Piano Pong

Team 3'b101

Alec Adair, Benjamin Antczak, Aaron Herrman

Piano Pong

Team 3'b101: Alec Adair, Benjamin Antczak, Aaron Herrmann

Abstract

This report details the design and goals of the project that our team worked on this semester. The goal for our team was to create a game using a unique method of input as a control peripheral. Using a piano keyboard as our input the game we made was a throwback to the Atari's Pong with keyboard presses acting as the receiving and sending signal for the ball. Along with the game our team created several Verilog modules and created an assembler from scratch. Important Verilog modules created included an ALU, a register file, block-RAM memory, a control, and VGA output.

I. Introduction

The idea for Piano Pong developed as our team accomplished other tasks throughout the semester. In the beginning of the class our idea was to simply make a MIDI visualizer, which turned out to not be as complicated as we thought. From there we decided to make the idea more complicated by making a matching and memory game where the keyboard would send a visual representation of the note played and that note would have to be matched and a new note added. This process would go on with matched notes being bounced back and forth until they were missed. In the end the memory portion of the game was dropped and the matching part of the game became the central idea.

The rest of this report will detail the various design specifications that our group created. First it will discuss the ALU and register file, along with our basic data path and supported operations. The next section details the choice of block-RAM memory and the way it is separated. After which the control module will be detailed, along with how all of the other modules will be tied into, and controlled by, this module. The last part of the report will be about the various peripherals which were used in the project, this includes the assembler, the MIDI input, the VGA and software the team utilized.

II. Design

The design of our game consists of the ALU and processor data-path, memory, the controller, the assembler, and I/O (MIDI and VGA).

A. ALU and Processor Data-path

The first two labs entail creating a data-path by combining an Arithmetic Logic Unit ("ALU") with a register file. The ALU reads data stored in the register file or an immediate value and performs an operation on the data and puts the output onto a bus that the register file reads from. The simple data-path is complete once the output is written back into one of the registers. The final processor will operate by instructions that include an ALU operation and identify either two registers or one register and an immediate value.

Our simple ALU takes two 16 bit inputs, A[15:0] and B[15:0], and has one 16 bit output RESULT. Additionally, there is an 8-bit input which controls the operation performed on A and B, and a series of five output condition flags. Our ALU is a combinational circuit that reacts to any change of inputs (ALU OP or either of the inputs). Since the ALU is a combinational circuit

it is very important that the circuitry surrounding and controlling the ALU is synchronized in a way that the inputs to the ALU are not changing before the surrounding circuitry is ready for a change in output.

Our ALU only perform ten different operations. ADD, ADDU, SUB, CMP, CMPU, AND, OR, XOR, LSH, and ASH. We have chosen to use an 8 bit opcode since 8 is a power of two, and we have plenty of room to add more codes and operations to our ALU as we need them.

The ADD operation adds the sources and treats them as signed integers and can set the Overflow flag to high. ADDU does the same as ADD but instead treating the sources as two's complement signed integers, ADD can also only set the carry bit to high, not the overflow flag. SUB subtracts register B from A (A - B) and can also set the overflow flag to high. CMP compares A and B as two's complement integers and outputs a 0x1 signal if A is less than B and 0x0 otherwise.. The only flags CMP may set are the zero or negative flag. CMPU acts the same as CMP except that CMP interprets A and B as signed two's comp. integers. AND, OR, and XOR are all bitwise logical operations that correspond to their title. These three operations can only set the ZERO flag. LSH is a logical shift that shifts left 1 bit if B is positive and bit right other wise. RSH is an arithmetic shift that arithmetically shifts one bit left or right based again on if B is positive or not respectively.

Our ALU's output flags are CARRY, LOW, NEGATIVE, ZERO, AND GENERAL. We may add more flags depending on what we see fit. The CARRY flag corresponds to a carry out or carry-in for an arithmetic operation. The LOW flag is set to high when the A input is less than the B input when they are interpreted as unsigned numbers. For now the LOW flag does not get set high for two's complement numbers, but we may implement this in the future. The NEGATIVE bit is set by the comparison or subtraction operations, when interpreting the inputs as two's complement integers and Source A is less than Source B, the Negative flag is set. ZERO is set at any time that the result of the ALU operation is equal to 0. We may also include setting the ZERO bit high automatically when both of the source operands are equal value.

We are including a flags register that will retain the flags of the last ALU operation that will be useful in implementing jumps, and other control flow instructions. Our processor contains 16 registers that are each 16 bits wide. The ALU's output is constantly being fed into the Register File's input, but only stored to one enabled register per cycle. The register file is a synchronous circuit with two sixteen bit wide inputs, one for data in, and for register enable. Its primary use is to temporarily hold values that are computed or loaded from memory (cache or other memories) for quick computation. Our ALU does not have access to any memories directly and data loaded from memory will probably have to be put onto the ALU output bus and interfaced through the Register File to go through the ALU.

Since our implementation of the processor is a single cycle implementation, every cycle the ALU does a computation and a data value is written to a register among other things in the processor (instruction fetches, memory loads and writes etc.). We chose to implement the enable signal as a five bit input that is decoded and turns one of the registers enable signals high. We chose five bits because we may add a couple extra registers for certain tasks, and so there is also a way to enable no registers for writing. Currently our registers are indexed 0 through 15. With only four bits a register would always be enabled, but with five there is room for no register to be enabled. To enable a register a five bit binary number corresponding to the index of the register you to be used is applied to the enable signal (input to our register file). The value is decoded and the appropriate register is enabled for writing. Register Data Out.

Each of the sixteen Registers in the Reg. File are constantly outputting a binary value. This

binary value is split between the A and B inputs to the ALU. This means that for every register in the register file there are 32 wires being output. This is so that every clock cycle two values can be feed into the ALU instead of just one. In between the ALU and the register file we have tri-state buffers to control what value gets fed to the A and B inputs of the ALU. The tri-state buffers are sixteen-bit wide tri-state buffers. For each register there is an A and B buffer to admit or impede the signal coming from the register into the ALU. The circuitry surrounding our data-path must follow our protocol for enabling and controlling these buffers.

Each of our sixteen buffers has a one bit enable input. If the input is high the buffer will act as a short circuit, and if the input is low the buffer will act as an open circuit. To let a value get from the data output of a register to get through to the ALU it must pass through a buffer, so that desired buffer must be enabled. It is important that the surrounding circuitry controlling these buffers only lets one "open up" for each ALU input at a time. If more than one opens up this will cause multiple drivers on one of the ALU inputs and the circuit will not work not function.

Our module containing the register file and the ALU takes two five-bit inputs for buffer enables, one for the A input of the ALU and the other for the B input of the ALU. The values are input as a five-bit binary numbers and that directly correspond to the output of the indexed register. The inputs are interpreted as unsigned integers and then decoded to open up the appropriate buffer to let the value of the corresponding register through to the ALU's A or B input.

Before each input to the ALU there is a bus that ties together the outputs of the buffers for each of the sixteen registers with the A and B inputs for the ALU. For bus A the Buffer A control signal will determine what value is put onto the bus and fed to the input of the ALU. The Bus for the B input ties all of the registers together and the ALU input B as well as a value for an immediate. The immediate buffer is its own signal that is controlled by one bit.

In addition to these busses there is a bus tied to the output of the ALU and the inputs for the Register read data. This bus always contains the result of the most recent ALU operation and is fed to the registers. Depending on the Register File's enable signal this value will be written to a register.

Our data-path provides a way of performing arithmetic or logical operations on immediate values. The immediate value can be interpreted as signed or unsigned depending on the ALU operation being performed. When doing a computation with an immediate value the immediate is loaded onto the B input of the ALU through the bus connected to it. There is a buffer that is sixteen bits wide for this immediate value with its own one-bit control signal. When this bit is set to high all other buffers connected to this bus are automatically set to their high impedance state and the immediate is loaded onto the bus and fed to the ALU's B input.

B. Memory

The memory interface must be written in a way that a future control unit can be implemented to take instructions such as fetch, decode, execute, and write-back, and access the Block-RAM using the interface. Our CPU architecture is centered around a 16-bit word length which would correspond to 2 raised to the power of 16 addresses, but this is too large for the on board RAM. We chose to use a 10-bit address which corresponds to 1024 byte(64 words) addresses which is also 1 block of the on board RAM. We chose to implement a dual-port write first memory. We do not plan on doing writes on the second port (Port B is read only for glyphs).

While part of our team implemented the memory module, the other part worked on designing and implementing a finite state machine to test the memory reads and writes. We took our FSM

from Lab 2, and expand it to use the on-board memory. This FSM uses the Fibonacci sequence to store future terms based on previous terms in memory. After implementing the FSM, we wrote a test-bench to test and simulate the machine. After tweaking the FSM we got it simulating as expected, but then when programming the FPGA board we had immense struggles getting the expected results due to memory calculations that we still have not fully figured out, but have gotten the memory to behave as expected using hard coded values.

We chose to expand on the memory example given on the class website and keep the write-first memory design. This example uses the fact that Xilinx supports the use of 2-D arrays to map to memory. In our memory module we use a row size of 16 corresponding to the word length of our CPU, and a column length of 10 corresponding to the size of the address in the memory. The memory module takes a 1-bit CLK as input for synchronization, a 16-bit Data-In which is the data to be written to memory, 1 bit Enable A and Enable B signals for enabling either port, and 1-bit write enable inputs for A and B to enable either port for writing. We plan to never enable Port B for writing. There are also 10-bit address inputs for A and B to tell the memory which address is being accessed, as well as 16 bit data outputs for each port, which are both connected to buffers and wired to the main bus of the ALU and Register File.

The memory has been written in a write-first fashion. This entails that the data appearing at the data-in input of the memory will be immediately written and sent to the output of the memory. This in effect means that the data-in appears immediately on the positive edge of the clock cycle at the output of the data-out output of the respective ports when both of the different enable signals are high for each port. To initialize our memory we use the $readmemh commands provided by Xilinx.

Once the memory module was implemented, we connected it to our ALU and Register File. The outputs of both ports A and B are connected to sixteen bit buffers and then connected to the main bus of the Register File and ALU. The data in of Port A was connected to the ALU Result. The enable, write enable, address, and clock signals are left to the control. Once this was done, a state machine to control all the buffer signals, and memory reads and writes was written. This state machine was derived from the state machine to run our register file with the ALU. The main difference is that not all of the general purpose registers are used in the new one, and instead values are coming from memory.

Our state machine simulates the Fibonacci sequence. The first state sends an immediate of one that gets written to the memories second slot. The following terms are then computed from just this one immediate by summing the ith term with the ith - 1 term and putting the result into the ith + 1 memory address. This FSM controls all of the buffer signal inputs, as well as the memory inputs and ALU and Register File inputs. Once we had a properly written FSM we wrote a test bench to simulate the FSM. After a few tweaks our simulation gave us our expected Fibonacci values. We then went to implement this on the board and did not get the values we expected. After hours of tweaking the code, we could still not get the board to produce the results we were expecting. After simulating and retesting we hardcoded the values into the state machine instead of calculating memory addresses, which then gave us the results we expected. We still do not know why the state machine was not working when calculating memory addresses and plan on trying to figure out how to still get the state machine to work by calculating values instead of hard-coding.

Designing and implementing the memory module was relatively straightforward. Connecting it to the previously built modules and simulating the modules was also relatively straightforward. What gave our team the most trouble was getting the uploaded code to behave as expected,

which is possibly the most important part. We spent many hours trying to figure out why the machine wouldn't react to calculated memory addresses instead of hard coded ones, but eventually discovered the problem was due to simultaneously attempting to write to, and read from, the register containing the memory address, which ensured that we were seeing garbage as the memory's output.

C. Control

A processor contains multiple hardware modules that each perform different functions. One of these modules is a control unit. The control unit controls the signals given to various other components such as memory, the ALU, and register file. Each of these modules takes inputs (either data or control signals) and produces outputs based on digital logic. Our team has designed and implemented a Register File, Memory Unit and Arithmetic Logic Unit for a RISC type processor. To complete our processor and turn it into a closed system a Control Unit is needed. In designing a control unit for our processor many design issues were faced, due to the nature of interfacing with other previously built hardware modules.The Control Unit we designed and implemented is in the form of a Moore type finite state machine. In addition to a Control Unit a Program Counter was designed, implemented and added, and several buffers were also added.

After designing and implementing all hardware modules needed for a Single Cycle RISC Architecture Processor (SCRAP) except for a control unit, we then designed and implemented a control unit. Our previously built hardware operates with 16-bit registers as well as a single cycle operation in mind with tri-state buffers connecting the different modules. Our control unit must take in instructions from memory and control the buffers in a way that all appropriate data will be put on the appropriate bus as well as sending the appropriate signals to the ALU, Register File, and Memory modules for correct operation. Several design considerations and issues were faced with interfacing the control unit with the previously built hardware.

The input signals for our control unit are a 16-bit instruction, 5-bit flag input, 1-bit clock, and a 1-bit reset signal. The output control signals are 1-bit memOutEnA to enable port A's data-out onto the main bus as well as for memOutEnB, a 5-bit regEn to choose which register in the register file to load data to, 1-bit Carry-in for carrying in to ALU, ImmEn for putting an immediate onto ALU, memReadEn, to enable memory, a buffer aluToMemEn signal for putting alu data onto the memory bus, 8-bit ALU op-code for controlling the ALU's function, two 5-bit bufen signals for buffering registers onto the ALU, a 16-bit immediate signal that's used in the ALU if immen is enabled, two 1-bit memEn signals for port A and B memory enabling, as well as two 1-bit write signals for each port to enable writing, 10-bit branchoffsetImm for branching relative to the PC, 1-bit branchEn to enable branching, 1-bit jumpEn for jumps, 1-bit PCen that when enabled increments the PC, 1-bit pcAddrEn that when enabled buffers a pc value given from a register to the PC register, and a regAddrEn that when enabled puts a value from a register to the data memory input. Each different state in our processor sets all signals to a specific value based on the state.

This paper will focus on the following challenges and how we designed our control unit to cope with these challenges: The Memory Interface, Instruction Decoder, Program Counter Design and Signed Arithmetic for Unsigned PC Addressing, Status Registers and Flags, and Sign Extension for Immediates.

We chose to implement our control unit as a Moore type finite state machine. We originally thought that with our architecture we could fetch, decode, and execute an instruction all in one clock cycle. We quickly realized that each of these processes would take their own clock cycle

to complete and thus the program counter not incremented every clock cycle which means that we could not fetch a new instruction every clock cycle. If we chose to pipeline our design we could have still fetched an instruction every clock cycle, but we did not choose to implement a pipelined processor. The states for our process are FETCH, DECODE, EXECUTE, LOAD DATA, LOAD DATA EXECUTE, STORE, BRANCH, and JUMP. Fetch and Decode are needed for every instruction, and then depending on the type of instruction either execute, load data, store, branch, or jump will be the next state. From branch the next state is Fetch, from store the next state is fetch, from load data the next state is data execute and then to fetch, and from execute goes back to fetch. In our Verilog code we have a task assigned to every state, which uses combinational logic to set the control signals.

Our memory uses only 14 bits for a memory address length. We also implemented our memory as a read through write-first memory. Both our data and instructions are held in the same memory unit, and thus must be organized in a way that they don't interfere with each other. Our memory is also a dual-port memory, but our processor can only access port A of the memory, and port b is read-only (for later use by VGA controller).

For any given instruction (except NOP) the memory will be accessed a minimum of once and a maximum of twice, although for each of these accesses the values on all registers in the memory module must remain the same for two clock cycles. This is because we are built in registers in the memory module to hold the data instead of our own external registers. For any instruction it takes one clock cycle to fetch the instruction from memory, and then an additional clock cycle to decode the instruction. Since we do not have any registers to hold the values of the memory address, or data inside of it, all signal values for the memory module must remain constant and none of its inputs should change from FETCH to DECODE.

One of the most critical design problems is how to decode an instruction coming out of memory. Depending on the protocol for decoding an instruction, and what hardware is supported, this can determine how many ALU operations can be supported, how big of immediates can be used, how many registers can be accessed, and what the assembly language will be formatted as among other things. Already we have 16 registers in the register file that will be accessed up to two at a time. This corresponds to a four-bit register address for buffering register outputs to the ALU. The ALU takes two 16-bit(corresponding to the 16-bit wide registers) inputs that can both be from the register file. Since two registers must be specified, this corresponds to 8 bits being taken up in the opcode for a source and destination registers for R type instructions. This leaves another 8 bits for ALU op code. If the instruction is an I-type instruction, the source register uses 4 bits, and the op-code uses 4 bits which leaves 8 bits for a signed immediate value.

The program counter must be able to increment when regularly enabled, calculate and assign a new PC address with a signed immediate offset, jump to a specific address, and remain constant all depending on the control signals applied. We implemented the PC as its own module with a single 10 bit output register. This output register is 10 bits wide because right now our memory module has a ten-bit address corresponding to 1 kilobyte of memory. Our PC module takes a 1-bit input CLK input, 1-bit PCEn input, 1-bit BranchEn input, 10-bit branch immediate offset, 1-bit JumpEn, 10-bit JumpAddress, and a 1-bit Reset.

Besides the CLK input, the control unit must ensure that none of the 1-bit input signals are high at the same time. On every positive CLK edge the PC is re-evaluated. If Reset is high, then the PC jumps to the initial address of instruction memory. If BranchEn is high then the Branch immediate offset is interpreted as a signed two's complement number and is added to the current PC value to determine the value of the next address of the next instruction. If JumpEn is high

then the PC is set to the 10-bit JumpAddress input and interpreted as unsigned.

Our control Unit takes in a five bit input for flags. These flags correspond to zero, carry, overflow, unsigned low, and signed low. These flags are used with conditional jumping and branching. A conditional branch or jump will be done immediately after a comparison of two values, and then set the PC based on the comparison. The most important flag for our control unit is the zero flag. If the zero flag is high this means that the values in the comparison were equal and a jump must be taken/not taken. We plan to implement a heap and stack in the memory for passing status variables as well using the RISC specified general purpose register format.

Our ALU takes two sixteen bit inputs and does computations on them. The ALU does not care where the source for the numbers are. Since this value could be an immediate value coming from an instruction, the immediate value must be sign extended before going to the ALU. Our control unit takes care of this. The control unit analyzes the instruction to first determine if it is a signed or unsigned operation, extends the 8-bit immediate appropriately depending on the instruction and then feeds this to the immediate buffer on Bus B of the ALU.

Our FSM consists of eight possible states that it can be in. These are FETCH, DECODE, EXECUTE, LOAD_ DATA, LOAD_ DATA_ EXECUTE, STORE, BRANCH, JUMP and STORE_ MIDI_ NOTE. The fetch state is the initial state of the machine when reset, and points to the first instruction in memory when turned on or reset. THE next state after FETCH is always DECODE. The DECODE state analyzes the instruction and determines what type of instruction it is. If the instruction is a load instruction then LOAD_ DATA is the next state, which prepares the memory for an appropriate followed by LOAD_ DATA_ EXECUTE which stores the data to a register. The next state is then FETCH again. If the instruction is a branch or jump the next states are BRANCH and JUMP respectively, which both set the PC appropriately. The next state after a BRANCH or JUMP is FETCH. IF the Instruction is an I or R type instruction then EXECUTE is the next instruction. The next state from the EXECUTE state is the FETCH state.

The FETCH state enables the memory to be read with the address given by the current PC. This state enables the pcAddrEn buffer which puts the PC value into the address input of the memory module, as well as storing the Instruction at the current PC address into a register. The ALU is set to a No-Op state. It turns all other control signals to off as well and sets the NextState register to DECODE. The DECODE state keeps most of the control signals off as well and maintains the pcAddrEn buffer to a one and also maintains the ALU to at a No-Op state. The DECODE state analyzes the 8-bit instruction code and states the next state appropriately. If the Op-code corresponds to an I or R type instruction the EXECUTE is set as the NextState. If the Op-Code corresponds to load, store, branch or jump, it then sets the NextState to LOAD_ DATA, STORE, BRANCH, and JUMP respectively.

The EXECUTE state is for R and I type instructions only. The EXECUTE state enables the PC to be incremented, disables branching and jumping, as well as memory. This state determines if the type of instruction is an I or R type of instruction. If the Instruction is I type this is where the sign extending of the immediate happens. This state looks at the op-code to determine if the operation is a signed or unsigned operation and if it is an I-type instruction extends the immediate appropriately. This state also sets the ALU control signals, and bus signals for the register file and ALU so that the appropriate sources and destinations are used. If the instruction is a load an instruction the state after DECODE is LOAD_ DATA. Loading from memory takes two clock cycles. The first is to give the correct address to the memory module without actually enabling the memory. The second is to store the loaded value into a register. The LOAD_ DATA is the first state which puts the appropriate address of the data to be loaded on the memAddrBuff

and disables all memory signals. The state following directly after LOAD_ DATA is LOAD_ DATA_ EXECUTE. After LOAD_ DATA sets the address for the correct memory address to be loaded from, LOAD_ DATA_ EXECUTE enables the correct register for the loaded value to be stored to, as well as setting the appropriate memory read signals and buffer signals to get the data from the previous state's address.

If the current instruction is a store instruction, the STORE state will be after DECODE. The store STATE sets the memory control signals to a write mode at the appropriate address. The data inside of the source register specified by the instruction is stored into the address of the destination register. Immediately after a store instruction the state following will be FETCH.

If the current instruction is a branch or a jump, the BRANCH or JUMP state will be immediately after the DECODE state. The BRANCH state uses the branch immediate offset to calculate a new PC address relative to the current PC as well as setting the appropriate control signals for branching. The JUMP instruction uses a direct address to jump to (stored in the source register specified in the instruction). For both the BRANCH and JUMP state the Next State register is set to the FETCH state. To test our control unit we wrote a Fibonacci program in binary and loaded into the instruction memory. We also wrote a separate program to test looping. After some tweaking of our states, our programs produced the expected results.

In order to incorporate the MIDI input, we also include a STORE_ MIDI_ NOTE state, which allows the MIDI memory manager to set the memory address and write an incoming MIDI note to memory. The logic is situated in the clock-sensitive always block, which watches for a flag that indicates a note is ready and pauses the control flow before a new FETCH to do so.

D. Assembler

The assembler is written in Java, and contains a simple GUI. The left hand pane of the window is where assembly code is written, and the right-hand pane is where the binary translation appears. Our assembler supports labels, which it accomplishes by implementing a two-pass structure. The first pass locates labels and stores them in a list, the second pass creates a jump table starting at address 0x0. When a label is referred to for jumping purposes, the assembler replaces the single line of code for a jump with four lines: the first loads the address of the jump table entry as an immediate into a designated register; the second line loads the data at the jump table's address into another register (which is the location to jump to on the proper condition); the third line performs the comparison required for the jump (equal to, etc); the final line then jumps to the address in the register or falls through, based upon the flag configuration from the previous line.

Our assembler also contains some other features, including: passing comments from the assembly to the binary; saving binary text files or assembly text files; opening saved assembly text files; and the ability to change the registers used for jumping.

E. I/O: MIDI Input

MIDI input is used as the game's controller, and required both a physical circuit and a Verilog hardware module to instantiate. MIDI has a well-established protocol and extensive documentation from the MIDI Manufacturers Corporation. In order to connect the MIDI output of a piano-keyboard to the Nexys 3, we created a small circuit that uses an opto-isolator. The opto-isolator completes the MIDI output current loop, allowing the keyboard to use its internal ground. With the current loop completed, the keyboard is able to run 5mA through the output to represent a zero, and stop the current to represent a one. The other end of the opto-isolator

uses the voltage source and ground of the Nexys 3 board, which allows us to output 3.3 V to represent current flowing from the MIDI output, or 0V to represent no current, in a way that does not harm the Nexys 3.

With a working opto-isolated signal, we use a dedicated module to detect MIDI bit-streams. Since MIDI is a serial bit-stream, our module waits for a start bit from the keyboard, then samples every several hundered clock cycles to get the next bit, until 30 bits (plus a start and stop bit) have been received. Since our game only is interested in key presses, the module only locks the input into a register and raises a "note ready" flag if the data is a key press.

Upon a "note ready" signal, the processor's controller is interrupted before the next FETCH state, and is forced into a STORE_ MIDI_ NOTE state, which puts the note in an open "Notes in Play" section, then returns control flow to the software, and sends a "reset" signal to the MIDI input module, allowing for another note to be received.

F. I/O: VGA

The VGA monitor we used in our project has a resolution of 480 x 640 pixels. The VGA protocol acts similarly to a digital typewriter. The protocol is built around the Cathode Ray Tube monitors, which have an electron gun in the back shooting at each pixel one at a time. With an eight-bit input for color, the value on these eight bits at any given time is the value to be displayed at the current pixel the electron gun is "shooting".

In addition to the eight-bit input for color there is also a vertical sync and horizontal sync that are both one-bit active low signals. When the horizontal sync is active the gun gets reset to the first pixel of the next row, when the vertical sync is active the gun gets reset to the top row of pixels. Most of the graphics used in our game are directly mapped. These graphics include the background colors, the piano keyboard, and the rope. This means that for the direct mapped graphics, memory is not being accessed to generate the glyphs. The ball, the right man, and the left man are not directly mapped and are being pulled from memory.

To put graphics onto the screen we created a bitgen module that uses counters to display the correct information at the right time onto the screen. These counters count clock cycles and simulate a 25MHz clock for the VGA module. The counters include a pixel counter to know where the electron gun is on the screen, as well as outputting the appropriate vsync and hsync pulses for the VGA. The bitgen module also has access to memory and stores the memory addresses of the glyphs and calculates offsets for viewing the glyphs on the screen as well as the offset for the memory address to grab.

To keep track of the glyphs, we created a midinotemanager for each pong ball in play. This module contains a timer to keep track of appropriate times that the ball can be hit for points or misses. Our original design goal was to have one or two shared registers to keep track of the notes in play. Our assembly code was pretty much a continuous loop to check these status registers and update the score, and the screen as changes happened in the game.

III. SOFTWARE

The software was unfortunately never fully developed. But what was created began by initializing the memory that stores the notes to zero. It then consisted of a single large loop, which would start at the lowest memory address that stores notes, and check its value. If the value was 0, the memory location contained no note, and the software added one to the memory location and checked again. If the value contained a 1 or a 2, then a note that had previously been in play was no longer in play and the score was to be adjusted in against whichever player missed

the note, (i.e., a 0x1 means player one had missed the note). If the data was any other value, it was a note, and the software updated a bit in R8, which acted as a status bit for the note managing modules.

The status bit in R8 was controlled by using the temporary registers which would contain a one and all zeros or all zeros and a one, shifted to the proper position. For example, if the status of note two was to be set to one, a register would be set to one and shifted left twice, then R8 would be set to R8 OR the temp register. This was how we were planning on controlling multiple notes at once on screen, where each note would have its own note manager.

When a player missed a point, the corresponding bit in R8 would be set to 0, and the score register would be added or subtracted by one depending on the player. Since the tug-of-war score keeping is a zero-sum game, we only needed one register to track the score.

IV. System Integration

There were two phases of system integration for our group: first, the attempt to get our processor and software working, which encountered fatal bugs (see subsection "Problems, Solutions, and Lessons Learned").

Since we use direct mapped graphics, the module in charge of calculating the note's position on the screen was also responsible for starting a timer at the correct position on the screen in order to check for an incoming key-press. If the key-press was the correct note, the module did nothing, but if the key-press was incorrect, a flag was raised to indicate which player was to lose the point.

The next major task when Integrating was allowing both the MIDI input, the note manager, and the processor to share the ability to write to memory. We successfully accomplish this by instantiating a new module which we call the Memory Manager. Upon the receipt of an interrupt signal from the Control unit (which indicated a note ready for storage and ensured no other instruction was interrupted), the Memory Manager used a MUX to switch memory's data input from the processor to the MIDI input module.

The same happens if a note manager signaled that a player had pressed an incorrect key. The Memory Manager then writes a 1 or 2 to the memory address which allows the software to detect that a note's status was no longer valid, and the software needed to change the score.

A. Problems, Solutions, and Lessons Learned

Our largest problem was an ALU bug which didn't show itself until the week before the demonstration day. Our mistake was that certain operations, such as NO_ OP and CMP were assigning values to be the previous value, instead of "don't care" values. As a result, latches were being created which would cause our ALU to be out of sync with the rest of the processor.

As a result of this bug, we were unable to ever use the software we had created for the processor, and as a work-around, created a version of the game that was completely hardware. This required us to pull the memory out of the processor for glyph access, and create a lot of extra logic in the note manager module to control the game state. The final module we had to create was a score-keeping module. Another result of this work-around was the game became a much simplified version of what we had originally envisioned: it is less of a memory game, and rather, requires the players to react to quick changes in the key presses.

V. Future Development

As far as future development is concerned, the best thing to do would be to first get everything working properly. The amount of latches created in the ALU became a huge stumbling block for the project. The first step in any sort of development would be to fix that issue. Once this is done other things that could be added would be varying game modes, things like being able to select a smaller portion of the keyboard to use, or changing the speed of the ball.

VI. Conclusion

While all of our pieces worked very well individually, the cohesive whole of the project was never fully realized. The ALU utilized some poor design choices which created more problems than our team could fix. The fact that the problem was not quickly recognized and dealt with made the end goal of the project harder to attain. The team was able to put together a very hardware oriented version of the end goal, but had to leave out parts of the processor which would have streamlined the whole thing. There was a lot of success as well. The team was able to isolate and use MIDI signals as an input. This along with the VGA, control unit, and memory operations made it so we were able to show the basic working function of the peripherals with the hardware that we wanted. With a bit more time the problems could be worked out and the processor would be made to work as intended.

TABLE I
SIMPLE ASSEMBLY INSTRUCTIONS

Instruction	**Arguments**	**Format**	**Notes**
ADD	rsrc, rdest	add [rsrc] [rdest]	[rdest] ← [rsrc] + [rdest]
ADDI	imm, rdest	addi [imm] [rdest]	[rdest] ← [imm] + [rdest]
ADDU	rsrc, rdst	addu [rsrc] [rdest]	[rdest] ← [rsrc] + [rdest]
ADDUI	imm, rdest	addui [imm] [rdest]	[rdest] ← [imm] + [rdest]
SUB	rsrc, rdst	sub [rsrc] [rdest]	[rdest] ← [rsrc] - [rdest]
SUBI	imm, rdest	subi [imm] [rdest]	[rdest] ← [imm] - [rdest]
CMP	rsrc, rdst	cmp [rsrc] [rdest]	sets flags; preserves [rdest]
CMPI	imm, rdest	cmpi [imm] [rdest]	sets flags; preserves [rdest]
AND	rsrc, rdst	and [rsrc] [rdest]	[rdest] ← [rsrc] & [rdest]
ANDI	imm, rdest	and [imm] [rdest]	[rdest] ← [imm] & [rdest]
OR	rsrc, rdst	or [rsrc] [rdest]	[rdest] ← [rsrc] | [rdest]
ORI	imm, rdest	ori [imm] [rdest]	[rdest] ← [imm] | [rdest]
XOR	rsrc, rdst	xor [rsrc] [rdest]	[rdest] ← [rsrc] ^ [rdest]
XORI	imm, rdest	xori [imm] [rdest]	[rdest] ← [imm] ^ [rdest]
MOV	rsrc, rdst	mov [rsrc] [rdest]	[rdest] ← [rsrc]
MOVI	imm, rdest	movi [imm] [rdest]	[rdest] ← [imm]
LSH	ramount, rdest	lsh [imm] [rdest]	[rdest] ← [rdest] << $\pm$1
ASH	ramount, rdest	ash [imm] [rdest]	[rdest] ← [rdest] << $\pm$1 (sign extended)
LOAD	rdest, raddr	load [rdest] [raddr]	[rdest] ← mem[raddr]
STOR	rsrc, raddr	stor [rsrc] [raddr]	mem[raddr] ← [rsrc]

TABLE II
ASSEMBLY JUMP CONDITIONAL WITH LABEL USAGE

Instruction	**Operation**	**Resulting Instruction Sequence***
jcond [label] [rsrc1] [rsrc2] [condition]	*if ([rsrc1] [condition] [rsrc2]) $\{PC \leftarrow [label]\}$* *else $\{PC \leftarrow PC + 1\}$*	*movi [label pointer] [rx]* *load [ry] [rx]* *comp rsrc1 rsrc2* *jcond [condition] [rx]*
jcond loopHere r1 r2 eq	if (r1 == r2) $\{$PC $\leftarrow$ [.loopHere + 1]$\}$ else $\{$PC $\leftarrow$ PC + 1$\}$	movi [loopHere pointer] [rx] load [ry] [loopHere address] comp [r1] [r2] jcond eq [rx]

*Choose registers [rx] and [ry] via the "Memory Config" button; they default to R13 and R14, respectively.

TABLE III
ASSEMBLY CONDITION CODES

Condition	**Condition Code** *[condition]*	**Flag Configuration**
Equal	EQ	Z = 1
Not Equal	NE	Z = 0
Greater than or Equal	GE	N = 1 or Z = 1
Carry Set	CS	C = 1
Carry Clear	CC	C = 0
Higher than	HI	L = 1
Lower than or the Same as	LS	L = 0
Lower than	LO	L = 0 and Z = 0
Higher than or the Same as	HS	L = 1 or Z = 1
Greater than	GT	N = 1
Less than or Equal	LE	N = 0
Flag Set	FS	F = 1
Flag Clear	FC	F = 0
Less than	LT	N = 0 and Z = 0

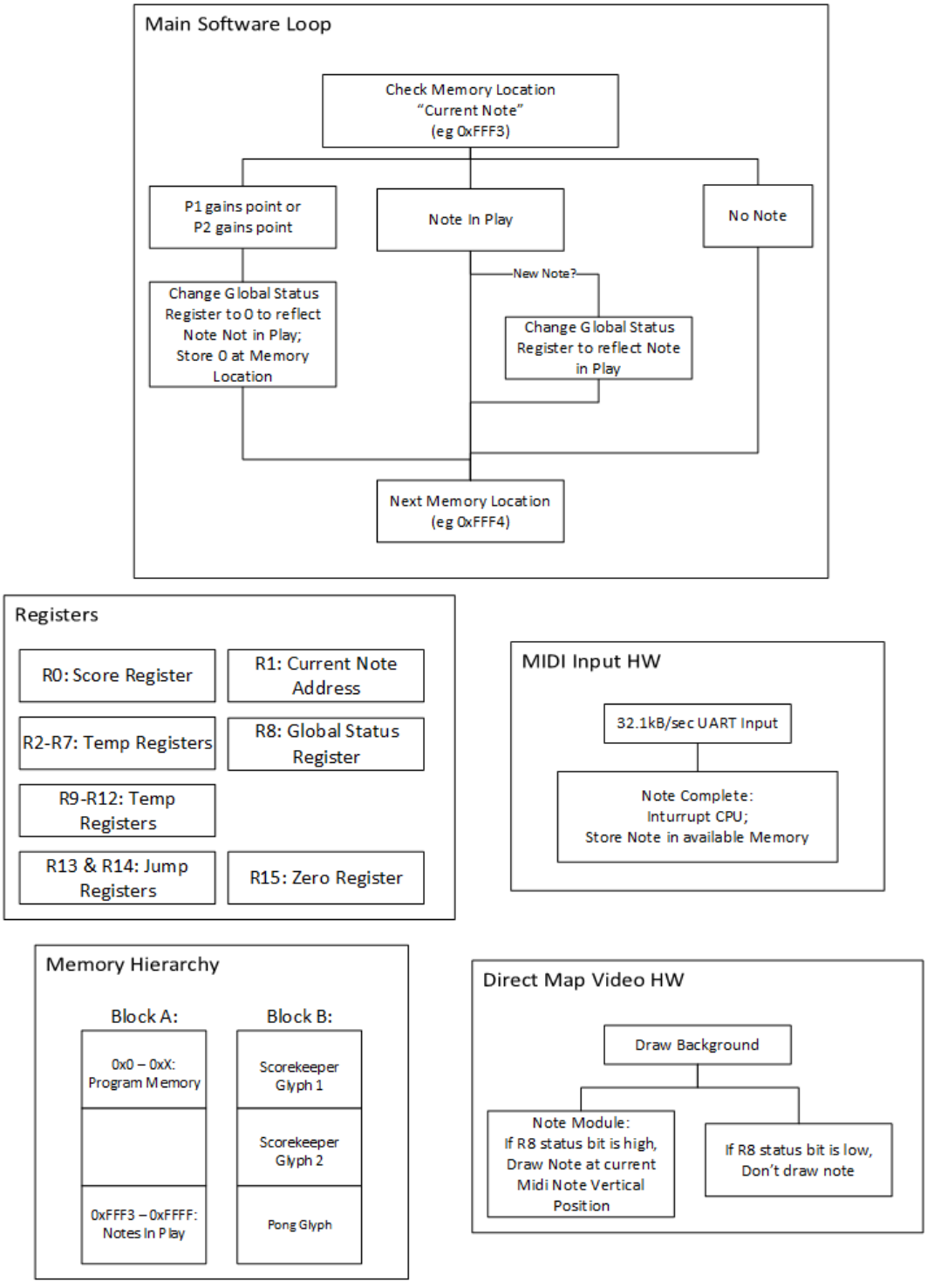

Fig. 1. Structures and Flow Charts of Piano Pong

The top figure shows the main loop our software executed. The rest of the figures show how our memory, registers, and hardware modules were structured.

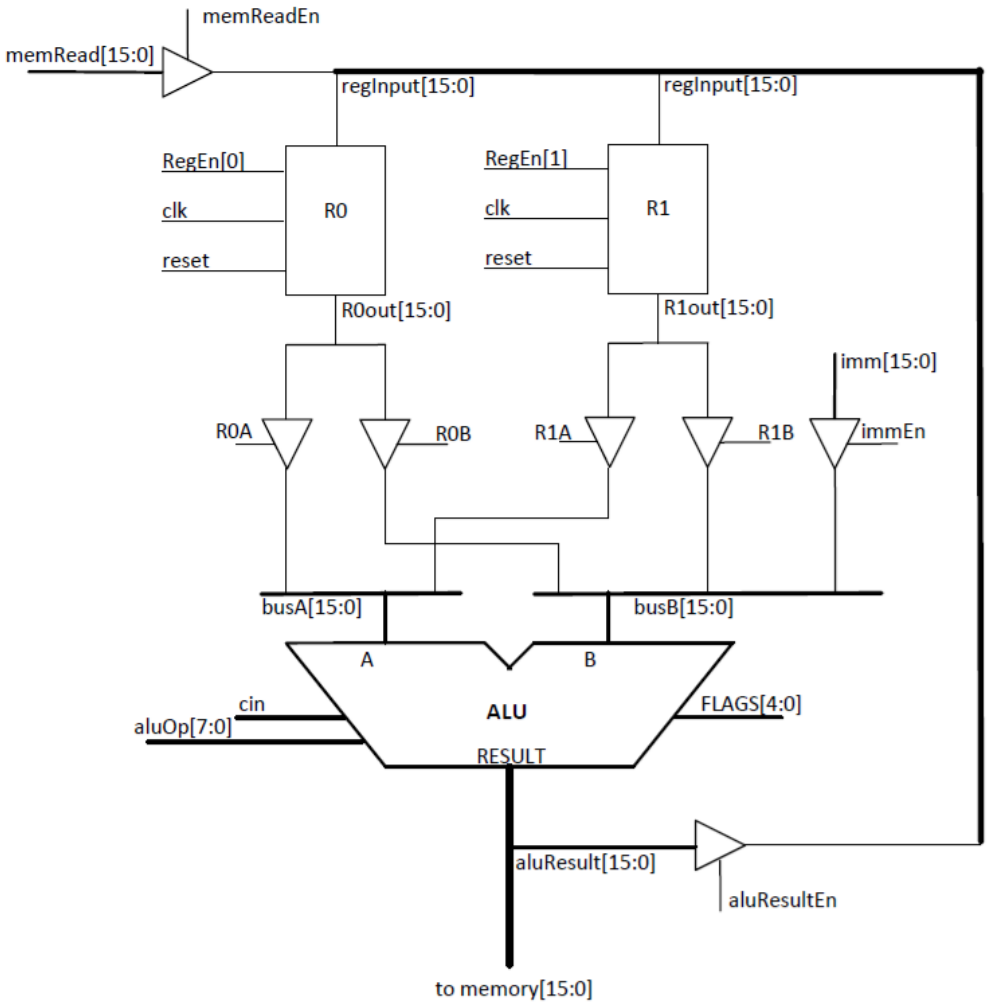

Fig. 2. Structure of the ALU

This is an abbreviated version of the ALU, which in reality contains 16 registers.

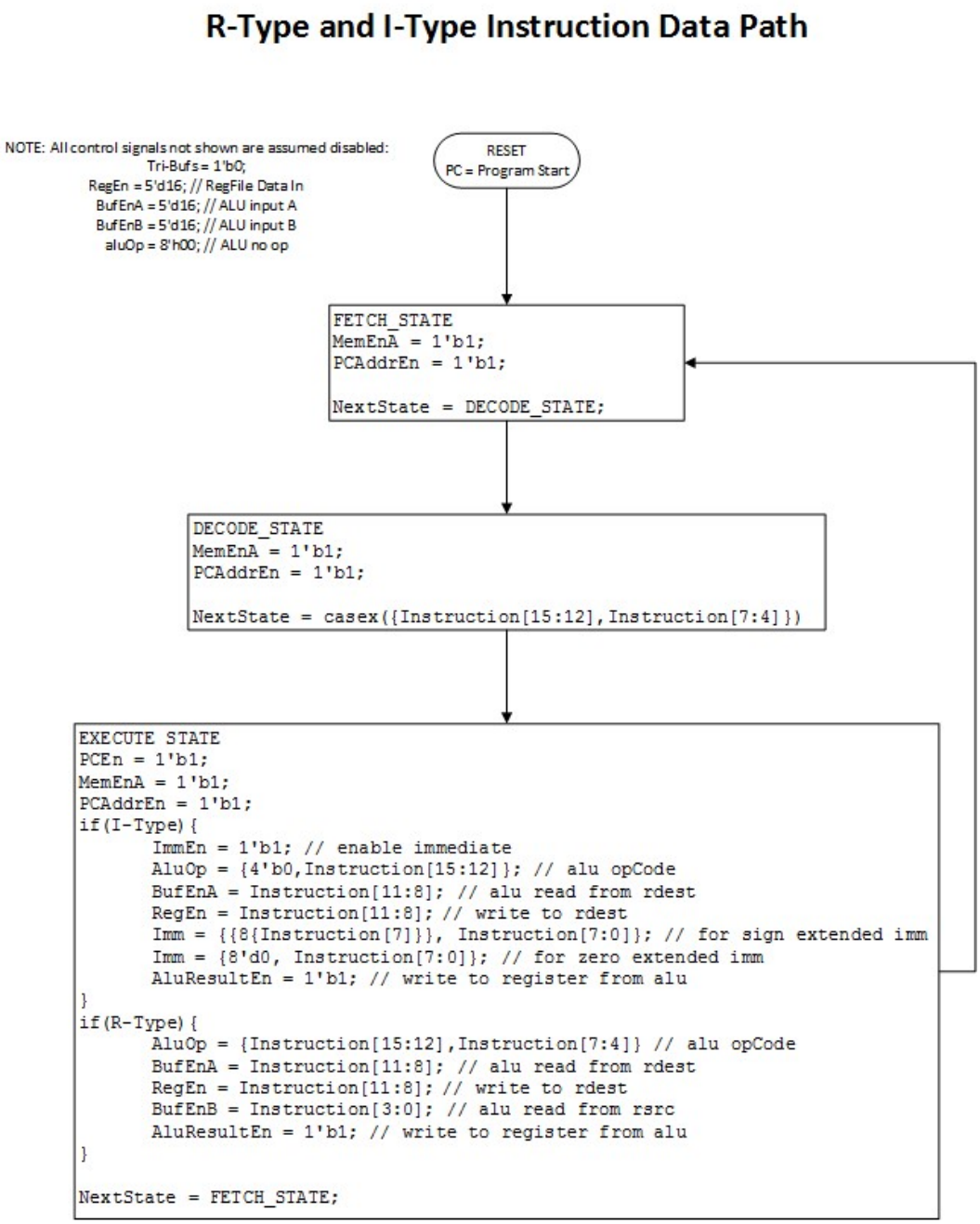

Fig. 3. Control Flow of R- and I-Type Instructions

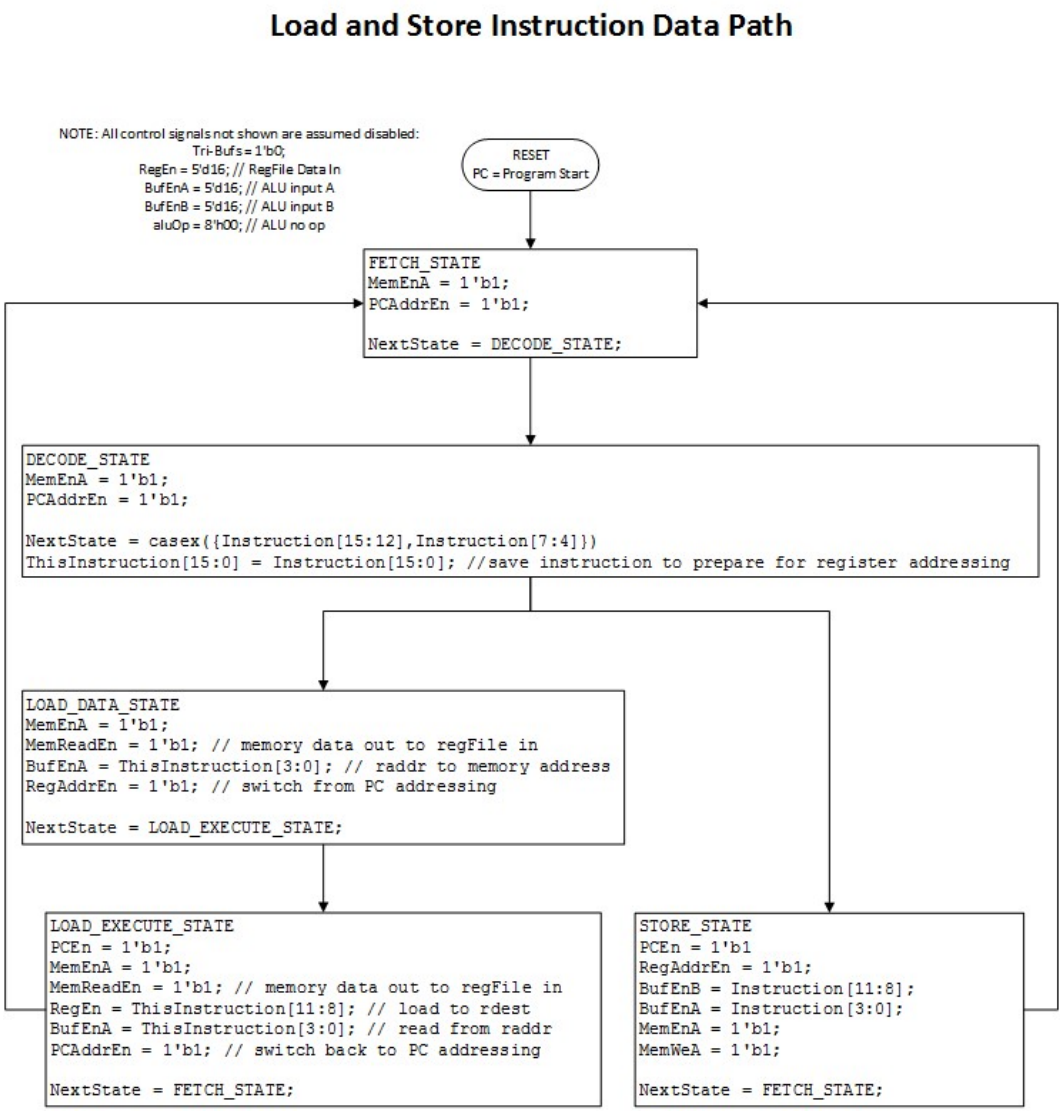

Fig. 4. Control Flow of Load and Store Instructions

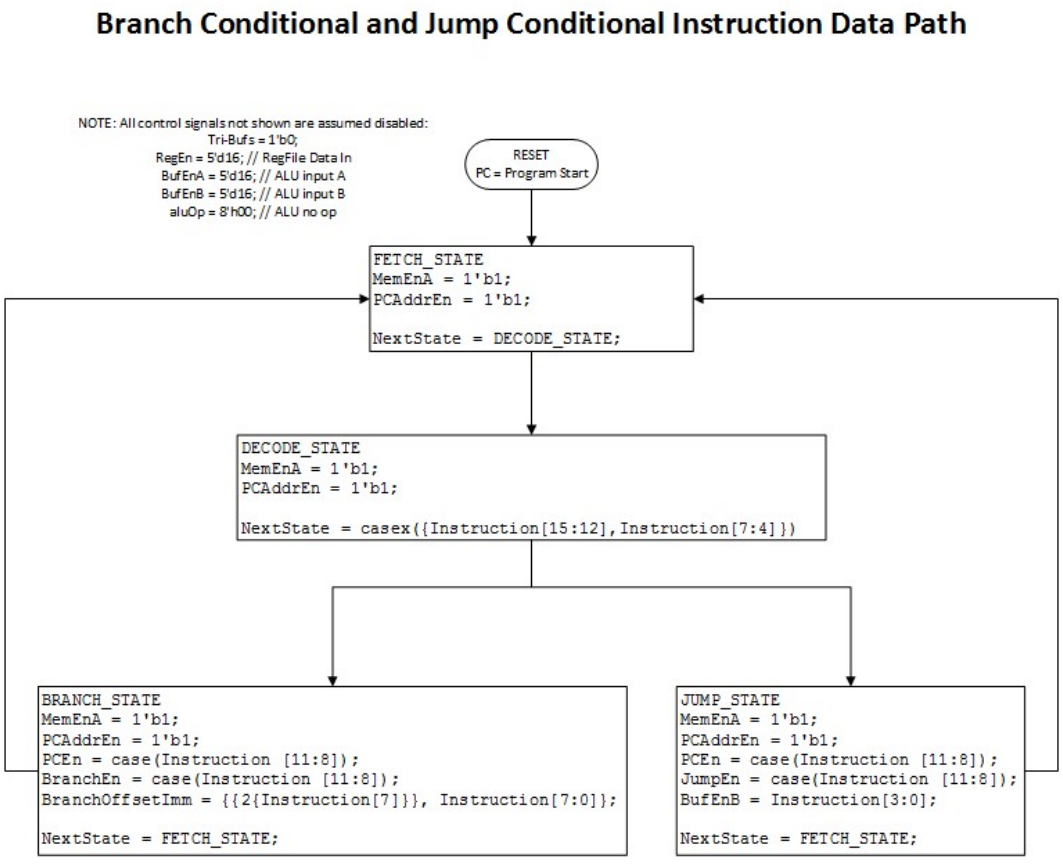

Fig. 5. Control Flow of Branch and Jump Instructions