

Computer Design Laboratory Project

Levi Balling Robert Christensen T. James Lewis

December 2011

Abstract

We implemented a 16-bit computer with VGA graphics for output, a NES gamepad used for input, and an RS-232 type serial controller used for communications. The software developed for this computer is a maze game in which players maneuver through a maze trying to reach the exit first. The players play on two machines, each with its own monitor, and communication between the machines is over a serial connection.

1 Introduction

Our goal was to produce a computer platform capable of running a game with the following features: Levels that were larger than the screen and which scrolled when the player moved near an edge, two players on different machines playing against each other with interaction, and using an NES game pad for user input. To allow large levels we needed a mechanism to store the level information compactly, as the block memory we were using was not large enough to store bitmaps as large as the levels. To allow linked play between two machines we needed to implement some form of communication between them. To use NES game pads for input we needed to research how they transmitted the input to the machine and device and interface to work with them.

The overall organization of the system can be seen in figure 1 we have a main memory module which is a two port block memory that both the CPU and VGA controller have access to. The Gamepad Controller module is connected to the CPU, and the Serial Controller is also connected to the CPU. The VGA Controller has pin-outs to the VGA port on the board, the gamepad controller has pin-outs to the gamepad, and the serial controller has pin-outs to the serial port.

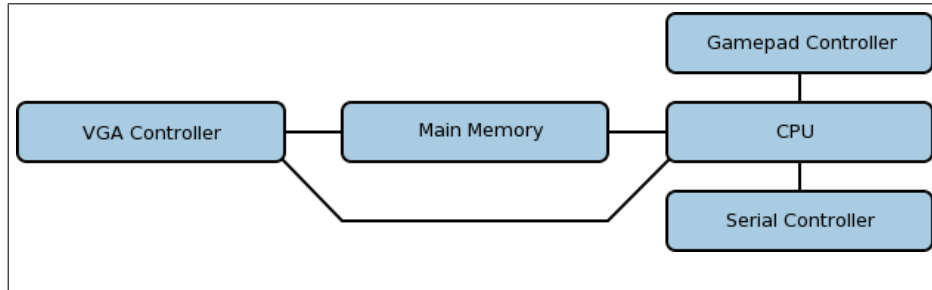


Figure 1: Overall Organization

A lot of the work required to render a game level to the screen is taken care of by hardware in the VGA Controller. There is a section in main memory where the map to be drawn is stored as a series of tile numbers, the VGA Controller reads these tile numbers from main memory, and for each tile number looks in its internal memory for the pixel map corresponding to the tile number and draws the pixels in the right section of the screen. The CPU has two registers which

are directly wired to the VGA controller that tell the VGA controller where to look in main memory for the tile to be placed at the upper left corner of the screen and also what the row offset is. This makes the code for scrolling the screen to show only a section of the entire map as simple as setting the address for the upper left corner of the screen to a different location, rather than having the CPU rewrite the entire map area with new data to scroll.

2 CPU Design

Our CPU is composed of a control module, register file, ALU, instruction register, program counter, processor status register, VGA start address register, VGA row length register, system clock module, a two input 16 bit multiplexor and a four input 16 bit multiplexor. (See figure 2)

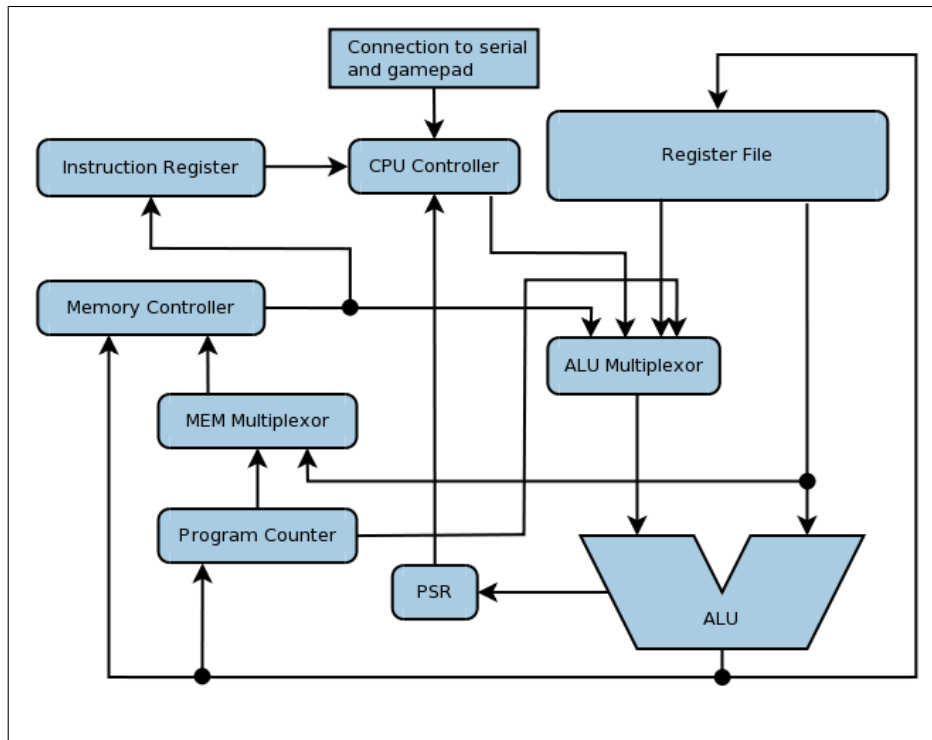


Figure 2: CPU Design

The CPU control module is a state machine with two states: fetch and execute. In the fetch state the memory address contained in the program counter is accessed and stored in the instruction register. In the execute state the control unit sets the control lines to all modules in order to execute the given instruction. The memory clock is inverted with respect to the CPU clock, and since the memory can be clocked at up to 800Mhz while our CPU is clocked at

25Mhz memory accesses are completed within a single CPU cycle.

The execute state in the CPU control module is entirely composed of combinational logic which for every possible instruction sets all control wires (mux select, register enable, ALU opcode, and select lines for the register file to execute the instruction.

2.1 Instructions Implemented

Class	Instruction	15:12	11:8	7:4	3:0	function
ALU	AND	0000	0000	Src	Dst	Bitwise AND
	OR	0000	0010	Src	Dst	Bitwise OR
	XOR	0000	0011	Src	Dst	Bitwise XOR
	NOT	0000	0100	Src	Dst	Bitwise NOT
	ADD	0000	0101	Src	Dst	Addition
	ADDU	0000	0110	Src	Dst	Unsigned addition
	ALSH	0000	0111	Src	Dst	Arithmetic left shift
	ARSH	0000	1000	Src	Dst	Arithmetic right shift
	SUB	0000	1001	Src	Dst	Subtraction
	LSH	0000	1100	Src	Dst	Left shift
	MOV	0000	1101	Src	Dst	Move value
	RSH	0000	1110	Src	Dst	Right shift
Memory	LOAD	0100	0000	Addr	Dst	Load data from address in Addr into Src
	STORE	0100	0001	Addr	Src	Store Src to address in Addr
Immediate	ADDI	0100	Immediate		Dst	Add immediate
	ADDIU	0110	Immediate		Dst	Add immediate unsigned
	MOVIU	0111	Immediate		Dst	Move immediate upper
	MOVI	1000	Immediate		Dst	Move immediate
	SUBI	1001	Immediate		Dst	Subtract immediate
	CMPI	1011	Immediate		B	Compare immediate with B
Branch	BGE	0001	0000	—	—	Branch on greater than
	BHS	0001	0001	—	—	Branch on higher or same
	BEQ	0001	0010	—	—	Branch on equal
	BLT	0001	0011	—	—	Branch on less than
	BLS	0001	0100	—	—	Branch on lower or same

Continued from previous page						
Class	Instruction	15:12	11:8	7:4	3:0	function
Jump Cond	JGE	0001	0000	Offset		Jump by offset on greater than
	JHS	0001	0001	Offset		Jump by offset on higher or same
	JEQ	0001	0010	Offset		Jump by offset on equal
	JLT	0001	0011	Offset		Jump by offset on less than
	JLS	0001	0100	Offset		Jump by offset on lower or same
	JOFFSET	0001	0101	Offset		Jump by offset no condition
Special	NOP	0010	—	—	—	No operation
	CMP	0010	—	A	B	Compare A and B
	JUMP	1100	0000	Src	—	Jump to address in Src
	STOREPC	1100	0001	—	Dst	Store program counter to Dst
	SETBEGINVGA	1101	0000	Src	—	Set value in begin VGA register
	SETROWVGA	1101	0001	Src	—	Set value in VGA row length register
	READSERIAL	1101	0010	—	Dst	Attempt to read data from serial
	WRITESERIAL	1101	0011	Src	—	Send data to serial
	READGAMEPAD	1101	0100	—	Dst	Store gamepad input to Dst
	CLOCK	1101	0101	—	Dst	Store value from system clock

2.2 Implementations of instructions

The ALU instructions are implemented simply by setting the register file to enable writing to Dst and to have Src and Dst be the outputs of the register file. The multiplexor between the register file and ALU is set to pass through the input from the register file and the program counter is incremented by one.

To perform a LOAD instruction the register file is set so the B output is the Addr register and the Src register is the A output. The multiplexor between register file and ALU is set to pass through the memory controller's data output. The multiplexor in front of the address of the memory controller is set to pass through the B output of the register file. The ALU is set to pass through input A unchanged. The register file has write enabled for register Dst. The memory controller is set to read and the program counter is set to increment. For a STORE instruction the register file has write disabled, the multiplexor between the register file and ALU passes through A from register file, and the Memory controller is set to write.

For all Immediate instructions the proper Opcode is sent to the ALU and the multiplexor between the register file and ALU passes through the immediate value, which the CPU control module sets as the immediate value in the instruc-

tion (sign extended). All other control signals are set as for ALU instructions.

For Branch instructions the CPU control module examines the flags in the processor status register and if the condition is met increments the program counter by one, if not met the program counter is incremented by two. This means that the instruction following the branch instruction will only execute if the condition is met. We didn't actually use these instructions in our code, as the conditional jumps by an offset were more convenient.

For conditional jumps the CPU control module examines the flags in the processor status register and if the condition is true increments the program counter by the specified offset and otherwise only by one.

A NOP instruction has all registers disabled for writing, so nothing happens except the program counter being incremented.

A CMP instruction compares the values in two registers by performing a subtraction and setting the processor status register flags, but not storing the result.

A JUMP sets the program counter to the value in the specified register.

The SETBEGINVGA instruction sets the value in the VGA Begin register to that in the Src Register. The SETROWVGA does the same for the VGA row length register.

READSERIAL attempts to read from the serial controller's buffer by asserting the read control line to the serial controller. If there is not valid data to be read then the program counter is incremented by one and the registers disabled for writing. If there is valid data the CPU control module puts the data as the immediate value which gets passed through the ALU and written to the Dst register. In this case the program counter is incremented by two. This branching behavior is how a running program can determine whether there is valid data to read in.

WRITESERIAL sends data from the Src register to the serial controller buffer by passing it through the ALU. The CPU control module asserts the send line to the serial controller at the same time.

READGAMEPAD sends

3 VGA Controller

We used a 640×480 VGA display with a 25 Mhz clock. This was done with the use of sync signals, and memory manipulation. The 640×480 pixel display was broken up into 20×15 blocks on the screen. This allowed us to conserve memory and still have meaningful graphics. The graphics used 3 bit colors, and we were able to display 8 different colors. Using different patterns with these colors we were able to display 14 32×32 pixel images.

3.1 Sync Signals

For the monitor to recognize which pixel it is on, it needs to receive 2 signals, Vsync and Hsync. These signals will be sent on specific patterns because it

generates different resolutions. We used a 25 Mhz clock input to generate the different signals. While we generated the signals, we kept track of which columns and rows the VGA was on. We used the columns and rows to find out which group of pixels it was on. As well as the individual sub pixels that it is on in the 32×32 block of pixels.

3.2 VGA Memory

We had organized our memory to have 2 separate memory blocks. Main memory used a 2 port 14 bit addressable memory block containing 16 bit words. Our VGA memory contained a 14 bit addressable single port containing 9 bit words. One of the 9 bit words contains 3 separate pixels. We used each memory value twice, so we display the value of 3 bits for 2 pixels. We have an input to the bit generator where the starting address and row size of the image are. This way we can fetch the entire cell in main memory that we would like to display.

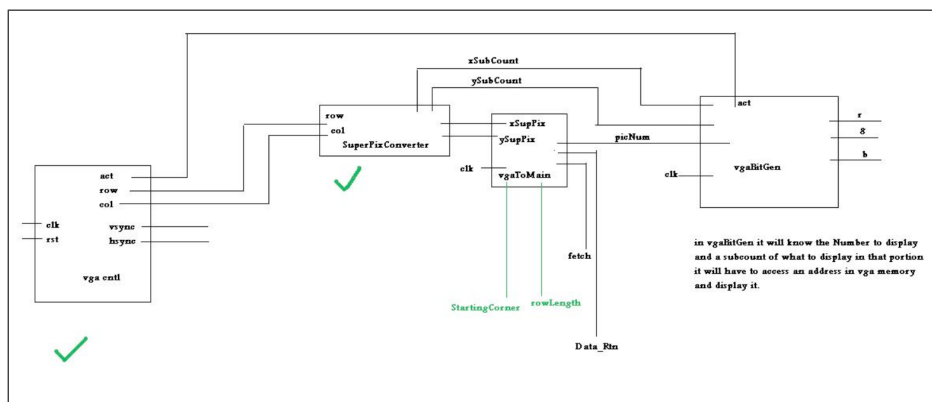


Figure 3:

In main memory we have an area specially set aside just for our levels. These memory words contain a number from 0 to 14. These words are related to the image that we want to display on that portion of the screen. The VGA controller will read from main memory, find out what should be displayed on the block of the screen based on what number it is, and then go and display all of the values on the screen based on the number from 0 to 14. From that number it will find the correct address in VGA memory that it should display for that pixel.

4 NES Controller

For our game system we used a classic Nintendo Controller (figure 4) with specifications found from the web site of another student project using NES controllers [2]. This device required 5 lines to the controller: Power, Ground, Latch, Pulse, and Data. The timing input for our controller was 25 Mhz, the module throttle was around 1Mhz and we achieved around 100 samples per

second. This allowed us to read the input, detect whether the values were high or low and store it in a register.



Figure 4: Classic Nintendo Gaming Controller

For us to read the data from the controller we had to generate two signals (Figure 5). The Latch signal will be generated every time one wishes to read the Data signal. The Latch pulse width is 2 times the length of the Pulse signals pulse width. The Latch initiates the data read and the pulse initiates each new chunk of data. While the Latch is high the Data line will provide the signal for the A button 5V for pressed, and 0V for not pressed. After the Latch signal is sent, the Pulse signal will need to send 7 pulses for the rest of the buttons. After that signal is sent, the data can pause until the controller sends the next Latch signal. In our case

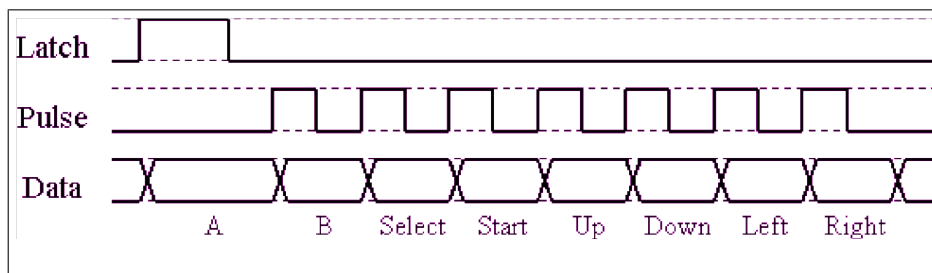


Figure 5:

We were able to test if the controller would work with 3.3V logic versus the 5.0V logic that it states it needs. We established that it is compatible with both logic levels, and that the Vt levels were probably around 2.5V.

5 Serial Controller

6 Assembler

The assembler architecture was heavily influenced by an assignment given in a hardware design lab at a different university. The assignment provided sugges-

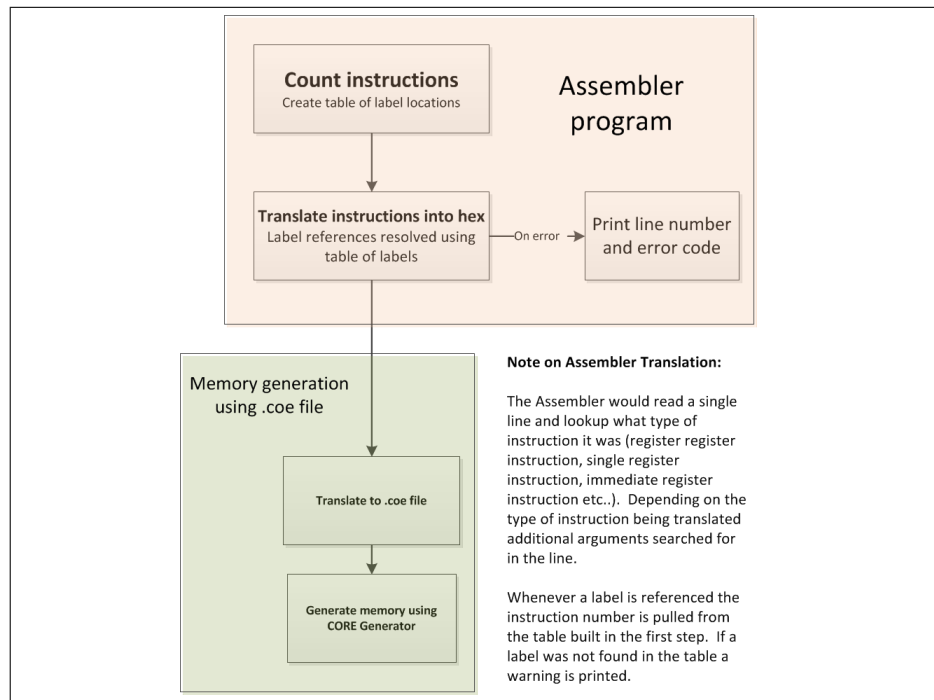


Figure 6: Tool chain for assembling and loading a program

tions for writing a simple assembler using the C programming language. These hints were considered in the development of the assembler that was used in this project.

The assembler was written using the C++, as opposed to C, in order to use the standard template library. A map was used to store the table of labels and their locations in code. The program was written to be compiled using the g++ compiler.

Figure 7 shows how the assembler tool chain works. The assembly file is scanned twice. The first pass fills the table of label locations. In order to do this each instruction must be counted. Pseudo code instructions are counted as however many assembly instructions it will take in the output. By linearly counting the number of previous memory locations used each label can be associated with the correct memory location in the output.

The second pass though the input translates each line into the appropriate hex value for the instruction. The instruction could be a pseudo instruction, which is split into multiple assembly instructions; a fill value, where the assembler will place a specific value at that memory location; or an assembly instruction. The only pseudo instruction used in the current implementation was used to load label locations into a specific register. Since a single move instruction only load 8 bits into a register, and only the assembler knows the value of 16 bit label reference, this pseudo instruction was created. Whenever a label reference

is found in any instruction, the value of the label is pulled from the table and substituted appropriately.

When an assembly instruction is found, the assembler does a look-up for what type of instruction is being used. Seven unique instruction types were used in the CPU implementation. Doing a look-up would tell the assembler how many registers and immediate value to expect following the instruction, and how to pack the bits for the output. Anytime the program found something unexpected it printed where the error was encountered and stopped the assembly process.

Compiler options were created to print extra information to help debug assembly programs for the CPU. If the assembler was built using the special debug flags additional information was reported to the user. As bugs were found in assembly programs the assembler was modified to help detect future occurrences of the error. However, the assembler was desired to be robust and accepting of programmer desired, so warning were issued for common problems rather than errors.

7 Software

A coding standard was developed for the CPU. The following were explicitly defined in the standard: save registers, temporary registers, register use for passing information to methods, stack usage, global variables, and naming of labels. The standard was designed to increase readability and consistency of assembly code. This standard was found to increase development speed and speed of bug fixes.

The game that was developed for the CPU was a simple multiplayer maze puzzle game. The goal of the game is to move the player to a goal location. The game supports movable blocks, which could be used to cover lava squares to create a bridge, or the blocks could be used to block the path of the opposing player. The movable blocks created a variety of entertaining gameplay.

The architecture of the multiplayer game was designed to be simple. No error correction or special timing synchronization was added. This architecture made the software easy to write to allow for either single or multiplayer gameplay using the same excitable. The basic flow of the game is shown in fig. 7.

When the game was played as a multiplayer game, one player would move and send the its movement to the other device. Following its movement it would enter a waiting state for 250ms. If data was received on the serial line the non-local player would move. The movement of each player was offset by 125ms. By offsetting each player movement, no turn priority would need to be programmed, since only one player will be moving at a given time. This design decision was made with the assumption that a game would not last very long, both boards run at nearly the same clock speed, and all data sent across the serial line would be received by the other board.

Multiplayer functionality was developed using this method. However, the two games would quickly become unsynchronized. It was found that a significant number of writes to the serial line were never received by the other board. The

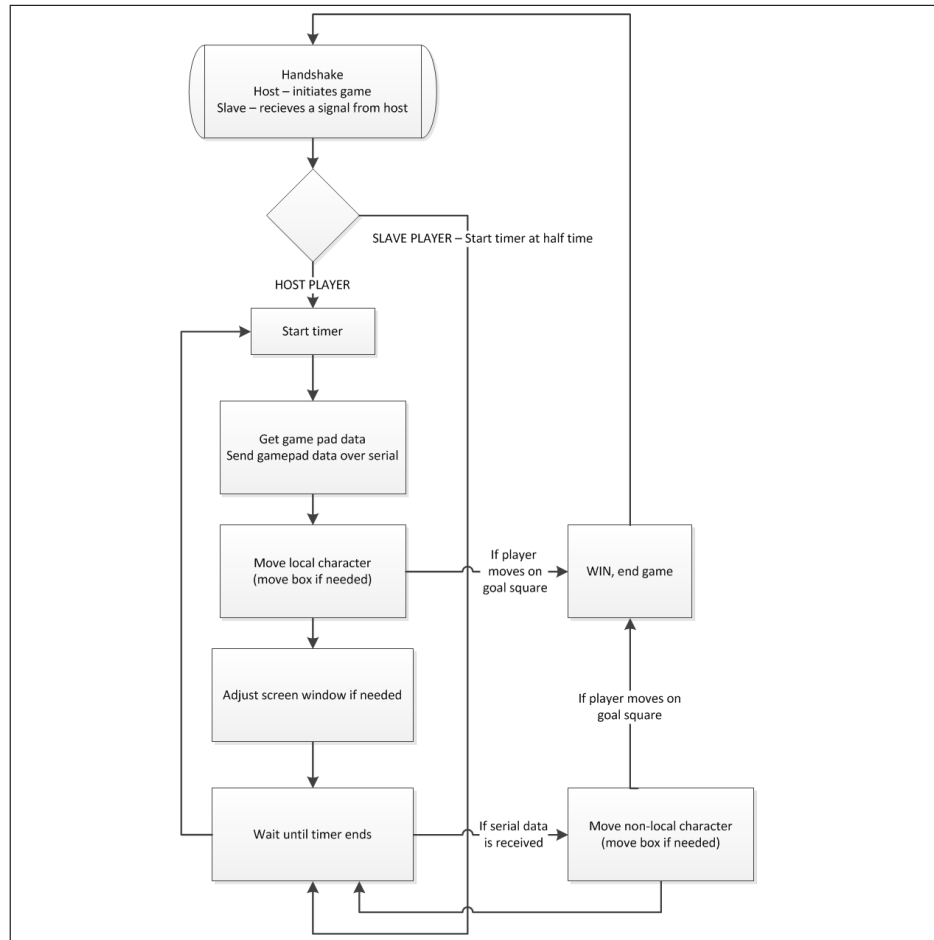


Figure 7: Progression of states in multiplier maze game

assumption that all data sent across the serial line would be received by the other board was not true, so this architecture would not work for multiplayer.

A redesign of the multiplayer portion of the code was started that would detect and resend lost packets was started. The design of the patch was developed, but the programming was not finished due to lack of time.

8 System Integration

9 Conclusions and Further Work

10 Individual Contributions

10.1 Levi Balling

10.2 Robert Christensen

10.3 T. James Lewis

References

- [1] Serial Interface (RS-232), <http://fpga4fun.com/SerialInterface.html>
- [2] igradeplay, <http://web.mit.edu/6.111/www/s2004/PROJECTS/2/game.htm>