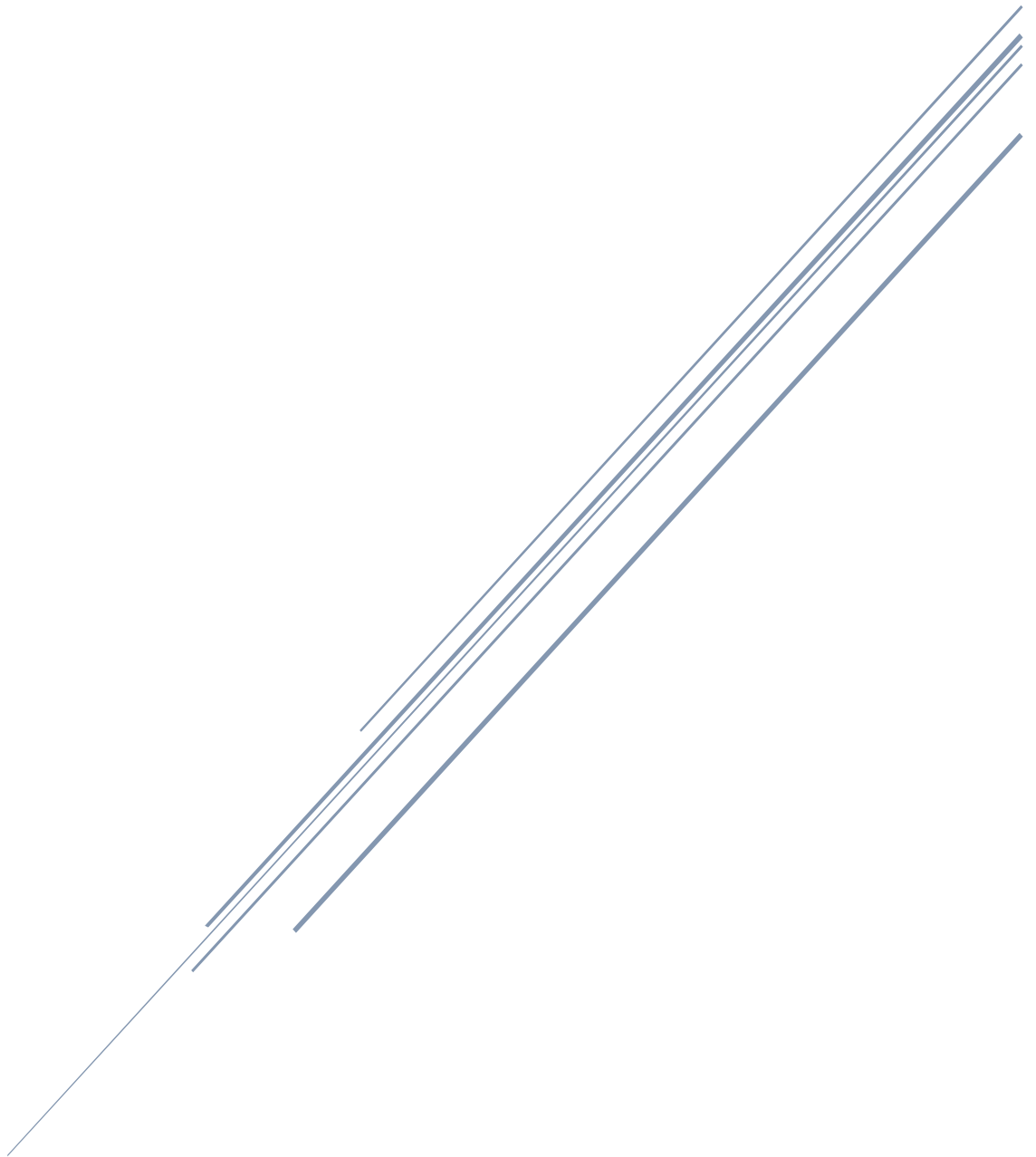


# 18-545 DESIGN DOCUMENT

VOICE CONTROLLED PACMAN



Rohit Banerjee  
Xiaofan Li

## Table of Contents

<b>Background Information</b> .....	<b>2</b>
Purpose .....	2
History .....	2
Platform .....	3
<b>System Implementation</b> .....	<b>4</b>
Z80 CPU .....	4
Memory Map .....	6
GPU .....	7
Tile Unit .....	8
Sprite Unit .....	10
Top Level Graphics Module .....	11
Sound .....	12
Mixer .....	13
PWM amplifier .....	14
External I/O .....	14
Voice Control .....	14
Alternative Inputs .....	15
Software .....	16
The mysterious rst #0x28 .....	16
The Main Loop .....	17
The Jump Table .....	19
<b>Tools</b> .....	<b>20</b>
<b>Schedule</b> .....	<b>20</b>
<b>Challenges Faced</b> .....	<b>21</b>
CPU integration .....	21
GPU design .....	21
Sprite and Tile encoding .....	21
Reverse Engineering the Assembly .....	23
<b>Personal Statements</b> .....	<b>23</b>

## **Background Information:**

### **Purpose**

The goal of this project was to produce an emulated version of the original 1980 Pacman game that takes voice commands as control inputs. We intend to create this game using ROMs obtained from video game enthusiast websites<sup>1</sup> as well as a Z80 CPU obtained from OpenCores.

### **History**

Pac-man is an arcade game that was developed by NAMCO and released in 1980. The game was novel as the most popular arcade games at the time were space-shooters such as Space Invaders and Asteroids<sup>2</sup>. Pac-man managed to create a new genre of arcade games and became one of the best-selling arcade games ever with inflation adjusted revenue being approximately 8 billion \$USD<sup>3</sup>. Its success and popularity also led to it become an important part of pop culture.

The game involves the player controlling Pac-man through a maze while making him eat pellets. When all of the pellets in the maze are eaten, the game proceeds to the next stage. The four ghosts (Blinky, Pinky, Inky and Clyde) roam the maze trying to catch the Pac-man. If Pac-man collides with one of the ghosts, he loses a life and both the ghost and Pac-man reset to their starting positions. In addition to the regular pellets, there are also 4 larger flashing pellets known as power pellets. Eating a power pellet causes the ghosts to turn deep blue and move slower. In this state, Pac-man can eat the ghosts and cause them to temporarily return to the “jail” that they were in at the start of the game. Higher levels cause the ghosts to move slightly faster and also reduces the time for which they remain vulnerable when the power pellets are eaten.

In addition to its economic success, Pac-man is regarded as one of the most influential games of all time as it showed the potential of characters in video games and also was the first game to feature power-ups and cut scenes<sup>4</sup>.

---

<sup>1</sup> [http://www.emuparadise.me/M.A.M.E.\\_Multiple\\_Arcade\\_Machine\\_Emulator\\_ROMs/Pac-Man\\_\(Midway\)/15250](http://www.emuparadise.me/M.A.M.E._Multiple_Arcade_Machine_Emulator_ROMs/Pac-Man_(Midway)/15250)

<sup>2</sup> <https://en.wikipedia.org/wiki/Pac-Man>

<sup>3</sup> <http://www.usgamer.net/articles/top-10-biggest-grossing-arcade-games-of-all-time>

<sup>4</sup> <https://en.wikipedia.org/wiki/Pac-Man>

## Platform

We decided to use the NEXYS 4 FPGA for this project as the logic requirements for the project were not particularly intensive and also because the NEXYS 4 has sufficient amounts of block RAM.

## System Implementation:

A diagram of the top level architecture used in our implementation is as follows:

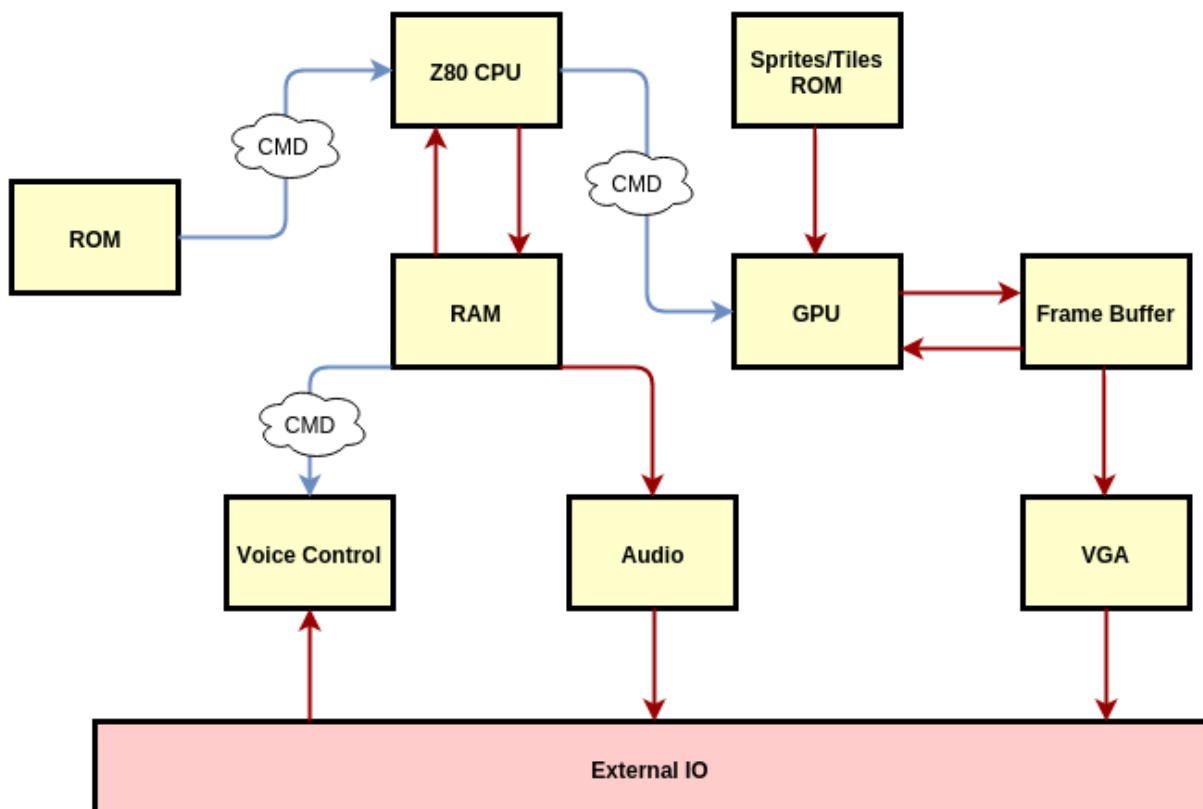


Fig 1: Top level system architecture

As seen from Fig 1, the major parts of the system are as follows:

- Z80 CPU
- GPU
- VGA
- Audio unit
- Voice control
- Memory

The following sections will cover each of the components that make up the system in greater detail.

## **Z80 CPU**

The 8-bit Z80 processor used in the original pacman game machine was available on OpenCores as T80 processor. However, the implementation from the website was not completely usable. Specifically, the timing for communicating with the memory systems, the interrupt system, IO communication and some instructions such as DJNZ are not implemented. In general, the CPU is not as easy to work with as advertised.

The timing requirement for the Z80 processor depends on the type of memory requests and the type of instruction executed. Since we are running on a FPGA, RAM refreshment has been turned off with System Verilog preprocessor macros. Memory fetches, RAM accesses and PC jump instructions all have different timing requirements. Furthermore, the module pacman\_mm in the top module arbitrates among accesses between the framebuffer, the RAM and ROM accesses. For access conflicts between the GPU and CPU (this case happens when GPU tries to read from and CPU tries to write to the framebuffer at the same time), GPU takes precedence over the CPU.

All the IO communication is simulated in the CPU's top module. Whenever a IO request is issued in the software, the IO values are recorded in the top module. Fortunately, there is no large storage device or DMA that needs to be simulated. Furthermore, since the Pacman game uses MMIO memory space (address 0x5000 and above), there needs to be hardware support in our implementation. In the top module, all the reads and writes to the MMIO space are captured and recorded in

registers. The data in the MMIO space behave according to the Pacman software requirement. For example, reads from 0x5000 are inputs from the first IO port (the first joystick control), while writes to 0x5000 controls interrupt enable/disable.

## Memory Map

### Frame buffer:

Name	Start Address	End Address
Tile indices	0x4000	0x43FF
Tile palettes	0x4400	0x47FF

### Other memory:

Name	Start Address	End Address
General purpose RAM(undocumented)	0x4800	0x4C01
Sprite Variables	0x4C02	0x4C7F
Tasks and Timers	0x4C80	0x4CCF
Game Variables	0x4D00	0x4DB5
Difficulty Settings	0x4DB6	0x4DBF
Counters	0x4DC0	0x4DD6
Main States	0x4EE0	0x4E65
Coins and Credits	0x4E66	0x4E8A
Sound Tables	0x4E8B	0x4EFF
Stack	0x4F00	0x4FBF
Unused	0x4FC0	0x4FEF
Sprite #,xflip and yflip	0x4FF0	0x4FFF
IN0(Joystick, coin, credit input)	0x5000	0x5000
Sound RAM(Data for 3 voices)	0x5040	0x505F
Sprite Coordinates	0x5060	0x506F
Flip screen	0x5003	0x5003

We decided to store the sprite data(0x4FF0-0x4FFF and 0x5060-0x506F) as well as Sound RAM(0x5040-0x505F) in registers as such storage makes it easier for the sprite and sound modules to interact with the data.

## GPU

A diagram of our GPU architecture is as follows:

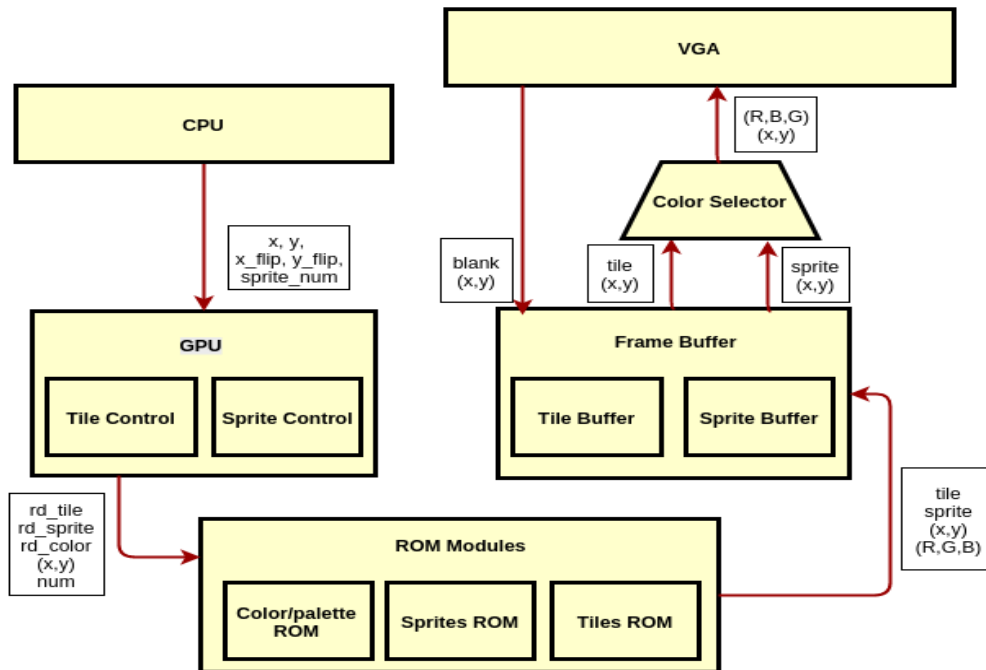


Fig 2: GPU architecture

The graphics in Pacman can be divided into 2 major types – tile graphics and sprite graphics. Tiles are 8x8 pixels large and there is a static mapping of screen locations to memory addresses. On the other hand, sprites are 16x16 pixels large and can be placed anywhere on the screen save the top 16 rows and bottom 16 rows.

The Pacman screen is 224x288 pixels large. When measured in terms of tiles, each row has 36 tiles and there are 28 such rows in total. In an effort to compress memory required to store graphics data, both sprites and tiles are coloured using palettes.

The Pacman color ROM has a 16 8-bit colors<sup>5</sup> each with 3 bits of red and green and 2 bits of blue. In addition the palette ROM has 32 palettes where each palette contains pointers to 4 colors in the color ROM<sup>6</sup>. This allows the pixels in the sprites in the tiles to simply require 2 bits. The mechanism for determining the color of a pixel is as follows:

<sup>5</sup> Pac-Man Emulation Guide v0.1, Oct 2008, Chris Lomont

<sup>6</sup> ibid



1. Use the 2-bit pixel value to index the palette in order to get an address in color ROM.
2. Use the color ROM address to obtain an 8 bit color value.

In our implementation we divided the GPU into a sprite unit, a tile unit and a top level module used to overlap sprite and tile colors. It is also important to note that our implementation used registers to store color ROM data to allow for faster access.

### Tile Unit:

As mentioned earlier, the game has a static mapping of screen positions to memory locations. This mapping is illustrated in the following image:



Fig 3: Mapping of memory locations to screen positions<sup>7</sup>

As seen in Fig 3, each of the “boxes” corresponds to memory locations that holds information about the tile the tile to be drawn at that location. RAM addresses 0x4000 – 0x43FF hold the ROM address of the tile to be drawn (this is between 0 and 255 as there are 256 tiles). Memory addresses 0x4400-0x47FF holds the

<sup>7</sup> Pac-Man Emulation Guide v0.1, Oct 2008, Chris Lomont

corresponding palette used to draw a tile at that location. As the graphics unit was a major component in the Pacman system, we separated the graphics RAM 0x4000-0x47FF into a separate block of RAM called the frame buffer to allow for simultaneous development of the main system and the graphics system and also to reduce conflicts between the GPU and CPU on memory reads.

The steps taken by the hardware to draw tiles on the screen is as follows:

1. Get a (row, col) value from the VGA unit
2. Translate the (row, col) value into an 8-bit tile RAM address (0x4000-0x43FF) and a 6-bit palette RAM address (0x4400-0x47FF).
3. Access the frame buffer to get a tile ROM address and a palette ROM address.
4. At the same time, use the lower 3 bits of the row and col to determine the 6-bit index of the pixel in the tile (between 0 and 63 inclusive).
5. Concatenate the 8-bit tile ROM address with the top 4 bits of the pixel index to obtain a 12 bit pixel address.
6. Use the pixel address to access tile ROM to obtain a byte containing 4 pixels. Also use the lower 2 bits of the pixel index to access data for the pixel we want.
7. At the same time use the palette ROM address to access data for a single 32 bit palette from palette ROM.
8. Use the pixel data to index a byte from the 32-bit palette data to get the index to a color.
9. Use the color index to access an 8-bit RGB value stored in registers.
10. Repeat for all the (row, col) values obtained from the VGA unit.

During our initial verification phase, we used a test bench that displayed all 256 tiles on screen and allowed the user to choose palettes via switch input (on the FPGA) to debug and validate this part of the GPU.

Sprite Unit:

The Pacman game sprite ROM contains a total of 64 16x16 sprites, however there can only be 8 sprites on screen at any point in time. In addition, sprites cannot be drawn in the top 16 or bottom 16 rows of the game screen. It is also important to note that unlike tiles, sprites also have the ability to be flipped in the x or y directions. This allows for fewer sprites to be stored in ROM. Data pertaining to how these 8 sprites should be drawn is stored in RAM at the addresses shown in Fig 4.

Sprite	X-location	Y-location	Sprite#, flip-x, flip-y	Palette
0	5060h	5061h	4FF0h	4FF1h
1	5062h	5063h	4FF2h	4FF3h
2	5064h	5065h	4FF4h	4FF5h
3	5066h	5067h	4FF6h	4FF7h
4	5068h	5069h	4FF8h	4FF9h
5	506Ah	506Bh	4FFAh	4FFBh
6	506Ch	506Dh	4FFCh	4FFDh
7	506Eh	506Fh	4FFEh	4FFFh

Table 5 – Pac-Man sprite addresses

Byte 0								Byte 1							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Sprite # 0-63						X flip	Y flip	Palette entry 0-15							

Table 6 – Sprite register definition

Fig 4: RAM locations for sprites<sup>8</sup>

As seen in Fig 4, each of the sprites has 4 bytes of data associated with it:

1. X position
2. Y position
3. Concatenation of the sprite index (0-63) and the xflip and yflip(both asserted high).
4. The palette used to color the sprite.

It must be also be noted that the x and y positions that the CPU writes to the above memory locations assumed that the screen coordinates start at the bottom right of the screen and go right to left, bottom to top. Thus it was necessary for us to pass the x and y coordinate through a transformation module that mapped the x and y positions to the top left, left to right, top to bottom system used by the VGA module before we could do comparisons.

<sup>8</sup> Pac-Man Emulation Guide v0.1, Oct 2008, Chris Lomont

Unlike the tile module, the sprite module does not access RAM, instead, it redirects writes made by the CPU to the addresses shown in Fig 4 to registers. The steps taken by the sprite module to draw sprites is as follows:

1. Get a (row, col) value from the VGA unit.
2. Get the sprite with the lowest number that occurs in the region. If no such sprite exists output RGB values of 0(black).
3. Normalize the (row, col) coordinates between (0-15) i.e. normalize to the dimensions of a sprite.
4. Use the lowest 4 bits of the normalized row and col values as well as the xflip and y flip values of the sprite to determine the 8-bit index of the pixel in the sprite.
5. Concatenate the 6-bit sprite number with the top 6 bits of the pixel index to create a pixel address.
6. Use the pixel address to access sprite ROM to obtain a byte containing 4 pixels. Also use the lower 2 bits of the pixel index to access data for the pixel we want.
7. At the same time use the palette address for the sprite to access data for a single 32 bit palette from palette ROM.
8. Use the pixel data to index a byte from the 32-bit palette data to get the index to a color.
9. Use the color index to access an 8-bit RGB value stored in registers.

During our initial verification phase, we used a test bench that displayed all 63 tiles on screen and allowed the user to choose palettes as well as xflip and yflip values via switch inputs (on the FPGA) to debug and validate this part of the GPU.

#### Top level Graphics module:

The Top level Graphics module contains both the Sprite module and the Tile module and is used to select which color is sent to the VGA interface. The top level graphics module (TLGM) receives the color outputs from both the sprite and the tile modules. If the TLGM detects that the sprite color output is black (which is used as a transparent color in Pacman) it sends the tile color output to the VGA, else it sends the pixel color output.

## Sound

The original Pacman sound used the NAMCO WSG – a 3 channel waveform generator. In this setup voices 1 and 2 play the background song while voice 3 is used for SFX. Each of the voices has a frequency and a volume associated with it. In addition, the sound ROM contains 16 waveforms (of which only the first 8 are used). Like the on-screen sprites, the frequency, volume and waveform indices for each of the voices are mapped to RAM addresses as shown in the figure below.

Register	Address	Notes
Voice 1 Waveform	5045h	low 3 bits used – selects waveform 0-7 from ROM
Voice 1 Frequency	5050h-5054h	20 bits in low nibbles
Voice 1 Volume	5055h	low nibble – 0 off to 15 loudest
Voice 1 Accumulator	5040h-5044h	low nibbles, used by H/W only
Voice 2 Waveform	504Ah	low 3 bits used – selects waveform 0-7 from ROM
Voice 2 Frequency	5056h-5059h	16 bits in low nibbles
Voice 2 Volume	505Ah	low nibble – 0 off to 15 loudest
Voice 2 Accumulator	5046h-5049h	low nibbles, used by H/W only
Voice 3 Waveform	504Fh	low 3 bits used – selects waveform 0-7 from ROM
Voice 3 Frequency	505Bh-505Eh	16 bits in low nibbles
Voice 3 Volume	505Fh	low nibble – 0 off to 15 loudest
Voice 3 Accumulator	504Bh-504Eh	low nibbles, used by H/W only

Fig 5: Address mapping for sound<sup>9</sup>

The sound system is clocked at 96KHz. Every clock cycle, each of the voices perform the following actions to choose the sample to send to the amplifier:

1. Add the voice frequency to the voice amplifier
2. Concatenate the waveform index with the top 5 bits of the accumulator to create a sample address.
3. Use the sample address to get a byte from the sound ROM.
4. Send the lower nibble of the byte to the amplifier.

<sup>9</sup> Pac-Man Emulation Guide v0.1, Oct 2008, Chris Lomont

In our implementation, we had our sound module redirect CPU writes to the RAM locations shown in Fig 5 to registers, similar to the sprite module. A block diagram of our sound system is shown below:

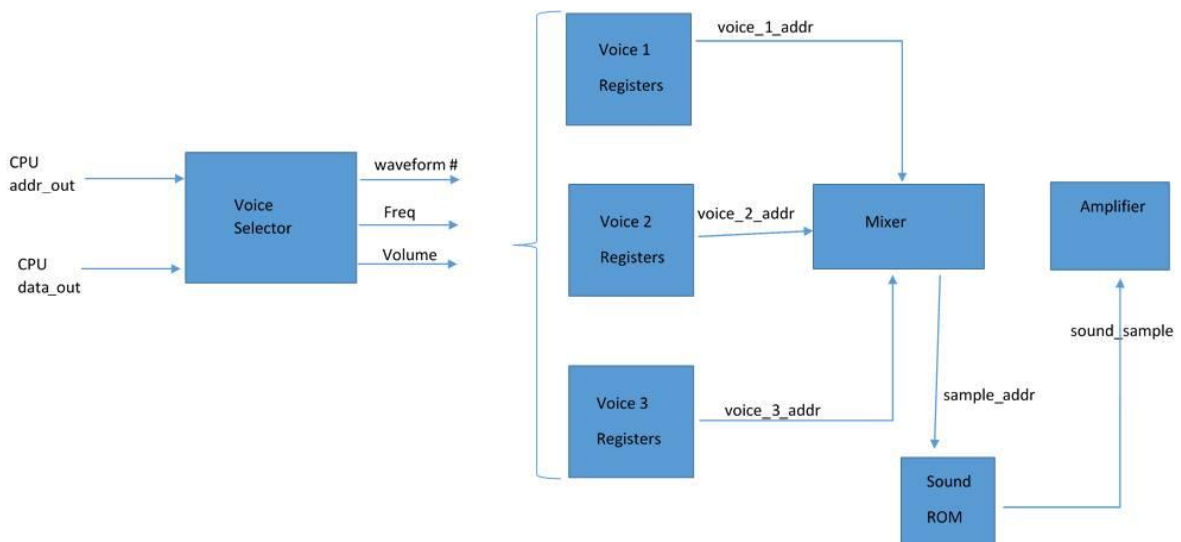


Fig 6: Block diagram of sound system

### Mixer:

Since, Pacman only uses mono sound our mixer simply added the samples produced by each of the voice registers and then shifted the result left by 2 bits to produce the resultant output. This output was then sent to our PWM amplifier.

### PWM amplifier:

The NEXYS 4 outputs sound via PWM. Thus we had to write a sample to PWM conversion module in order to send sound input to the speaker. Thus our PWM module accomplished the following:

1. Have a PWM counter continuously run much faster than the sampling rate of the sound system (96 KHz).
2. Convert the sound samples sent by the mixer into PWM duty threshold values (simple combinational unit).
3. If the PWM counter is less than the threshold value obtained in step 2 output a 1 to the speaker, else output a 0.

Our testing for the PWM audio was done on the Pacman waveforms themselves.

### **External I/O**

#### Voice Control

As we proposed in the pitch presentation, we implemented a voice control feature. Theoretically, the voice control feature is possible to implement without consuming much resources because of the nature of the problem: after all, we are only trying to control the game with only 4 words -- up, down, left and right. It is definitely possible to store moderate amount of data to distinguish about 4 seconds' of audio.

We designed a datapath to control recording, storing and comparing of audio data read from the audio hardware on the board. We also verified that this process works with simulation. However, when we tested it in real time, the comparison is not accurate enough to output clear answers. We suspect that it was because we have time-space shifts in recording the sample for the comparisons. Without a FFT module to transform the audio signal into frequency space, we cannot account for time-space shifts in audio inputs. As we realized this fact in the last week of testing and did not have enough time to research the FFT module, we do not have a working voice control feature.

### Alternative Inputs

As a “fail-safe” plan, we also implemented input controls from the switches. The first two switches can generate a 2-bit input signal that translates to the 4 possible movement directions. The input is generated and written to RAM space at 0x5000 and 0x5040 which correspond to the P1 and P2 input addresses.



## Software

The Software used was downloaded from online as a series of ROM files. Sparsely commented disassembled code was found online on MsPacman game which shares similarities with the ROM file we are using (we did find Pacman assembly code but it was not well commented at all). I suspect that documenting such reverse engineered source code was strictly forbidden by the copyright owner. Thus, much of the code was still mysterious to the people who published the assembly code. For example, from the comments on the code, no one seemed to understand the working of the `rst #0x28` function but it was called many time throughout the source code. Through reverse engineering and intense assembly reading, we were able to understand more about the Pacman and MsPacman assembly code than any published websites online (that we can find). In this section, we will explain two important findings of the assembly code.

### 1. The mysterious `rst #0x28`

This figure is the complete assembly code on the function at 0x28 from the `mspacman.asm` webpage:

```
        ;; rst 28
        ; this is completely baffling me
0028  e1      pop     hl          ; next byte after call
0029  46      ld      b,(hl)
002a  23      inc     hl
002b  4e      ld      c,(hl)
002c  23      inc     hl          ; bc gets the word after the call
002d  e5      push    hl          ; adjust return value
002e  1812    jr      #0042        ; (18)

        ;; continuation of rst 28
0042  2a804c   ld      hl,(#4c80)          ; bc has the word after the call (above)
                                ; hl = (4c80)
0045  70      ld      (hl),b
0046  2c      inc     l
0047  71      ld      (hl),c
0048  2c      inc     l          ; ((4c80), (4c81)) = bc
0049  2002    jr      nz,#004d
004b  2ec0    ld      l,#c0          ; if( c==00 ) l = c0 (spins c0-ff)
004d  22804c   ld      (#4c80),hl          ; (4c80, 4c81) = hl
0050  c9      ret
```

The key to understanding this routine is at 0x2c, “bc gets the word after the call”. This literally means that the register BC gets the word after the “`rst #0x28`” instruction **in the program ROM file**. Below is one of the call sites of `rst #0x28`:

```
05f6 ef rst #28
05f7 0001
05f9 ef rst #28
05f8 0100
05fc ef rst #28
05fd 1c07
05ff ef rst #28
0600 1c0b
0602 ef rst #28
0603 1e0
0605 21034e ld hl,#4e03
0608 34 inc (hl)
0609 3e01 ld a,#01
060b 32d64d ld (#4dd6),a
060e 3a714e ld a,(#4e71)
0611 feff cp #ff
0613 c8 ret z

0614 ef rst #28
0615 1c0a
0617 ef rst #28
0618 1f00
061a c9 ret
```

As we can see, the bytes “0001” after the rst instruction will be put into the register BC and the return value will be modified to point to the next rst instruction.

## 2. The main loop:

In the code segment beginning at 0x42, the register BC is inserted into the memory location pointed to by the data pointed to by 0x4c80. From this memory access pattern, we are certain that this is some kind of a queue and the rst #28 instruction is inserting data into this queue. Thus if we can find the location where we pop off things from this queue, then we would be sure. Furthermore, I was suspecting it is not only a queue, but it's some kind of a task list because the data we are inserting are actually static ROM data. So if we can find some places where this data is used to affect program control flow, then this guess can surely be verified. A long time later, we found the following code segment:

```
;; 0xff -> 4cc0-4cff
2382 3eff    ld     a,#ff
2384 0640    ld     b,#40
2386 cf      rst     #8
2387 3e01    ld     a,#01
2389 320050  ld     (#5000),a      ; enable interrupts
238c fb      ei      ; enable interrupts
238d 2a824c  ld     hl,(#4c82)
2390 7e      ld     a,(hl)
2391 a7      and     a
2392 fa8d23  jp     m,#238d
2395 36ff    ld     (hl),#ff
2397 2c      inc     l
2398 46      ld     b,(hl)
2399 36ff    ld     (hl),#ff
239b 2c      inc     l
239c 2002    jr     nz,#23a0      ; (2)
239e 2ec0    ld     l,#c0
23a0 22824c  ld     (#4c82),hl
23a3 218d23  ld     hl,#238d
23a6 e5      push   hl
23a7 e7      rst     #20          ; where does it jmp from here?
```

This code segment is actually the main loop of the game. As we can see at 0x23a3, when we push 0x238d onto the stack, this value will later be used as the return value for whatever task routine it executes later. This will be explained in the next section. Now, we can see that this routine is in fact getting data from the data pointed to by 0x4c82 and writing -1 to it before it calls rst #0x20. Because of the proximity of 0x4c80 and 0x4c82 and the similarity of the data access between the two locations, I was certain that this is the place where this queue is popped off. However, do I necessarily know the data it gets from the queue will affect the control flow?

### 3. The jump table

The segment below was a very confusing piece of code at first because it does not make any sense whatsoever. However, the second byte of every two-byte word has similar values, which means they are probably all jump targets that point to the beginning of some function calls of code segments:

```

23a8 ed23 db #ed, #23
23aa d7 rst #10
23ab 24 inc h
23ac 19 add hl,de
23ad 24 inc h
23ae 48 ld c,b
23af 24 inc h
23b0 3d dec a
23b1 25 dec h
23b2 8b adc a,e
23b3 260d ld h,#0d
23b5 24 inc h
23b6 98 sbc a,b
23b7 2630 ld h,#30
23b9 27 daa
23ba 6c ld l,h
23bb 27 daa
23bc a9 xor c
23bd 27 daa
23be f1 pop af
23bf 27 daa
23c0 3b dec sp
23c1 2865 jr z,#2428
23c3 288f jr z,#2354
23c5 28b9 jr z,#2380
23c7 280d jr z,#23d6
23c9 00 nop
23ca a2 and d
23cb 26c9 ld h,#c9
23cd 24 inc h
23ce 35 dec (hl)
23cf 2ad026 ld hl,(#26d0)
23d2 87 add a,a
23d3 24 inc h
23d4 e8 ret pe

;; subroutine
23d5 23 inc hl
23d6 e3 ex (sp),hl
23d7 28e0 jr z,#23b9
23d9 2a5a2a ld hl,(#2a5a)
23dc 6a ld l,d
23dd 2b dec hl
23de ea2be3 jp pe,#e32b
23e1 95 sub l
23e2 a1 and c
23e3 2b dec hl
23e4 75 ld (hl),l

```

After verifying this guess in the assembly code by finding those targets and reading the code treating it as the beginning of a function call, I was certain that `rst #0x20` is

used to index a jump table and change control flow. This can further be verified by reading the code located at 0x20. Interestingly, many of the functions listed in the above table has comments such as “I can’t find a jump to here”. It seems like whoever wrote the comment was really confused and pondering if whatever he is reading is actually useless dead code.

## **Tools:**

In this project we used Vivado for synthesis and simulation during integration. We also used vcs to simulate module when we were debugging them in isolation. Python was our language of choice for scripting and we used Github for version control and collaboration.

## **Schedule:**

- Week 1-2: Choose topic (we ran a bit late during the start due to team assignment issues).
- Week 3: Acquaint ourselves with CPU and assign responsibilities, also perform further research on Pacman hardware.
- Week 4: Verify CPU
- Week 5: Implement VGA, write scripts to create block RAMs/ROMs
- Week 6-7: Implement tile unit and validate
- Week 8: Establish top level module for system, start work on sprite unit
- Week 9: Write scripts to test sprite decoding scheme
- Week 10-11: Finish sprite unit, begin integration
- Week 12: Continue integration, check assembly source code, finish GPU top module. Complete sound module.
- Week 13-14: Test and debug software, work on voice recording unit.

We had a few difficulties keeping the schedule initially due to the lack of documentation on key components of our system such as the GPU and incorrect information regarding graphics encoding from our literature sources. We were able to make up this time later in the semester during our integration testing period.

However, the software issues we encountered late in the semester while testing the game software were hard to recover from and resulted in us not being able to get a fully functional Pacman game.

## Challenges Faced:

### CPU integration:

Learning to interface with a full sized CPU has proven to be challenging. Because of good design principles, the processor interfaces require other hardware implementation and customization. Thus we had to read the hardware specs in detail and configure the hardware. Also it took us a long time to get the timing correct to IO with the memory systems.

### GPU design:

One of the problems that we encountered early during our design phase was the fact that there was no documentation NAMCO NVC293 video shifter<sup>10</sup> the original Pacman GPU. Hence, we chose to use a VGA display because of its simplicity and subsequently engineered all of the other components to be compatible with the VGA interface.

### Sprite and Tile encoding:

Another issue we encountered while validating the tile and sprite units was that the images that we obtained on the VGA display were garbled. Further inspection revealed that the decoding scheme that we were originally using<sup>11</sup> for sprites and tiles ROM data was incorrect. As such we used Python scripts to validate the decoding scheme in software before implementing those schemes in hardware.

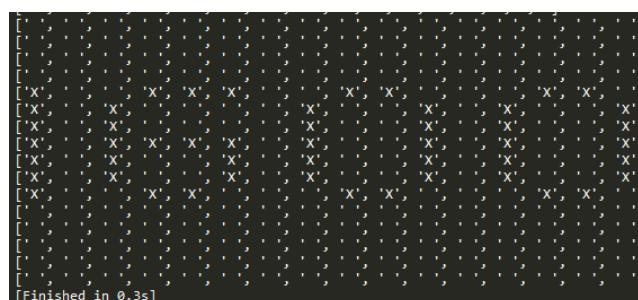


Fig 6: Sample output of python script for sprites

<sup>10</sup> [https://en.wikipedia.org/wiki/Namco\\_Pac-Man](https://en.wikipedia.org/wiki/Namco_Pac-Man)

<sup>11</sup> Pac-Man Emulation Guide v0.1, Oct 2008, Chris Lomont

### Reverse Engineering the Assembly

In order to bypass the start-up hardware verification and certain hardware requirement such as coin slots and start button. We had to patch the software to skip certain part of the software. This turned out to become a huge challenge since reverse engineering poorly documented assembly code was obviously very time consuming. In this end, the tests were skipped but the collision detection bug was most likely caused by some unknown effects of the software patch.

## **Personal Statements:**

### **Rohit**

I worked primarily on the GPU and sound hardware for the game. Working with the GPU allowed me to understand a lot about how the typical image compression techniques using palettes and flips work. I also found working with the sound system particularly interesting as this was the first time I had attempted to generate sound from an FPGA. Trying to convert sound samples into a single bit PWM output was particularly frustrating in the beginning however, hearing the classic Pacman sound being played by our hardware was immensely satisfying.

In addition, I also wrote the python scripts that allowed us to convert the ROM files (code, sprite, tile, palette and sound) into .coe files and thus load them into BRAMs on the board. I also wrote the scripts that allowed us to validate the decoding scheme used for sprite and tile data when we found that our literature sources were incorrect.

In the later stages of the project, I worked on integrating the sound and GPU with the main CPU. I also worked on the voice control module that used the on board microphone. I was able to get the voice control module to record but, unfortunately I was not able to get it to recognize commands. I was however, able to learn a lot about PDM (Pulse density modulation) sampling in the process.

My advice to other teams would be to not underestimate the time it takes to complete integration. It took us significantly longer (about 2-2.5x as long) to integrate individual components than it did to design and verify each of the individual components in isolation. Also, take the time to read over the synthesis warnings that Vivado gives as it tends to be rather temperamental (on some synthesis runs it tied random pins to power or ground for no reason). Lastly, abuse the ILA (integrated logic analyser) cores and their triggering capabilities as they are an invaluable resource while debugging.



## Li

I worked on designing of all major components, the integration of the Z80 processor with the rest of the hardware devices, understanding and patching of the software, and the implementation of external IO controls.

Hardware-wise, my design was mostly successful and worked well after integration. The only exception is the voice control module, where we ran out of time and prioritized debugging the collision detection over the voice control module. In software, while reversing and understanding the assembly code was challenging and rewarding, the most troublesome bug -- the collision detection bug -- was most likely caused by the lack of total understanding of the software.

I had the most fun when I was integrating the CPU and reversing the software. During the CPU integration, I had to go through the Z80 specification and experimented with the timing requirements. This part was easy to debug because it was possible to be simulated and tested on a cycle-by-cycle basis. When I was reversing the software, I understood some embedded system coding techniques. Tricks such as the above mentioned `rst #0x28` are very intriguing. I wonder whether the reason for deploying such coding styles was because the programmer needed to save ROM space for optimize for performance.

At the end of the project, the collision detection bug gave me much headache. At first I was devoted into finding a software solution. After many hours of reversing the source code and reading the software collision detection code, I was not sure what parameter values the collision routine was called with because simulation could not run long enough to trigger the collision code. With standard fuzzing techniques (changing the input values to the routine), I was about to conclude that the routine was called with values stored in the framebuffer to detect collisions. However, I believed that fixing the issue with software inputs was not possible without simulation because of the complexity and uncertainty about the software structure. After that, I devised a hardware module to attempt to fix the issue in the GPU. Basically, the GPU would read the tile information and perform the collision detection in hardware. Unfortunately, we ran out of time trying to write this module.

In general, I believe we did a good project as our setup and requirements are different from the other teams. We set out to do Pacman with voice control so we understood that we had to know and control all the hardware modules and the software assembly.

Taking a hardware module from online sources and hoping it works without knowing enough to modify the module will not work in the long term. Moreover, the GPU had no documentation online, which means we had to re-design and rewrite this module completely to merge it into the other parts of the projects. Therefore, I think we learned a lot through researching online and tinkering with different modules on our own.