Pac-man: Revisited

Ari Ferro, Jeff Thacker, Todd Cronin, and Zachary Wilcox

***Abstract*—*Pac-man* was an influence game. We decided to recreate Pac-Man with our own hardware and software approach. We recreated the hardware using a CR-16 processor implemented with Verilog on a Spartan-3E FPGA board. The software was done using the CR-16 ISA assembly. We successfully created 97% of the game. Given more time, we would have fully recreated the game.**

***Index Terms*—FPGA, *Pac-man*, CR-16**

I. Introduction

THIS project was to demonstrate our computer engineering knowledge up to this point in our academic career. We designed a processor based on the CR16 ISA created for ECE/CS 3710 at the University of Utah (see Appendix A). It was designed for the Spartan-3E FPGA using the Verilog hardware description language. The ultimate goal of this project was to create a video game which demonstrated interactivity between a user and our computer system.

In order to achieve interactivity the processor interfaced with several different input/output (I/O) devices. The I/O for the project consisted of VGA for video, a NES controller for character movement, a MP3 trigger for game sound, and a memory-mapped controller which linked the processor, memory, and I/O. We created an assembler to translate an English representation of the ISA into the machine language. This allowed us to build a larger application without manually encoding the assembly.

We thought *Pac-man* would be the perfect game to implement on our processor due to its iconic status and relative simplicity. The graphics in *Pac-man* were conducive to a glyph-based graphical implementation and all of the movement patterns in *Pac-man* were very simple.

The visual graphics, sounds, and inputs were completed by the demonstration of the project. At the demonstration we had not completed the end-game conditions or the ability for the player to eat ghosts and vice versa.

II. BackGround

Pac-man was first released in May 22, 1980 by Namco, a japanese based arcade company [1]. *Pac-man* is often credited with being the landmark video game in history. It is one of the most famous arcade games of all time, having generated more than $2.5 billion in quarters by the 1990s [2]. In order to keep the old-school, retro feel the project was designed to use the controller that came with the original Nintendo Entertainment System (NES) in 1985 [3]. The NES controller had basic directional input which met the requirements of *Pac-man*. We also used eight color VGA graphics and blocky glyphs to recreate the retro-feel of *Pac-man*.

Ari Ferro is with the School of Computing, University of Utah, Salt Lake City, USA, e-mail: ari.ferro@gmail.com.

Jeff Thacker is with the School of Computing, University of Utah, Salt Lake City, USA, e-mail: tyrsis00@gmail.com.

Todd Cronin is with the Department of Electrical Engineering, University of Utah, Salt Lake City, USA, e-mail: cronin@eng.utah.edu.

Zachary Wilcox is with the Department of Electrical Engineering, University of Utah, Salt Lake City, USA, e-mail: zachary.wilcox@gmail.com.

III. Processor

The processor has a 50 Mhz clock at 3 CPI and 32Kb of direct addressed ram. The processor has a 16 bit datapath and a 16 bit instruction format. The set of supported instructions and the instruction format is listed in (Appendix A). The control logic in the processor is run by a finite state machine. The processor reads and writes to the block ram module on the Spartan-3E board.

A. Memory

The processor uses 2^{14} words of block ram (14 bit address). Memory is word-addressed and all words are 16 bits. The processor has a 16 bit addressable memory space. Block ram is accessed with the lower order 14 bits, otherwise, when the higher order bits are set, memory mapped I/O is used. The last 2120 addresses of the block ram are used as the frame buffer, from 14263 to 16383. Writing data to this area of ram will display a corresponding glyph on the screen.

The processor executes all instructions in three clock cycles. Every instruction executed has an instruction fetch, register fetch and execution stage. The control logic of the processor is a finite state machine, with the same instruction and register fetch states for every instruction, but a separate execution state for every instruction type.

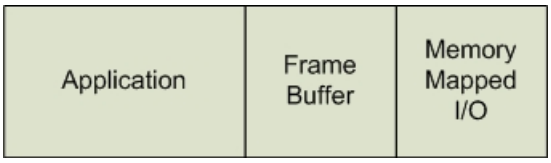


Figure 1: Memory Hierarchy

B. Control Logic

During the instruction fetch state, memory is read from the address specified by the program counter register. The result is then held in the instruction register during the register/decode and execution states. The state after the instruction is fetched decodes the opcode to determine what the next state will be. During this state, the register output in the datapath begins to output the correct values, which become available during the execution state. The processor then advances to the execution state that

The instruction is performed by setting the control logic appropriately such that the correct result is ready for writeback

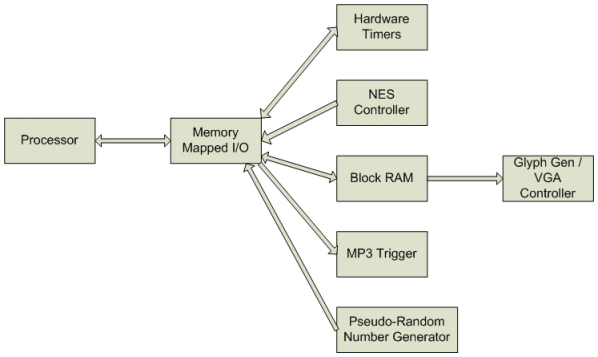


Figure 2: Pacman System Block Diagram

to the register file when the next clock cycle occurs. If it was not an instruction that changes a register, the writeback does not occur. Branch and jump instructions change the program counter to the computed branch address in the execution state as well. If the instruction does not alter the program counter, it increments normally. The processor then advances into the instruction fetch state, with the program counter pointing to the next instruction in program execution.

C. Branches and Jumps

The processor branches and jumps based on the values stored in a special processor state register. The processor state register is set after a CMP instruction or after certain arithmetic operations like add/subtract. The processor branches if the processor state register is set according to the condition code specified. A table of condition codes is included in the ISA document [4]. The processor can branch from -128 to 127 lines from the current program counter (8 bit two's complement immediate value.) The processor can jump to any instruction in memory by jumping to the address contained in a register.

IV. Input/Output

Our design incorporated many different I/O devices. The I/O devices either interacted with our user, like the NES controller and MP3 trigger or they interacted with our application, such as our pseudo random number generator and millisecond timer. Our design used port-mapped I/O between our processor and other devices. The design utilized VGA graphics with a glyph based scheme.

A. NES Controller

The NES controller communicates with our computer system with a serial and asynchronous protocol [5]. Our system sends two signals to the controller: latch and pulse. The latch goes high for 12 μs and then remains low for the rest of the cycle. Once the latch goes low, the Pulse signal is activated with a 50% duty cycle with a width of 6μs. There are a total of seven pulses for a cycle. The controller cycle has a frequency of 60 Hz. The external controller uses these signals to know which button to sample. Figure 1 shows the order in which the button's are sampled. The data line from the controller to our system goes low for 12 μs during it's sample time.

To prevent timing issues between the time our processor read the controller, and from when the controller sent the low signal for the button, we saved the data in a register. This helped with the smooth movement of our *Pac-man*. The controller was connected to the computer system by splicing the wires and attaching it to the FPGA connector board.

The NES Controller controller had a state for each on and off pulse, a state for the latch signal, and a idle. A clock divider was used to kick out of the idle state. A clock counter was used to time how long it was in it's present state. Once the NES Controller controller finished going through each of its states, it would wait in the idle state until it received another enable signal from the clock divider.

B. MP3 Trigger

The MP3 Trigger was a separate decoder circuit which played MP3 formatted audio files. The MP3 Trigger could either be interfaced by using a UART or 18 triggers. The triggers were each an active low data pin. Our design used the triggers just because they were simple to use. Our controller for the MP3 Trigger sent a high signal to all 18 triggers. The controller accepted a 16 bit unsigned number. It would send an active low signal to the appropriate MP3 Trigger. The controller ignored any signals sent to it above 18. The tracks on the MP3 trigger are as follows:

- Track 1: Openning Song
- Track 2: "Waka Waka"
- Track 3: Siren
- Track 4: *Pac-man* dies
- Track 5: *Pac-man* eats a cherry
- Track 6: Intermission.

We also wrote an initialization file which added functionality to the triggers which the device does not natively support. They are as follows:

- Track 12: Next track
- Track 13: Play random
- Track 14: Play previous
- Track 15: Start last played
- Track 16: Stop current track
- Track 17: Volume down
- Track 18: Volume up

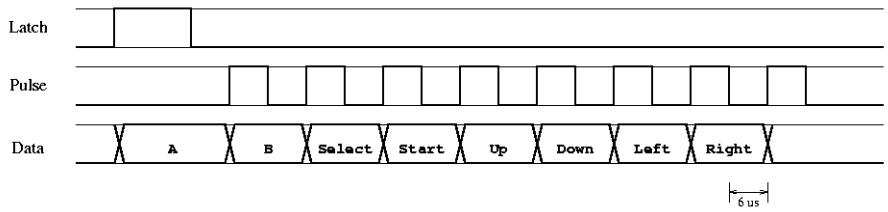


Figure 3: NES Controller Data [5].

C. Millisecond Timer

Our application needed to keep track of how often to do certain things. For example, we only wanted to move each character every so often. Our processor would loop through our code faster than we wanted to do a certain event; this lead to our characters moving faster than we wanted. We also wanted different events to occur at different intervals. We did not want the ghosts to move at the same speed as Pac-Man or Pac-Man to move as fast has his mouth movement. We solved this by creating multiple hardware timer.

Our hardware timer took a 16 bit unsigned integer value on one clock cycle. It would return a ready signal once the integer value in milliseconds has surpassed. The ready signal would stay high until another integer value was passed to the timer or upon reset. The integer value passed to the timer was saved into a register on the same clock cycle. The next clock cycle, the timer would do the appropriate math to figure out how long it would need to count down.

$$Cycles\ To\ Count = Time * Clock\ Rate - 2$$

Our clock rate was 50 MHz and the time was the millisecond value the timer was going to count down to. The timer would then start counting each clock cycle. The data signal was raised once it received the correct number of clock cycles. The -2 comes from the fact it takes two cycles before the counter starts. This is to increase the accuracy of the timer.

D. Pseudo Random Generator

The ghost characters moved in a random fashion. Our design used a pseudo random generator in order to accomplish this. The random generator generated a number between 0 and 3. Each of these numbers represented a different direction: up, down, left, right. The generator was just a basic two bit counter. It created a random affect when being accessed due to the application sampling it at different times and different inputs would change the time each character would access the generator.

V. Assembler

We created an assembler program to convert our assembly code into binary instructions for use in our processor. The assembler was written in C# using Microsoft Visual Studio. It provided a graphical user interface and was programmed to support branch and jump instructions using labels and incorporated error checking.

A. GUI Interface

Our assembler was published as a Windows 32 Forms application. The GUI allowed us to choose source and destination files and gave lists of assembly instructions with corresponding binary instructions. It also showed the output instructions in hexadecimal format and listed all labels with their addresses to assist in the testing and debugging of the application.

Initially, it was made using Visual Studio 2012. While using the Junior Hardware Lab computers to do application programming and hardware development/testing, we found that the assembler was not compatible with Windows XP as it used .NET 4.5. The project was then re-created in visual studio 2010.

B. Label Support and Branch Offset Calculation

We added support for assembly labels. The assembler makes an initial pass through the entire assembly file, creating a list of all labels and their memory addresses. Using this, we were able to implement pseudo instructions branches. Standard branch instructions were allowed to specify an offset using an 8-bit two's complement immediate as in the CR16 ISA. Alternatively, a branch instruction could include a label and the assembler would calculate the 8-bit immediate offset and output a binary instruction using the calculated value.

C. Jump to Label Support

Pseudo jump instructions were implemented as well. JAL and Jump conditional instructions were done using a slightly modified format (ie, JUC r0 loop5). Using the same idea as branches, the assembler would use the list of labels to calculate the jump address, then output three instructions to implement the jump. The specified register is used to create the jump address. The first expanded instruction was an LUI to the given register with the upper bits of the calculated memory address. Then an ORI instruction to the same register to set the lower bits of the memory address. Finally a Jump instruction (conditional or JAL) using the register which then contained the memory address of the given label.

D. Error Checking

Another helpful addition to our assembler is the error checking. Immediate values were checked to ensure they were within the correct boundaries for the given instructions. Duplicate labels, incorrect instructions or instruction formatting, etc were all output as errors in the binary file. If errors were encountered during file conversion, a message box was

also opened after conversion showing all errors with their corresponding source file line numbers. This error report could then be saved to a log file by the user.

VI. Application

Our processor implements the results of our own collaboration and creativity in the form of the classic arcade game, *Pac-man*. The game allows a user to move a character, *Pac-man*, around a map of walled corridors or allies using the directional pad on a NES controller. The user's goal is to help *Pac-man* escape from 4 oddly behaving ghosts who have the potential to kill him. Also he must collect as many 'pills' as he can, both large and small.

The original game allows for a bit more complexity than we implemented regarding *Pac-man*'s mission and environment. In the original game we find that ghosts move based upon different algorithms using *Pac-man*'s current location as a variable. We also see ghosts become edible when *Pac-man* eats a large pill. Other environmental features include ghost behavioral changes based upon the number of small pills eaten by *Pac-man* and objects appearing throughout the level such as cherries, bananas, etc.

The version we implemented did not, in the end, allow for consumption of the large pill to render the ghosts edible, nor did it allow for such fluid ghost movements or latent fruits. However, we were able to determine an alternate approach to ghost movement allowing them to be dependent a pseudo-random direction generator.

Basic Pseudo Strategy :

- Start Menu (Initial splash screen where we wait for user to press 'start' on the NES controller)
- Start Game (Once the user presses 'start' we initialize the map and enter a game loop)
- Game Loop (Update *Pac-man* and ghost states, then draw their new states)
- States include glyph and location.

Figure 4: Application Memory Usage

As stated previously, *Pac-man* and ghost movements are dependent upon user input and a random direction generator, respectively. In Game Loop we use a series of finite state machines to update two state variables associated with each character, location and glyph. These variables are then stored in memory where another function within Game Loop reads them from memory and draws the characters in their updated states, perhaps in new locations as different glyphs.

As character states update, the user watches them move 3 pixels at a time. The movement is simulated using a series of 4 different states for each direction a character moves and controlling the rate at which they change with various hardware timers. The hardware timers take a timer length in ms and return a 1 when they are finished counting, 0 otherwise. With multiple timers we can control the rate of change of 2 aspects of one character. For example, *Pac-man* opens and closes his mouth while he moves around the map. The timer for his mouth needs to be much slower than his movement around the map in order for the user to perceive the feature and so that *Pac-man* can comfortably escape and chase ghosts. Had we found the time we also would have used multiple timers to control the rate at which the ghosts flash apart from how fast they move during their 'edible' state.

Because of time, we were not able to implement an 'end of game' state. This came as a result of not successfully implementing the states in which *Pac-man* dies or all of the pills are eaten. Instead, what you found in our final product was an invincible *Pac-man* with the ability to pass through ghosts and not die, and pills which remained on the map despite *Pac-man*'s passing over them. It made for fun, easy, and stress free movements through the map as well as angry traditionalists.

Assembly code was used to implement this flow of logic. The final application code document contains over 8000 lines of code. For a more in-depth understanding of how the code works refer to the .asm file [6]. The code is well commented and shows how we implemented the basic pseudo strategy.

The biggest problems we faced came in the end as we put all portions of code together. The difficulty arose in debugging. It took too long to reprogram the board to test a new iteration of the application effectively. However, only a few functionalities failed in all 8000 lines. With more time we could have remedied it. Nonetheless, we debugged as much as we could until we were out of time. Our ratio of successful debugging to time spent performing such was very poor. This was due to the fact that our entire Xilinx project required synthesis in order to test our updated code. Due to the size of not only the .asm file, but also the processor, we found ourselves waiting for about 20 minutes on average each time.

VII. Programming Model

The processor design is a traditional sequential computer and uses the von Neumann programmers model. Code for the processor is written directly in assembly language with a few minor exceptions. The assembler is designed to support branches/jumps to labels in the assembly code. Thus the programmer does not need to know branch offsets and jump addresses – the assembler computes them automatically. There is special syntax for jumping which gets decomposed into three instructions that create the jump address then actually jump to it. The programmer must specify which register will be used to create the jump address. Additionally, the programmer needs to be aware that the Jump and Link (JAL) instruction uses the r15 register to save the return address. Consecutive JAL instructions need to preserve the old return address.

In order to write software for this processor one must consider that the graphics are based on glyphs. The glyphs are part of the hardware design and were developed for the *Pac-man* game. In order to make a different program, different glyphs need to be be created. The new glyphs can be made by replacing the glyphgen.v file in the Xilinx project folder with the glyphgen.v file that is output from the im2glyph script.

The im2glyph script requires 12x12 pixel glyphs. It calculates binary color values by rounding each pixel of each channel with a value above 127 to 1, and otherwise to 0. To ensure expected colors, jpgs should be made with RGB values of only 255 or 0 and saved without compression.

Once the jpgs are assembled into a single folder called JPEG2, the script can be run from the same directory the folder of jpgs is in, and a new glyphgen.v file is generated. This new .v file should be copied to the xilinx project folder. The glyph number associations that the project uses are enumerated in the parameter list of the glyphgen.v file. The glyph numbers are necessary to display graphics on the screen. Once the glyphs for the target application are generated, they can be displayed on the screen by writing glyph numbers to any of the locations in the frame buffer.

A. Glyph Solution

One of the interesting features of our project was the solution to creating the numerous glyphs and to create the game board. The tasks would have required tedious data manipulation, and in the case of initializing the *Pac-man* game board, would have taken hundreds of lines of assembly code. Instead, we opted to write a matlab script that streamlined the creation of both the glyphs and the game board.

The actual glyphs are coded into the hardware design of the processor. Specifically there is a standalone 'glyphgen' module that takes in a glyph number and row, and outputs red green and blue bits that represent the row of the glyph with the specified number. The game used hundreds of unique glyphs to draw different shaped walls, animate *Pac-man* and the ghosts, smooth the movement of *Pac-man* and the ghosts, and to write letters on the screen. Hand coding all of these glyphs into a verilog module would have been impossibly tedious. Our solution to this was a matlab script.

The matlab script looks for all jpeg files in a directory relative to where it is run, and outputs a syntactically correct verilog module. It looks for jpgs in a directory labeled JPEG2, and for every jpg file it encounters, it checks if it is 12x12 pixels. The filename of each jpg is used as a parameter in the verilog file. The parameters are used to write a nested case statement. The case statement examines the glyph number input and finds the correct glyph. Another case statement examines the row and outputs the binary pixel values of each color channel. To write this case statement, nested for loops were used to iterated over every row of pixels in each jpeg. For each jpg, the script computes the binary values of each of the red, green, and blue color channels. For each row in the jpg, it writes the values of each color channel.

To preload the game map, a similar pixel based solution was used. We wrote another matlab script that reads the color of each pixel in a 28x31 pixel image that represents the game board. Each pixel color represented a different glyph type in the game. The script was parameterized such that it wrote 868 lines of memory initialization that represented the glyphs needed to draw the map. This memory initialization was then linked with the assembler to be included as preloaded data for the program. To draw the game board, the glyph numbers were copied from memory to the frame buffer with assembly code.

VIII. Results

We completed the entire hardware design and the majority of the application successfully. The hardware portions were completed and tested propitiously. The glyph based graphics worked extremely well and we were able to create over four hundred unique glyphs for our application. The script that initialized memory to draw the game board also worked smoothly. For the *Pac-man* game, we implemented a start menu. *Pac-man* had the ability to move around the playable board and eat pills based on the input from a player. All four ghosts were added to the game and moved independently.

We originally attempted to write an AI that would cause the ghosts to follow *Pac-man*. This proved to be too complicated to implement in assembly code, so we opted for a pseudo random movement scheme for ghosts instead. While the ghosts did not chase *Pac-man*, the movement patterns were extremely random and made the ghosts roam the entire map without any observable pattern.

The features that actually made *Pac-man* a game (like the ability to die) were ready to be added to the application. The assembly process, however, began to take so long that we ran out of time to integrate them with the completed features. The end game conditions were never programed, so the end game screen that was written and tested was never implemented. We had the ability to keep a score based on the number of pills *Pac-man* ate that was ready to be called when *Pac-man* ran over a pill or power-up. We had code prepared to play all of the game sounds. Method calls, hardware timers, and application states for the ghost flashing and the death of *Pac-man* were completed as well but failed to make it into the final demonstration.

IX. Conclusions

The tools were the biggest bottleneck in the project. Toward the end of the project it took twenty minutes to complete the synthesis and place and route scripts. We would have implemented a bootloader or eprom if we had known this from the beginning. Our software practices where the next problem. We should have implemented simpler models and built upon them instead of jumping into complicated assembly. An example would be creating one ghost which moved freely around the board and had the ability to kill *Pac-man*. Instead, we had four ghosts which did not have this ability. Given twenty-four more hours, we could have finished the project to 100% completion.

The general consensus of our group is that we enjoyed working on this project. We all learned a great deal about computer systems and enjoyed designing and creating the entire project from the ground up..

In the end, our project was 97% complete at demonstration.

APPENDIX A
CR-16 ISA

Table I: Supported Instructions

				ImmHi/	ImmLo/
		Op Code	Rdest	OpCodeExt	Rsrc
Mnemomic	Operands	15-12	11-8	7-4	3-0
ADD	Rsrc,Rdest	0000	Rdest	0101	Rsrc
ADDI	Imm,Rdest	0101	Rdest	ImmHi	ImmLo
ADDU	Rsrc, Rdest	0000	Rdest	0110	Rsrc
ADDUI	Imm, Rdest	0110	Rdest	ImmHi	ImmLo
ADDC	Rsrc, Rdest	0000	Rdest	0111	Rsrc
ADDCI	Imm, Rdest	0111	Rdest	ImmHi	ImmLo
MUL	Rsrc, Rdest	0000	Rdest	1110	Rsrc
MULI	Imm, Rdest	1110	Rdest	ImmHi	ImmLo
SUB	Rsrc, Rdest	0000	Rdest	1001	Rsrc
SUBI	Imm, Rdest	1001	Rdest	ImmHi	ImmLo
SUBC	Rsrc, Rdest	0000	Rdest	1010	Rsrc
SUBCI	Imm, Rdest	1010	Rdest	ImmHi	ImmLo
CMP	Rsrc, Rdest	0000	Rdest	1011	Rsrc
CMPI	Imm, Rdest	1011	Rdest	ImmHi	ImmLo
AND	Rsrc, Rdest	0000	Rdest	0001	Rsrc
ANDI	Imm, Rdest	0001	Rdest	ImmHi	ImmLo
OR	Rsrc, Rdest	0000	Rdest	0010	Rsrc
ORI	Imm, Rdest	0010	Rdest	ImmHi	ImmLo
XOR	Rsrc, Rdest	0000	Rdest	0011	Rsrc
XORI	Imm, Rdest	0011	Rdest	ImmHi	ImmLo
MOV	Rsrc, Rdest	0000	Rdest	1101	Rsrc
MOVI	Imm, Rdest	1101	Rdest	ImmHi	ImmLo
LSH	Ramount, Rdest	1000	Rdest	0100	Ramount
LSHI	Imm, Rdest	1000	Rdest	000s	ImmLo
ASHU	Ramount, Rdest	1000	Rdest	0110	Ramount
ASHUI	Imm, Rdest	1000	Rdest	001s	ImmLo
LUI	Imm, Rdest	1111	Rdest	ImmHi	ImmLo
LOAD	Rdest, Raddr	0100	Rdest	0000	Raddr
STOR	Rsrc, Raddr	0100	Rsrc	0100	Raddr
Scond	Rdest	0100	Rdest	1101	cond
Bcond	disp	1100	cond	DispHi	DispLo
Jcond*	Rtarget	0100	cond	1100	Rtarget
JAL*	Rlink, Rtarget	0100	Rlink	1000	Rtarget
RETX		0100	0000	1001	1111
WAIT		0000	0000	0000	0000

*Note: The Jcond and JAL instructions are not directly supported. Instead the programmer must jump with the following format:

Jcond[regToDestroy][label] JNE r3 label1
JAL[regToDestroy][lable] JAL r3 label3

The assembler decomposes the above pseudo instruction to create the jump address in the specified register, then creates the actual jump instruction supported by the processor to jump to the address created in the register.

REFERENCES

[1] Namco Bandai Games Inc., "Bandai namco press release for 25th anniversary edition," Press Release, December 2005, http://web.archive.org/web/20071230012914/http://www.bandainamcogames.co.jp/bnours/hotnews/index.php?id=21.

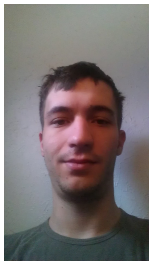
[2] C. Merris. (2005, May) Pac man turns 25: A pizza dinner yields a cultural phenomenon and millions of dollars in quarters. [Online]. Available: http://money.cnn.com/2005/05/10/commentary/game_over/column_gaming/index.htm

[3] (2010, May) Nintendo entertainment system (nes) - 1985-1995. [Online]. Available: http://classicgaming.gamespy.com/View.php?view=ConsoleMuseum.Detail&id=26&game=5

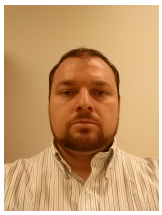
[4] (2008, Aug.) Cs/ee 3710 risc processor. [Online]. Available: http://www.eng.utah.edu/~cs3710/handouts/ISA.pdf

[5] Tarvizo. The nes controller handler. [Online]. Available: http://www.mit.edu/~tarvizo/nes-controller.html

[6] T. C. J. T. Ari Ferro, Zachary Wilcox. (2012, Dec.) Pacman assembly source code. [Online]. Available: http://eng.utah.edu/~ferro/AssemblyCode.zip

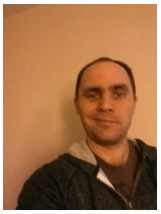


Ari Ferro is working on a double major in computer science and computer engineering. His interest is how software interacts with hardware.



Jeff Thacker is a Biomedical Engineer and working on degrees in Computer Engineering and Computer Science. His interests are in hardware and software development.

Todd Cronin is currently a junior at the University of Utah where he studies computer engineering and trains through Air Force ROTC to become an officer upon graduation. His interest primarily lay in hardware design.



Zachary Wilcox is a Computer Engineering undergraduate student at the University of Utah. His interests are in hardware design and architecture.