

Refer to FSM.v to see the flow of the states described. Below in Figure 3 is a visual representation of the state-machine.

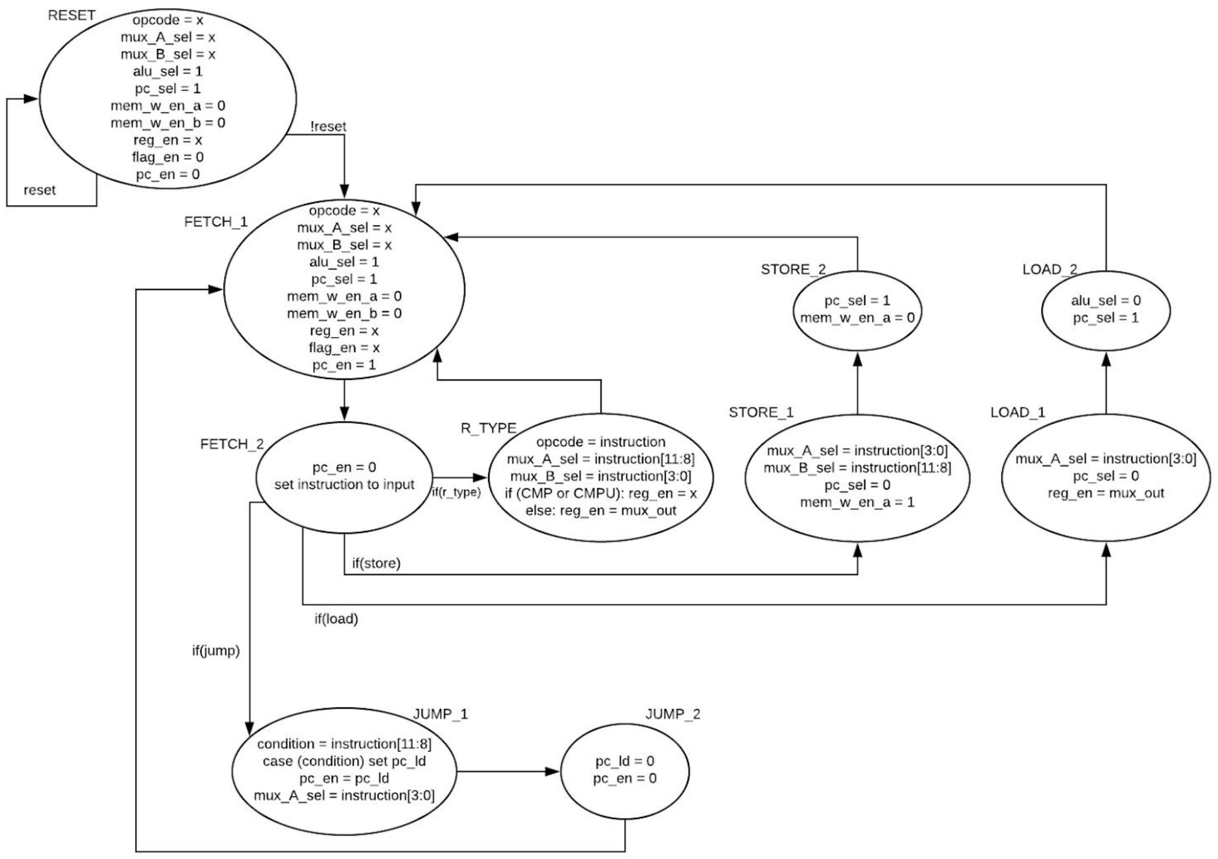

Figure 3 - A flow chart of the FSM described above

Tying this FSM to the other modules is the Datapath, which will also eventually become the big module that ties everything into the assembler to get the game working. At this stage, the Datapath only worked with the RegBank, the ALU, two muxes referred to as RegMuxA and RegMuxB, a ProgramCounter, Memory, and Flags module.

The RegMuxA and RegMuxB are modules created to act like muxes, that determine how the information is passed along the datapath from the registers to the ALU - whether it's from the RegBank or if it's from a different source.

The ProgramCounter is as it is named - keeps track of where the CPU is in its instruction set. It's enabled when there is data that needs to be passed through (called the pc load, or pc_ld).

The only inputs Datapath takes at this stage is the clock from the Altera Cyclone V FPGA and a reset signal. The reset signal of the Cyclone V is an active low, which means it's constantly on high until told otherwise. Due to this, many of the reset signals have to inverted when being passed through to the other modules.

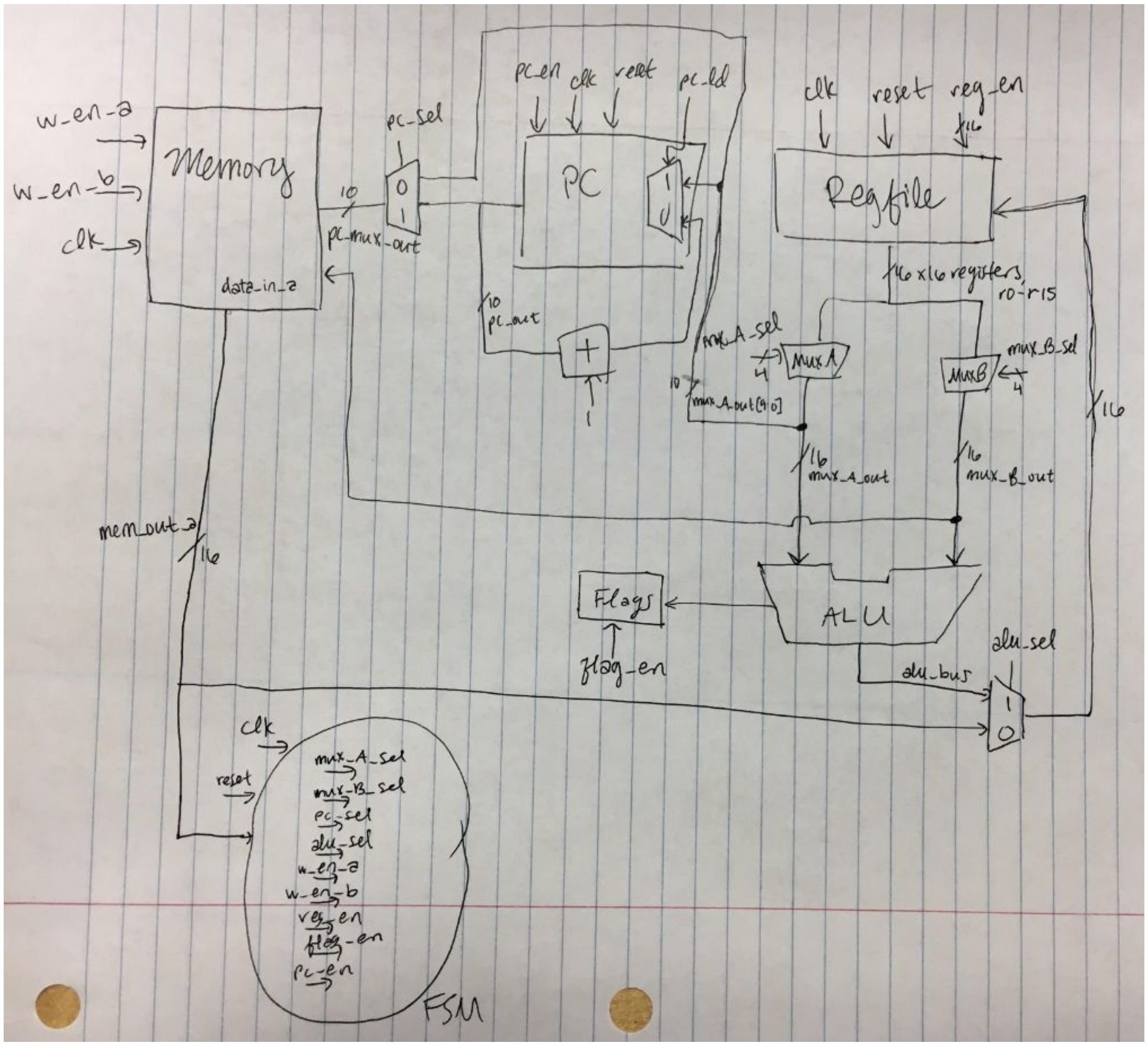

Figure 4 - A flow chart of how the initial Datapath module should look.

At the time of creating the game for the (mostly) completed CPU, Datapath was expanded to include SNES controls and the VGA control module, both of which will be further explained in the next section. The complete flow chart with all the peripherals and the modules put together can be referred to in the image below.

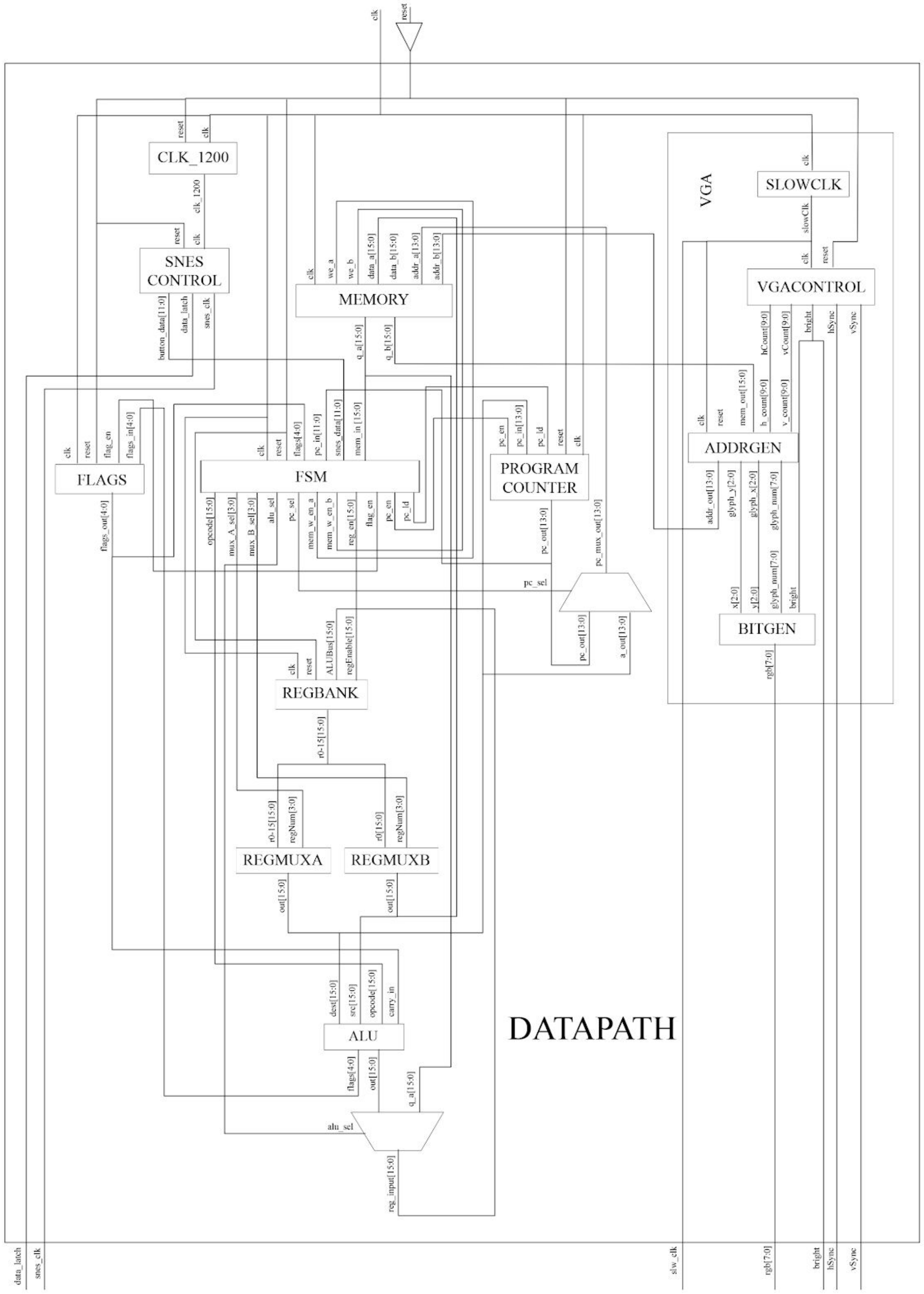
clk
reset
CLK_1200
clk_1200
SNES CONTROL
button_data[11:0]
data_latch
snes_clk
MEMORY
we_a
we_b
data_a[15:0]
data_b[15:0]
addr_a[13:0]
addr_b[13:0]
q_a[15:0]
q_b[15:0]
FLAGS
flag_en
flags_in[4:0]
flags_out[4:0]
FSM
flags[4:0]
pc_in[11:0]
snes_data[11:0]
mem_in [15:0]
opcode[15:0]
mux_A_sel[3:0]
mux_B_sel[3:0]
alu_sel
pc_sel
mem_w_en_a
mem_w_en_b
reg_en[15:0]
flag_en
pc_en
pc_ld
PROGRAM COUNTER
pc_en
pc_in[13:0]
pc_ld
pc_out[13:0]
pc_mux_out[13:0]
pc_sel
pc_out[13:0]
a_out[13:0]
REGBANK
ALUBus[15:0]
regEnable[15:0]
r0-15[15:0]
REGMUXA
REGMUXB
r0-15[15:0]
regNum[3:0]
out[15:0]
ALU
dest[15:0]
src[15:0]
opcode[15:0]
carry_in
flags[4:0]
out[15:0]
q_a[15:0]
alu_sel
reg_input[15:0]
VGA
SLOWCLK
slowClk
VGACONTROL
hCount[9:0]
vCount[9:0]
bright
hSync
vSync
ADDRGEN
mem_out[15:0]
h_count[9:0]
v_count[9:0]
addr_out[13:0]
glyph_y[2:0]
glyph_x[2:0]
glyph_num[7:0]
BITGEN
x[2:0]
y[2:0]
glyph_num[7:0]
bright
rgb[7:0]
DATAPATH
data_latch
snes_clk
slw_clk
rgb[7:0]
bright
hSync
vSync

Figure 5 - The completed flow chart of Datapath. Email bevyujw@gmail.com if the original, full-size copy is needed for clarity.

III. Extensions

Peripherals included in this CPU are two SNES controllers and a monitor, visuals outputted by a VGA cord. The memory used for the program is provided in the FPGA used to tie everything together and display it.

The VGA started with simple Verilog code and is further broken down into three modules - VGAControl, BitGen, and AddrGen. It takes in a clock and reset input as well as memory from Port B of the Memory module and outputs the hsync, vsync, bright, rgb, and slowClk signals needed to display the visuals in addition to the address of the glyph to display. The FPGA has a 50MHz clock that is fed into the other modules. For the modules in VGA, it's slowed down by half (25MHz) so the display is shown properly.

To ensure that memory is not entirely used up by drawing individual pixels, the screen is divided into 8x8 pixels referred to as glyphs. These glyphs are stored in a separate text file called GlyphTable that BitGen refers to and takes in the addresses for to find and draw.

VGAControl deals with the timing of the display. It takes in the slow clock from the top module VGA and determines when it's the right time to turn on the beam to start drawing on the monitor and when it's time to fire a signal to move down or move back to the top left of the screen. This never changes as the other modules are added.

AddrGen is a module that determines where the beam is on the screen, and if it's within the active video part of the screen, it outputs the address based on the frame buffer in memory and the location. At the same time, it saves the location for BitGen to use in the GlyphTable. It also saves the number of the glyph gotten from the memory read from Port B of the Memory. Since memory data is 16 bits, but pixel data is only 8 bits, two pixels are stored in each memory data. Depending on a variable, either the first pixel, the top 8 bits of the data, is read or the bottom 8 bits.

BitGen is the module that draws the glyphs onto the screen. It reads in the information AddrGen gives out, aside from the address to the frame buffer, and looks up the corresponding glyph in GlyphTable. Then, at that location, from the read-in address, the glyph is printed. The module automatically sets the background to black when a glyph isn't being drawn.

There is about 46 total glyphs used - the entire alphabet in caps, numbers, a dash, exclamation point, wall, food, and the two snakes and their head, tail, and body glyphs. Not all of them were used - just enough to output PLAYA, SNAKE, WIN, the wall, and the snakes. In memory, the walls and the words PLAYA and SNAKE are hardcoded in. Initially, the food is also hardcoded, but as soon as they are eaten, random ones pop up. The word WIN only appears when the game is won.

The two SNES controllers are read by inputs from the pins into the FPGA board. Based on those inputs, the software executes the according software. The module takes in a clock, as the controllers also operate on a different clock cycle compared to the one on the FPGA and the other modules. It's outputs are used in FSM, which looks up what to do if an input is received by the CPU/Assembler.

One SNES_Control module is needed per player, so if there are more players to be added, simply add another line in Datapath for that player.

An interesting feature of the team's VGA is that it's a standalone module. All it does it output the address of the glyph needed to be drawn, receive that glyph, then draw the glyph. Everything about the movement of the snakes and the appearance of additional food is taken care of the software. As mentioned previously, the software is what really manages the entire game with the help of the CPU.

Another little funky noticed by the Teaching Assistant is that the team's VGA is doubled the size than it should be. It's scaled somewhere, but the bug cannot be found, nor was it going to be fixed as it was the perfect size to display the glyphs. On the other hand, however, it causes some of the character glyphs to appear funkily, as though they were 'half' a pixel off. As a result, only certain characters were displayed (thus why player is PLAYA instead), although it adds spunk to the game itself.

IV. Software

Dirk built an assembler that could translate simple, human-readable assembly instructions into machine code that the CPU could read and execute. It was written in Python, and it accepts file names as inputs. It scans through the input file line by line, then matches the line against a fixed set of possible inputs. In addition to the standard CR16 instructions, which are implemented by the CPU, the assembler implements a few macros, which augment the capabilities of and, for the most part, replace the need for calling directly JMP, LUI, JAL, and MOVI. The macros have the following structures and functionalities:

```
JMP_REL offset reg cond
   - jumps on {cond} to {current address + offset}, using {reg}
   - NOTE: will overwrite {reg}
   - assembles into MOVI, LUI, JMP
   Examples:
   JMP_REL 10 r12 UC
   JMP_REL -7 r4 EQ
JMP_IMM addr reg con
   - same as JMP_REL, except using an absolute address instead of an offset
   - addr must fit in two bytes, and jump to a valid code location
JAL_IMM addr reg [lnk]
   - same as JMP_IMM, but JAL
   - lnk defaults to r15 if none specified
   - addr must fit in two bytes, and jump to a valid code location
MOV_IMM val reg
   - similar to MOVI, but allows full byte values
   - equivalent to
      MOVI val(lower byte) reg
      LUI val(upper byte) reg
   - val must fit in two bytes
```

The assembler also implements print_addr, which prints the line number and memory address at the given line to the console. This instruction has no effect on the Snake program itself; it simply allows the user to specify an address for JAL_IMM with ease. The assembler also performs some basic error-checking to make locating bugs in the assembly code easier. It can handle single-line comments and block comments.

The assembler has two possible output formats--either in a format readable by $readmemh, or formatted in ASCII hex, which can be converted into a memory initialization file (MIF) using srec_cat. srec_cat is a program made to convert various memory initialization files into different formats. In our case, we used it to convert ASCII hex into .MIF, which allowed us to skip the analysis part of compilation in Quartus, which saved us 5-10 minutes for every compilation.

Our memory was block RAM, large enough to hold 2^14 addresses, about 32 kb. Our Snake program uses a frame buffer to specify which glyph to write and to what location such that it appears where we want on the screen. We designated memory address 0x3000 as the beginning of the frame buffer and 0x3960 as the end, where 0x3000 corresponds to the top left corner of the screen and 0x3960 corresponds to the bottom right corner. These addresses hold two bytes, where each byte holds a 6-bit glyph number and a 2-bit direction. The default glyph number, 000000, corresponds to a black glyph. While there are several glyphs, there are only six “game” glyphs: wall, food, snake0 head, snake1 head, snake0 body, and snake1 body. Each of the “game” glyphs’ first three bits are 1. None of the other glyphs in the glyph table begin with three 1’s. This is how we easily distinguished game glyphs from empty space.

The walls of the game and the character glyphs at the top of the screen were static; they were stored in a frame buffer copy, and at the start of every game, our program put every glyph in the frame buffer copy into the actual frame buffer, so that each game had a clean start.

We designated some registers to hold the same type of information throughout the entire program.

Register Conventions:
NOTE: even regs SNES_0, odd regs SNES_1
r0: SNES_0
r1: SNES_1

```
 0: B
 1: Y
 2: Select
 3: Start
 4: Up
 5: Down
 6: Left
 7: Right
 8: A
 9: X
10: L
11: R
```

```
r2: snake_0_head
r3: snake_1_head
   [14:13] direction
      NOTE: bit 14 indicates vertical movement (0) vs horizontal movement (1)
      00: up
      01: down
      10: left
      11: right
   [12:1] offset from frame_buffer to head_location
   [0] head_byte
      -keeps track of whether the head is in the high or low byte of the word in memory
      0: high_byte
      1: low_byte

r4: snake_0_tail
r5: snake_1_tail
   [12:0] offset from board_start

r13: Overlap register
   [3:2] tail_dir
   [1:0] snake_0 overlap
      0: bad
      1: food
r14: JMP_IMM/JMP_REL
r15: JAL/Return
```

The rest of the registers were available for any data at any time. If we ran out of registers to use, we stored a value in one register to memory and then loaded it when we needed that old data again, so that register could be used for new data in the meantime.

At the end of the program, we included some “while loops” with a counter that incremented at each iteration of the loop, and the loop ended when the counter reached some large value. These loops functioned as “waits” in our program, so that the snake would move slowly enough for a player to control its movements in a manageable way.

Each snake moves by keeping track of where its head is by keeping the address of the head location in the snake’s designated head register. Each frame, the program checks for input from the SNES controller that would change the movement for the snake (for instance, an “up” direction from the controller while the snake is moving left or right should cause the snake to move up; however, a “left” direction from the controller while the snake is moving left or right should cause the snake to continue moving in its current direction). The head changes its location to the memory address in the proper direction, writes a head glyph to the frame buffer in that new location, and updates its previous memory address to a body glyph with the direction that now points to the head.

The tail updates in a similar way to the head. Since each body glyph points to the next glyph in the snake until the head, the tail can read the direction of the glyph in its current location, overwrite its current location to a blank glyph, and then update its location in memory according to the direction it read. If a snake's head overlaps a food glyph, the tail is not updated and the snake's length grows by one glyph.

Each frame, after the snake's head location has been updated, we check for overlap by checking that head location in the frame buffer. If that location in the frame buffer is already holding a game glyph, the snake has overlapped either a body glyph, a wall glyph, or a food glyph. If it has overlapped a food glyph, the game continues playing and that snake skips its tail updating process. Otherwise, the game ends and displays the winner (whichever snake did NOT overlap with a non-food game glyph is the winner).

For more info, we have detailed comments in snake.man and throughout snake.as.

V. Lessons Learned

When attempting to instantiate memory, the team was advised to combine the two always blocks into one as it would be easier to check for instances if both ports wrote into the same address. However, the team quickly learned that doing so, no matter which way the code was written, there was no way the code would function the same way if the two read and writes were placed within the same always block. The team resorted to leaving the ports into two separate always blocks with the thought to go back and create a wrapper for the condition later. It is discovered later that it was not needed and the second write port was not even needed.

VGA timing is as simple as it sounds. All it needs is to count the clock cycles or how many lines it has gotten through and keep track of those to know when to fire the hsync or vsync pulses. That's all it needs to do. Additional modules will be needed to ensure the pixels are drawn and in the way they need to be, when they need to be.

VI. Individual Contributions

For the entirety of the project, the work was split between two teams of Dirk and Michelle, and Sam and Beverly. Up until the latter half of the project where the programming and the VGA of the game was left to do, each team switched off between working on an essential part of the CPU and testing that code once it was completed. Sam and Beverly started off with creating the ALU while Dirk and Michelle tested it - from there, the two teams switched off.

After Thanksgiving Break, the team started working on the game part of the project. Beverly and Sam were in charge of the VGA; Dirk built an assembler, Michelle made a module for the SNES controllers, and Dirk and Michelle wrote the software for the game together. Toward the end, VGA fell on Beverly while Sam worked on getting a glyph table ready.

VII. Conclusion

With some setbacks, the overall project was completed with results better than originally expected. Not only is the game visually appealing, it works exactly as intended. There are a few features that differ or not added from the original game, such as scoring, but overall, the game is able to restart, play with two players, collect food, and end when the snake either hits the wall or hits itself.

There are several improvements that could be made to the project, one of which was just mentioned previously - adding scores. Currently, the game ends when a snake runs into the wall, itself, or into another player and whoever was the one who didn't run into something is the winner. That can be changed so that the software keeps track of the score and/or the length of the snake and base the winner off of that.

Another improvement can be to add sounds, such as background music or the sound of eating whenever a snake eats the food. End music/sound when a snake runs into an object. A start screen could be added to introduce the game, allow the player to choose what kind of color they want their snake to be, and enable keyboard input to allow the player to set their name as well. Maybe increase memory size to keep track of the highest scorers and display them on a separate winner/high score screen (similar to how arcade games display high scores).

At the end of the game, an end game screen could be added to show off the winner that would then move to the high score screen or back to the start screen depending on the input the user gives the software.

Overall, the game as it is is great. Though the game can be improved in many ways, most of them are merely functionality of the game and mostly aesthetics. In terms of the hardware, optimization may be needed as well as cleaning up code that is unneeded, but otherwise, it is as it should be.

VIII. References

[1] Alphabet glyphs - https://ece320web.groups.et.byu.net/labs/VGATextGeneration/VGA_Terminal.html

[2] Beginning of VGA - https://timetoexplore.net/blog/arty-fpga-vga-verilog-01 & http://www.eng.ucy.ac.cy/theocharides/Courses/ECE664/VGA.pdf