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 player is hit by a laser, 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 player has been hit by a laser.

**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**

The project will be broken up into four parts with both lead roles and backup roles for each one as follows:

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

**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
8. wbSrc
9. memSrc
10. shiftSrc
11. aluSrcb
12. regWriteEn
13. raWrite
14. shiftType
15. memWrite
16. pcEn
17. enROM
18. enRAM
19. aluop
20. PSRsel