GLYCH

Team: The FreqZ

Steven Brown, Travis Gray, Matthew Humphries, Mark Stacey

ECE 3710

University Of Utah

Fall 2014

**Application**

GLYCH is a four player third-person shooter game inside a maze. It is a game of elimination, meaning the last player remaining wins. The game begins by loading maze onto the screen and each of the four players being placed into one of the four corners of the screen. Each player will be a different color stick figure to allow the players to identify their character on the screen. When the game begins the players will navigate through the maze to eliminate other players by firing lasers. The rules of the game prohibit shooting through walls, and shooting in the north and south directions of the screen, allowing for only horizontal attacks. When two players are on the same level, and no wall separates them, they are poised for attack. When this occurs, if a laser hits a player, the player is eliminated and the character will no longer display on the screen. The game continues until only on player is left on the screen.

Audio effects will enhance the game by alerting a player when a laser had been fired, and termination effects being sounded when a laser has hit a player.

**Proposed Modifications to the ISA**

The main modification to be made to the provided ISA is to change the processor to a 32-bit processor. In this sense, we will ensure ample memory space with the possibility of accessing cellularRAM if necessary, and compress the amount of instructions required for our processor through the use of 32-bit registers.

**I/O**

The input and output required by this application will be:

1. VGA Display
2. 4 NES gaming controllers
3. Speakers

The VGA Display will be Glyph based to minimize memory consumption.

The NES gaming controllers will provide user the ability to interact with the application and will be used to play the game.

The sound will be provided through a music synthesizer written in Verilog, which can then be outputted to a speaker through a PMOD connector to the NEXYS board.

**Assembler Plan**

The assembler will be written in Python. The application will be written by using commas to separate both instructions, registers, immediate values, and comments, to allow for easy parsing.

**Team Responsibilities**

|  |  |  |
| --- | --- | --- |
| **Team Member** | **Lead Role** | **Backup Role** |
| Steven Brown | Application | Processor |
| Travis Gray | I/O | Assembler |
| Matthew Humphries | Processor | Application |
| Mark Stacey | Assembler | I/O |

**Instruction Set Architecture (ISA)**

Each instruction influences the data path in a unique way. Our processor by default sets all control signals to 0 (except for PCen, and enRAM/enROM), and therefore, as the control signals are explained, only signals that will need to change state from 0 will be listed for each instruction.

1. **addi**  
   addi is an I-Type instruction with an 18-bit, signed immediate value. Because the immediate value is interpreted as a two’s complement number, it can be used to add both positive and negative integer values. When the instruction is decoded, the Operation Code that goes to the Logic Controller is 4'b0101. The destination register is decoded from bits [27:23] of the instruction, and the source register, Rs, is decoded from bits [23:18] of the instruction. The immediate value is decoded from bits [17:0], but is fed into a sign extender (to preserve the two’s complement sign of the number) to be used in computation as a 32-bit number. From the Logic Controller, aluSrcb is set to a 0, which allows the immediate value to pass through the multiplexer determining the value of the second argument into the ALU. The Logic Controller also sets aluop to 3’b000 to perform addition. Thus the ALU utilizes the 32-bit data stored in Rs and the sign extended 32-bit immediate value to perform the add computation. The result of the ALU is passed through a mux controlled by memSrc, which is set to 1 to allow the ALU data to pass through. The data then passes through both the wbSrc and wbPSR muxes till it gets to the RaWriteMUX, where it also passes through. The Logic Controller sets regWriteEn high at the beginning of the instruction, and thus computation can be stored into the destination register. This instruction will have an effect on the C and F flags of the Program Status Register depending on the resulting ALU outcome. Refer to the diagram of the processor and the outlined control signals below for visualization.  
    *Control Signals:* CFwrite = 1;  
    memSrc = 1;  
    wbSrc = 1;  
    regWriteEn = 1;
2. **sub**  
   sub is an R-Type instruction. Because this is an R-Type instruction, the two arguments are registers that contain data to be subtracted. The Operation Code that goes to the Logic Controller is 4'b0000. The associated decoded Function Code comes from bits [3:0] of the instruction and are 4'b1001 for subtraction. The resulting control signals from the Logic Controller set the ALU to perform subtraction on two signed numbers. Three operands are considered for this instruction: Rdest, bits [27:23] of the instruction; Rs, bits [22:18] of the instruction; and Rt, bits [17:13] of the instruction. The Register File reads out the 32-bit data from Rs, called RsData, and the 32-bit data from Rt, called RtData to be used in the ALU to perform subtraction. The output of the ALU then contains the subtracted value of Rs minus Rt. This value is to be written back into the destination register, Rdest, so it bypasses the data RAM, and goes through a series of multiplexors until the data is ultimately passed into the port see at the Register File for writing data. This instruction will have an effect on the C and F flags of the Program Status Register depending on the resulting ALU outcome. Refer to the diagram of the processor and the outlined control signals below for visualization.  
    *Control Signals:* CFWrite = 1'b1;   
    wbSrc = 1'b1;  
    memSrc = 1'b1;   
    aluSrcb = 1'b1;   
    regWriteEn = 1'b1;aluop = 3'b001;
3. **multi**multi is an I-Type instruction with an 18-bit signed immediate value. The two arguments for the desired operation are contained in the source register field, Rs, and the immediate field. The Operation Code that goes to the Logic Controller is 4'b1110. Because this is an I-Type instruction, there is no associated Function Code. The resulting control signals from the Logic Controller set the ALU to perform multiplication on two signed numbers. The data for the arguments comes from a 32-bit immediate value which is the sign extended immediate value from the instruction, and the 32-bit RsData coming out of the Register File. The two argument data go into the ALU where the resulting output of the ALU contains the multiplied data. The value is to be written back into the destination register specified in the instruction, Rdest, so it bypasses the data RAM, goes through a series of multiplexors and ultimately passes into the port at the Register File for writing data. This instruction has no effect on the Program Status Register. Refer to the diagram of the processor and the outlined control signals below for visualization.   
    *Control Signals:* wbSrc <= 1'b1;   
    memSrc <= 1'b1;   
    regWriteEn <= 1'b1;   
    aluop <= 3'b110;
4. **cmpi**   
   cmpi is an I-Type instruction with an 18-bit signed immediate value. The two arguments for the desired operation are contained in the source register field, Rs, and the immediate field. The Operation Code that goes into the Logic Controller is 4'b1011. Because this is an I-Type instruction, there is no associated Function Code. The resulting control signals from the Logic Controller set the ALU to perform a comparison on two signed numbers. The data for the arguments comes from the 32-bit sign extended immediate value of the instruction and the 32-bit RsData coming out of the Register File. This comparison is done by performing a subtraction. If the difference of the two values is positive, then Rs is larger than the inputted immediate value. If they are equal, then the Z Flag in the Program Status Register is set high. If the difference is negative, then the N Flag in the Program Status Register is set high. It is important to note that though this instruction is an I-Type instruction, it is undesirable to write back the result into a register; therefore, regWriteEn remains low. The purpose of this instruction is to set the Program Status Register to use for conditional branches, jumps, ect. Refer to the diagram of the processor and the outlined control signals below for visualization.  
    *Control Signals:* LZNWrite <= 1'b1;   
    wbSrc <= 1'b1;   
    memSrc <= 1'b1;   
    aluop <= 3'b111;
5. **andi**  
   andi is an I-Type instruction with an 18-bit immediate value. The two arguments for the desired operation are contained in the source register field, Rs, and the immediate field. The Operation Code that goes into the Logic Controller is 4'b0001. Because this is an I-Type instruction, there is no associated Function Code. The resulting control signals from the Logic controller set the ALU to perform a bitwise AND on the two numbers, therefore, treating them as signed numbers has no effect on this operation. The data for the arguments comes from the 32-bit sign extended immediate value of the instruction and the 32-bit RsData coming out of the Register File. The two argument data go into the ALU where the resulting output of the ALU contains the bitwise ANDed data. The value is to be written back into the destination register specified in the instruction, Rdest, so it bypasses the data RAM, geos through a series of multiplexors and ultimately passes into the port at the Register File for writing data. This instruction has no effect on the Program Status Register. Refer to the diagram of the processor and the outlined control signals below for visualization.  
    *Control Signals:* wbSrc <= 1'b1;   
    memSrc <= 1'b1;   
    regWriteEn <= 1'b1;   
    aluop <= 3'b011;
6. **ori**ori is an I-Type instruction with an 18-bit immediate value. The two arguments for the desired operation are contained in the source register field, Rs, and the immediate field. The Operation Code that goes into the Logic Controller is 4'b0010. Because this is an I-Type instruction, there is no associated Function Code. The resulting control signals from the Logic controller set the ALU to perform a bitwise OR on the two numbers, therefore, treating them as signed numbers has no effect on this operation. The data for the arguments comes from the 32-bit sign extended immediate value of the instruction and the 32-bit RsData coming out of the Register File. The two argument data go into the ALU where the resulting output of the ALU contains the bitwise ORed data. The value is to be written back into the destination register specified in the instruction, Rdest, so it bypasses the data RAM, geos through a series of multiplexors and ultimately passes into the port at the Register File for writing data. This instruction has no effect on the Program Status Register. Refer to the diagram of the processor and the outlined control signals below for visualization.  
    *Control Signals:* wbSrc <= 1'b1;   
    memSrc <= 1'b1;   
    regWriteEn <= 1'b1;   
    aluop <= 3'b010;
7. **xori**xori is an I-Type instruction with an 18-bit immediate value. The two arguments for the desired operation are contained in the source register field, Rs, and the immediate field. The Operation Code that goes into the Logic Controller is 4'b0011. Because this is an I-Type instruction, there is no associated Function Code. The resulting control signals from the Logic controller set the ALU to perform a bitwise XOR on the two numbers, therefore, treating them as signed numbers has no effect on this operation. The data for the arguments comes from the 32-bit sign extended immediate value of the instruction and the 32-bit RsData coming out of the Register File. The two argument data go into the ALU where the resulting output of the ALU contains the bitwise XORed data. The value is to be written back into the destination register specified in the instruction, Rdest, so it bypasses the data RAM, geos through a series of multiplexors and ultimately passes into the port at the Register File for writing data. This instruction has no effect on the Program Status Register. Refer to the diagram of the processor and the outlined control signals below for visualization.  
    *Control Signals:* wbSrc <= 1'b1;  
    memSrc <= 1'b1;   
    regWriteEn <= 1'b1;   
    aluop <= 3'b100;

**Assembler**

The Assembler is a python script that reads in a coma separated value (CSV) configuration file and a CSV assembly code file. The configuration file contains information specific to the assembly of each instruction. This information is used by the assembler to assembly each line of assembly code into 32 bit binary instructions. Once the code has been assembled it is saved in a .dat file with the same name as the assembly code file. The assembler may be run from the command line by changing the directory to the directory containing the assembler, configuration file and the code file. Then type in the name of the assembler, “AssembleTitan.py”, into the command line and hit enter. It may also be run as an executable in windows by double clicking the file.

The assembler is set up to assemble three different types of instructions r-type, i-type, and j-type instructions. Each instruction is 32 bits long with a 4 bit operation (op) code. After the operation code the rest of the bits are decoded depending on the type of instruction that is specified in the op code.

R-type instructions include a 4 bit op code followed by three 5 bit register codes. The first register code is the destination register (Rd). The next two registers are source registers Rs and Rt. The last 4 bits specify a function code that determines the operation of the instruction in combination with the op code. In r-type instructions the ALU operations happen on Rs and Rt. Then the result of the ALU operation is stored into Rd

R-type Instruction [ Op][ Rd ][ Rs ][ Rt ][ 9’b0][ Func(4bit) ]

I -Type instructions include a 4 bit op code followed by two 5 bit register codes. The first register code is the destination register (Rd). The second is the source register Rs. The remaining 18 bits are used as an immediate value. In i-type instructions the ALU operation happens between Rs and the immediate value.

Then the result of the ALU operation is stored back into Rd.

I-type Instruction [ Op][ Rd ][ Rs ][ Immediate Value ]

J-Type instructions include only a 4bit op code and a 28 bit immediate value. The immediate value is the address of the next instruction that will be executed. The program counter will be set to the value of the immediate and then continue normally from that point.

j-type Instruction [ Op][ Jump Address Immediate Value ]

The Assembler is written in Python. Inside the assembler the data path to a configuration file and an assembly code file are specified. Both of these files are written in coma separated value (CSV) format to make parsing the files easier. Firs the assembler reads in the configuration file. The configuration file provides information about op codes and register encoding. The first value on each line designates the line as an op code or register encoding. Empty lines and comments beginning will // are ignored.

For op codes the remaining values specify in order the type, instruction name, op code, and for r-type instructions the function code.

op,rtype,add,0000,0101

op,itype,addi,1010

op,jtype,j,1101

Lines where the first value is reg specify register encoding. After the first value the register name is specified followed by its encoding.

reg,r1,00001

Each op code line is stored into an array then that array is stored into a large array called ‘op’ inside of the assembler. This array contains all of the information required for assembling each instruction except for the specific encoding for each register. The register information is stored into a python dictionary called ‘reg’. Each entry in reg is the register name with the definition equal to the register encoding. Once ‘reg’ and ‘op’ are populated the assembler is prepared to read in the code file and begin assembly.

When the code file is read into the assembler empty lined and comments are ignored just as in the configuration file. Then the code is assembled line by line. The first value in the line is checked to see if it is a label. If it is then the label is store in a dictionary labels where the definition is an 28 bit binary number represents the value of the program counter (PC) at that point. The value of the PC is stored in a second dictionary called addresses, where the definition for each label is the value of the PC as an integer value. These are used later on to finish the encoding for jumps and branches. If the line is and instruction and not a label then the assembler proceeds to assemble the instruction.

To assembly each institution the op code, type, and (if necessary) the function code are determined. Then the assembly for each instruction is determined by its type. Each instruction is assembled by concatenating the op code with the register encodings and the immediate value or the function code.

Then the assembled instruction is appended to an array that contains the assembled instructions in order. Finally the PC is incremented for the next instruction.

Once the Assembly is complete the assembler loops through the array of assembled instructions. Every time a label is encountered the label is replaced with the encoding from the ‘labels’ dictionary.

After the labels are inserted the assembly is complete. The code is written to a .dat file using a CSV writer. This file is ready for use by Xilinx ISE. This completes the assembly of the code into binary instructions.

**Instruction Decoder**

*Input Control Signals:*

RtSrcReg: RtSrcReg is a control wire for the multiplexor that determines whether bits [28:24] are passed into the RegFile as Rt, or if bits [17:13] are passed into the RegFile as Rt. The purpose of this control signal and multiplexor serves specifically for load and store instructions where the

Instruction: The 32-bit binary number that is read from the Instruction ROM that needs to be decoded for use in the execution and memory phases. The parts of the number will be broken out to be used in the Logic Controller bits [31:28], Rdest bits [27:23], Rs bits [22:18], Rt bits [17:13], and an Immediate value bits [17:0]. The top five bits fed into the logic controller will control the multiplexors that allow the instruction to propagate through the execution and memory stages, and determine the operation that needs to be performed by the ALU. If the instruction being fed into the decoder is an R-type instruction, the bottom 4 bits [3:0] are then determined to be the function code, which will allow the ALU to perform the correct operation.

*Outputs:*

opCode: The opCode is the top 4 bits of the 32-bit binary instruction (bits 31:28). These bits determine the outputs of the logic controller for R-type, I-type and J-type instructions to allow the propagation of the instruction to proceed through the execution and memory stages and to set the ALU to perform the operation required for the current instruction.

functCode: The functCode is used by R-type instructions in junction with its respective opCode to determine the outputs of the logic controller to allow the propagation of the instruction to proceed through the execution and memory stages and to set the ALU to perform the operation required for the current instruction.

Rs: This represents an argument register for R-type and I-type instructions.

Rt: This represents an argument register for R-type and I-type instructions.

Rdest: This represents the destination register in which the results of R-type and I-type instructions will be stored.

Immediate: This represents an 18 bit binary number that is used during the execution stage for I-type instructions.

**Logic Controller**

The Logic Controller is a large Finite State Machine (FSM) that controls the operations of the processor based on the instructions loaded from the instruction Read Only Memory. Every instruction requires a unique path for the data to flow through the processor, making the Logic Controller one of the most important pieces of hardware to design.

When an instruction is loaded form ROM, it is decoded, and three critical pieces of information are sent to the Logic Controller: first, the Operation Code (opCode) of the current instruction, from bits [32:28]; secod, the Function Code (functionCode) of every instruction, bits [3:0]; and third, the source register, Rs, bits [22:18].

By nature, the FSM of the Logic Controller works on a Present State, Next State basis. Using the three inputted values of the opCode, functionCode, and Rs, the Logic Controller can determine the Next State (NS) of the processor. When the NS of the process or is assigned to the Present State (PS) of the processor on the positive edge of the clock, then the output of the PS is determined and sent out to all areas of the processor combinationally. These signals can be seen in blue in the provided figure of the processor.

Within the processor exist two possible states for the machine to operate in: the Execution state, and the Memory state. All instructions will require the processor to be in the Execution state, but only two instructions, load and store, will require the processor to enter the Memory state.

The execution state consist of everything from loading data from the Register File, to performing operations in the ALU, to writing back to the Register File; thus the feeling of a single cycle processor is maintained in this state. However, when a load or a store instruction is loaded from ROM, these instructions will need to access data RAM and will require a second cycle to complete the instruction. This is because data RAM must be clocked in order to ensure latched values. When the processor enters the Memory state, two important things occur: the program counter and the instruction ROM are disabled for one clock cycle to avoid multiple instructions at the same time. When the Memory state is completed, the processor then enters the Execution state on the next clock cycle to continue with the next instruction.

The Verilog code for the Logic controller is written with four always blocks: Next State, Present State, Program Counter and instruction ROM, and Output Logic. These four always blocks explicitly represent the results desired during the Present State.

The Next State is always updated on the positive edge of the clock, and cased on the Present State. It takes into account if the current instruction is a load or a store. If it is, it sets the Next State to the Memory State, otherwise, it sets the next state to the Execution State. If the Present State is the Memory State, then the Next State will again be the Execution State. On the positive edge of the clock, the Next State is always set to the Present State.

The Program Counter and instruction ROM also case on the Present State. If the Present State is the Execution State, then the opCode for a load or store is checked to see whether or not the Program Counter and instruction ROM should be disabled. If the Present State is the Memory State, then the Program Counter and instruction ROM are re-enabled on the positive edge of the clock.

The Output Logic is a combinational always block. It only changes when the opCode or the functionCode changes. These two signals together combine to determine the unique control signal outputs of every instruction. Because I-Type and J-Type instructions do not have any function codes, these instruction opCodes are checked first. If the instruction is not an I-Type or J-Type instruction, then it is check on both the opCode and the functionCode to determine which R-Type instruction is being executed.

The Program Status Register module contains a multiplexor that is also controlled by the Logic Controller. The control signal to the multiplexor, PSRsel is simply assigned to the value inputted on the Rs line as the Rs will always determine the output of the multiplexor in the Program Status Register module.

In total, the Logic Controller has twenty total control signals as follows:

1. branch: This control signal is set high when a branch instruction is being executed. It is used to combine with the PSRcond bit through an AND gate to determine whether the Program Counter should increment by the branch amount or not.
2. jump: This control signal is set high when a branch instruction is being executed. It is used to determine whether or not the Program Counter should be set to the desired jump location.
3. jumpRA: This control signal is set high when a jump to return address, or jra, instruction is being executed. It is used to determine whether or not the Program Counter should be set to the location stored in the return address register, Ra, or register 31.
4. CFWrite: This control signal is set high when an operation is performed in the ALU that will set the C or F flags of the Program Status Register. This control signal is used within the Program Status Register module to determine whether or not the Program Status Register should update the C and F flags.
5. LZNWrite: This control signal is set high when an operation is performed in the ALU that will set the L, Z, or N flags of the Program Status Register. This control signal is used within the Program Status Register module to determine whether or not the Program Status Register should update the L, Z, and N flags.
6. wbPSR: This control signal controls the wbPSR multiplexor that differentiates between the data coming from the ALU/RAM or writing back the 32-bit, zero extended condition bit output of the PSR module. If wbPSR is a 0, then the data from the ALU/RAM will pass through; if it is a 1, then the 32-bit, zero extended PSRcond bit will pass through.
7. RtSrcReg: This control signal controls the RtSrcReMUX, which differentiates between the Rdest and Rt registers to use as a source to the Register File. If RtSrcReg is a 1, then the Rdest bits are passed to use as the second source register in the RegFile. This is useful for loads and stores where RtData has to be what is contained in the register in the Rdest field of the instruction. Otherwise, for all other instructions, RtSrcReg is 0, which passes in the decoded portion of the instruction representing the Rt field.
8. wbSrc: This control signal controls the wbSrcMUX that differentiates between the data coming out of dataRAM and the result of the ALU. If the result of the ALU is desired to be written back to the destination register of the Register File, then wbSrc is set high. If the data to be written back comes from dataRAM, for load instructions only, then wbSrc is set low.
9. memSrc: This control signal controls the memSrcMUX that differentiates between the data coming from the result of the ALU, or the shifted data coming out of the shifter. If memSrc is low, then the data coming out of the shifter is passed. If memSrc is high, then the data representing the result of the ALU is passed. The output of this mux then presents itself as the address to the dataRAM module.
10. shiftSrc: This control signal controls the shiftSrcMUX that differentiates between the RtData and immediate data as the amount to shift by. If the signal is set high, then the immediate value is the value by which the shifter will shift the inputted data. If the signal is low, then the RtData is passed as the amount by which the shifter will shift the inputted data.
11. aluSrcb: This control signal controls the aluSrcbMUX that differentiates between the RtData and immediate data as the second argument to be passed into the ALU (arg b). If the signal is high, then the data from RtData will be passed as argument b for the ALU. If the signal is low, then the 32-bit immediate data will be passed as argument b for the ALU.
12. regWriteEn: This control signal determines whether or not writing back to a specified register in the Register File will occur or not. Most instructions will want to write back to the register defined in the Rdest field, but some instructions, such as a store or compare, will not want to write back to a register, and therefore, this signal will be set low for said instructions.
13. raWrite: This control signal determines whether or not the Return Address register (RA) is written back in the Register File. If the signal is asserted, then the data for the RA register is saved into the designated RA register (in the case of this 32-bit machine, it is register 31). This signal is necessary when a jal occurs and the return address must be saved.
14. shiftType: This control signal informs the shifter if the shift to be performed is a logical or arithmetic shift. If the signal is high, then an arithmetic shift is performed. If the signal is low, then a logical shift is performed.
15. memWrite: This control signal informs the dataRAM whether something is being written back to RAM or not. This signal will only be set high for a store instruction, in which case writing to dataRAM will be valid; otherwise it will be low.
16. pcEn: This control signal tells the Program Counter whether or not to increment. This signal is always kept high unless memory is being accessed, in which case it will be pulled low for one cycle to allow accessing memory without moving onto the next instruction.
17. enROM: This control signal tells the Instruction ROM whether or not to read out the data. Just as the pcEn, this signal will always by high except when accessing dataRAM is required, in which case it will be pulled low for one cycle.
18. enRAM: This control signal is always low unless a store instruction is being performed. When this signal goes high, it means that dataRAM will be enabled to write the RtData into the desired address.
19. aluop: This control signal determines the operation for the ALU to perform.
20. PSRsel: This control signal determines what PSRcond to output from the Program Status Register.

**Instruction Execution**

The following is an example that was used to test the functionality of the processor. The example that was used was to compute the first fourteen Fibonacci numbers and to store them into memory. The Fibonacci numbers were then read sequentially out of memory to show the functionally of the load instructions. This example encompasses the processor’s ability to manipulate numbers using the ALU, perform comparisons, utilize the PSR for branches, and the ability to jump to the correct location. The following is the custom assembly code that was written specifically for the way the assembler works:

//Start the Fibonacci code

Fib:

addi,r2,r0,128

addi,r3,r0,test

addi,r4,r0,1

addi,r5,r0,-1

//Enter the Fib loop

loop:

cmp,x,r3,r0

beq,end:,//Branches to the end

add,r4,r4,r5

sub,r5,r4,r5

str,r4,r2,0

addi,r2,r2,1

addi,r3,r3,-1

j,loop:

//Display what we computed

end:

addi,r6,r0,14

addi,r2,r0,128

main:

cmp,x,r6,r0

beq,finish:

ld,r7,r2,0

addi,r2,r2,1

subi,r6,r6,1

j,main:

finish:

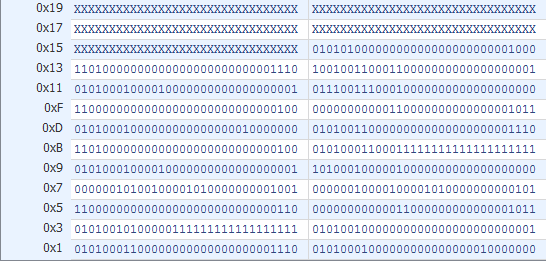
addi,r8,r0,8

//This really is the end

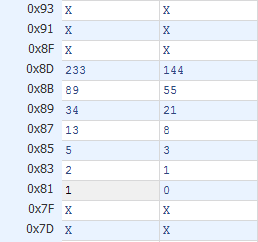
The Fibonacci assembly code was then assembled by the assembler into our binary instruction ROM .dat file. This file was then loaded into the instructionROM module of the processor for initialization.

The following explanations show the correct operation of the processor as expected.

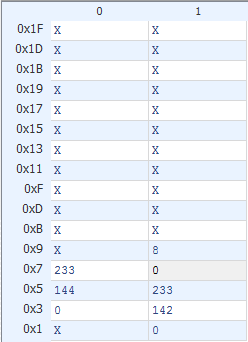
The figure below is a representation of the instructions that have been loaded into memory. The instructions begin at address 0x0 and then increment with the PC. After the processor is reset the PC will begin again at address 0x0. There were only 21 total instructions required to assemble the above assembly code, and therefore, starting at address space 0x15, the instruction is at an unknown state. This will be taken care of in the final processor design, as all memory locations need to be properly initialized.



As the application executes, each instruction is loaded from instructionROM as indicated by the Program Counter. The program counter is updated according to the necessary branches and jumps that occur in the application. As the Fibonacci numbers are calculated, they are then stored into dataRAM so that they can be used for future reference. The following figure shows the outcome of the data in dataRAM for computing the first 14 Fibonacci numbers and storing them in memory starting at location 128 (0x80) as indicated by r2 in the application.



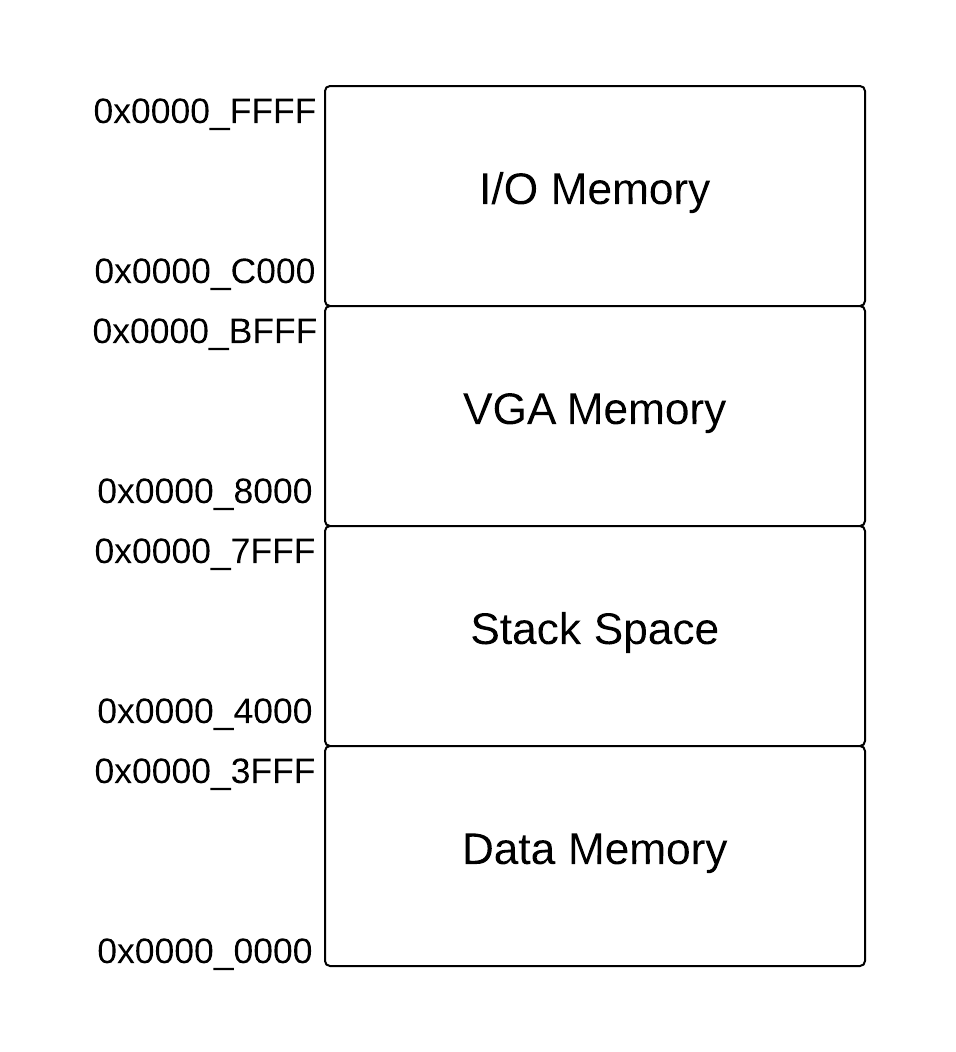
Throughout the program, the value of r2 is incremented by one preceding each of the 14 stores that occur. The figure below shows the register file at the end of execution with a value of 142 in r2, as expected. It also shows that the counter from 14 to zero that was stored in r3 decremented properly to zero. It also shows that at the end of the program, the final Fibonacci number was loaded into r7, where the 14th Fibonacci number is 233. Finally, a 8 was loaded into r8 to show the completion of the application.



**Memory Map**

In the design of the processor, it has been seen that different memory blocks have been created to keep track of instructions and data to be utilized throughout the application. It is important to know where this memory is and to keep track of it so that it isn’t corrupted on accident.

As the application will not only include instructionROM and dataRAM, but will also need to interface with Memory Mapped Input/Output (MMIO), the following map of memory has been created to keep track of where each block of ram will be located within the 64KB of memory space provided by the Sapartan 6 FPGA.



As instructionROM is a Read-Only-Memory, there is no need to include it in the Memory Map because it can only be read from, not written to. Therefore, the rest of the memory was broken up to indicate where each section can be located.

With 64KB of memory, it was decided that 20KB would be needed for the application memory, or instructionROM. Therefore, only 44KB of memory were left to be configured into this memory map.

It was decided that each section of memory should be split up equally. In this manner, a simple mux could be used, using the top two address bits to access the desired portion of memory. Now, though this is a 32-bit machine, 32-bits of addressing space is not possible in 44KB of memory. In fact, only , or 16, bits of addressing space is possible in 44KB. Therefore, the memory was access was based on using bits [15:14] of the [15:0] address accessing memory.

The lowest address in memory was reserved for dataRAM. This area will be used to store and load the computations performed by the processor. The next block was reserved for stack space. This will allow for function calls to be made within the application without overwriting other calculated data.

The top two memory portions are for the I/O, split between the VGA and other I/O. The other I/O will contain space for the four NES controllers to be used in the application, and the music generator for sound. The I/O therefore will be distinguished further by bit [14] of the address.

**Input/Output**

**NES Controller**

The Nintendo Entertainment System’s (NES) video game controller was selected as the input device for the user will be able to control their respective player. The controller uses a shift register to serially communicate which button on the controller was pressed if any. The controller requires 5 wires to operate, outlined as follows:

1. **Power**: This is a 5 Volt wire that is used to power the controller. However, because the Nexys 3 board does not have a 5 V power supply, the 3.3 V supply was tested and works well.
2. **Ground**: This wire is the common ground between the Nexys 3 and the controller.
3. **Data**: A one wire line that is used to transmit the data from the controller in a serial fashion.
4. **Latch**: A one wire line that sends out a 60 Hz pulse to initiate the data transfer from the controller.
5. **Pulse**: A one wire line that is used to capture the data from the controller.

The controller runs off of a 60 Hz latch pulse that initiates the shifting of the 8 bits of information that correspond to the 8 buttons on the controller. The latch pulse lasts for 12 μs, once completed the data for button A is captured on the negative edge of the latch. 6 μs later the pulse line gets pulled low, and begins to toggle ever 6 μs. Each time the pulse line is pulled low, the data corresponding to each button is latched within an output register. Once the data transfer is complete, the pulse line is set high until the process repeats. The following figure displays the timing diagram used for the NES controller.

The buttons on the controller are active low, therefore if a button has been pressed its corresponding bit will be 0 when it is shifted out, else it will be a 1. In the code however when the information is latched from the controller it is inverted to make the code clearer to implement.

The controller logic was implemented using an implicit finite state machine based on the count that was used to control the output’s based upon a counter because the Nexys 3 runs off of a 100 MHz clock, the counter was necessary to be able to control the timing requirements needed for successful operation. If a reset is encountered, the count is reset to 0, and the latch and pulse lines are set to low and high respectively. The counter counts from 0 to 1666667 which gives the latch pulse the ability to occur at a 60 Hz rate. Important counts for the latch and pulse lines are in the following ranges:

0-1200: In this range the latch pulse will be high, this range gives the latch’s duration of 12 μs. After the count reaches 1200 the latch line is pulled low, and is kept low until the counter resets.

1800: At this point the pulse line begins to toggle every 6 μs, each time the pulse line is pulled low, data is captured.

10201-1666667: In this range the pulse line is pulled high, and the latch line is kept low as required once the data transfer is complete.

Important count ranges for the data capture are as follows:

1200-1800: The data for button A is captured and stored in the 7th position in the array containing the information about which button had been pressed

2400-3000: The data for button B is captured and stored in the 6th position in the array containing the information about which button had been pressed

3600-4200: The data for button SELECT is captured and stored in the 55h position in the array containing the information about which button had been pressed

4800-5400: The data for button START is captured and stored in the 4th position in the array containing the information about which button had been pressed

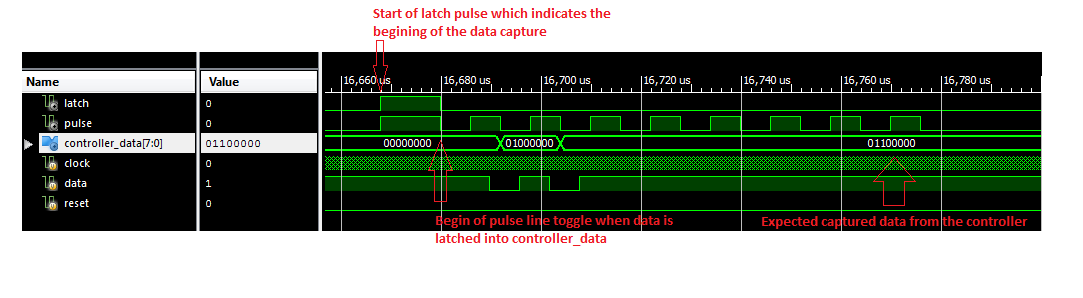
6000-6600: The data for button UP on the D-pad is captured and stored in the 3rd position in the array containing the information about which button had been pressed

7200-7800: The data for button DOWN on the D-pad is captured and stored in the 2nd position in the array containing the information about which button had been pressed

8400-9000: The data for button LEFT on the D-pad is captured and stored in the 1st position in the array containing the information about which button had been pressed

9600-10200: The data for button RIGHT on the D-pad is captured and stored in the 0th position in the array containing the information about which button had been pressed

The following annotated waveform shows the simulated behavior of the controller:



**VGA**

In order to minimize memory costs, a glyph-based display was chosen for the game. The resolution of the screen was chosen to be a standard 640x480 pixel display. The glyph size was chosen to be 8x8 pixels, based on the decision that this size would be optimal for allowing the movement of the characters in the game to be somewhat fluid, yet still reduce the memory needed to store the display. This means that the screen is divided into 8x8 “glyph locations” where glyphs can be displayed. The screen resolution can then be thought of as 80x60 glyph locations. It was found that approximately 54 glyphs would be needed for the game, though space for 64 will be allocated in memory. These glyphs are stored in a memory called the glyph library. The VGA game display was accomplished using three key Verilog modules that were implemented together in a top module. A description of the functions of each module is given below.

1. **VGA Control Module**:  
   The function of the first module is to provide a control interface for the VGA display. Two counters were implemented, one that keeps track of a horizontal count as the VGA beam moves horizontally to the right across the screen, and a vertical counter that keeps track of a horizontal count as the beam moves vertically down the screen. These counts were used to keep track of the precise timing needed to properly control and interface with the VGA display. The timing for a 25MHz pixel clock and 60 Hz screen refresh rate (a standard timing for a 640x480 pixel display) is kept track of using the two counters. The 25 MHz clock is generated in a simple always block. A 2-bit counter increases its value on every positive edge of the internal 100 MHz system clock. When the counter reaches its max value it sends out a pulse and then resets. This essentially creates a 25MHz pulse used to control the horizontal and vertical timers.  
     
   A second always block is used to describe the behavior of the two timers. If-statements are used to describe what happens when the timers are in certain key ranges. These ranges correlate to key timings associated with the VGA display. A signal, hSync, needs to be asserted low every time the beam has reached the right edge of the screen to bring it back to the left. Another signal, vSync, needs to be asserted low to bring the beam back to the top of the screen once it has reached the bottom. Since the display is not valid during the time that the beam is being moved, an output signal called “bright” was created and made to be high only when it is OK to display pixels to the screen. Also, for convenience, two more outputs are generated, called hPixel and vPixel, that represent the current horizontal and vertical position of the pixel that is currently being displayed on the screen.
2. **Character Display RAM Module**:  
   A Verilog module was created to describe a dual-ported memory used to store which glyph should be displayed at every glyph location on the screen. Each glyph stored in the glyph library was assigned an index. This index is what is stored in the dual-ported memory. The memory is dual-ported because both the VGA display and the processor need to be able to access the memory at the same time. The processor needs to access this memory so that the location of characters can be known and updated. Two memory addresses are needed as inputs, one for the processor and one for the VGA display. For convenience memory reads were made combinational so that the memory values at the input addresses are immediately output. Only the processor can write to this memory, and the memory will only get updated on the positive edge of the clock. Whether the processor is writing to or reading from this memory is determined by a 1-bit input. Each memory address contains 7 bits (the number of bits needed for all the possible glyph indexes). There are 4800 memory locations since there are 4800 glyph locations on the screen.  
     
   The VGA display is done in a unique way. The lower three bits of hPixel represent the actual horizontal pixel inside the glyph character being accessed and the lower three bits of vPixel represent the vertical pixel inside the glyph character being accessed. This means that the upper 7 bits of hPixel indicates which of the 80 horizontal glyphs in a line is being displayed. The upper 6 bits of vPixel indicate which of the 60 vertical glyphs in a vertical line is being displayed. Each successive glyph location is stored successively in memory. This means that the first 80 memory locations correspond to the glyph locations on the top horizontal line of the screen and the next 80 locations correspond to the glyph locations of those directly below the first line of horizontal glyphs.  
     
   This memory space will be initialize with a .dat file so that the when the application begins, the VGA will display the correct sequence of glyphs.
3. **Character ROM Module (Glyph library):**  
   The character ROM stores all of the glyphs. Each glyph contains 64 pixels and each pixel has 8 bits of color. This means that each memory location in the glyph library stores 8 bits of data. Similar to the layout of the character display RAM, the first 64 memory locations store the colors of the pixels in the first glyph, then the next 64 memory locations contain the colors of the pixels for the second glyph and so on. Within a glyph the first horizontal line of pixels correspond to the first 8 memory locations, then the next horizontal line is stored and so on. The glyph index from the character display RAM indicates which glyph is currently being displayed. The lower three bits of vPixel and hPixel indicate which pixel in the current glyph is being displayed.  
     
   This memory space will be initialized with a .dat file that contains the 8-bit RGB encoding of each 64 pixels in every glyph that will be used.
4. **VGA Display Module**:  
   A top module is used to connect the VGA controller to the RAM and the ROM. An assign statement is used to display the color of the current pixel stored in ROM if bright is high (it is OK to display a pixel) or to display black if bright is low.