GAME BOY® Nintendo

GameBoy Project

About GameBoy

The GameBoy is an 8-bit handheld video game device developed and manufactured by Nintendo. It was released in 1989. The backwards-compatible GameBoy Color was released in 1998.

Team Member

Wan Lee,
  Senior ECE major
  Computer Architecture

Pierce Lopez
  Senior ECE major
  Computer Architecture, Operating Systems

Sean Moorman
  Senior ECE major
  Computer Architecture, Computer Graphics

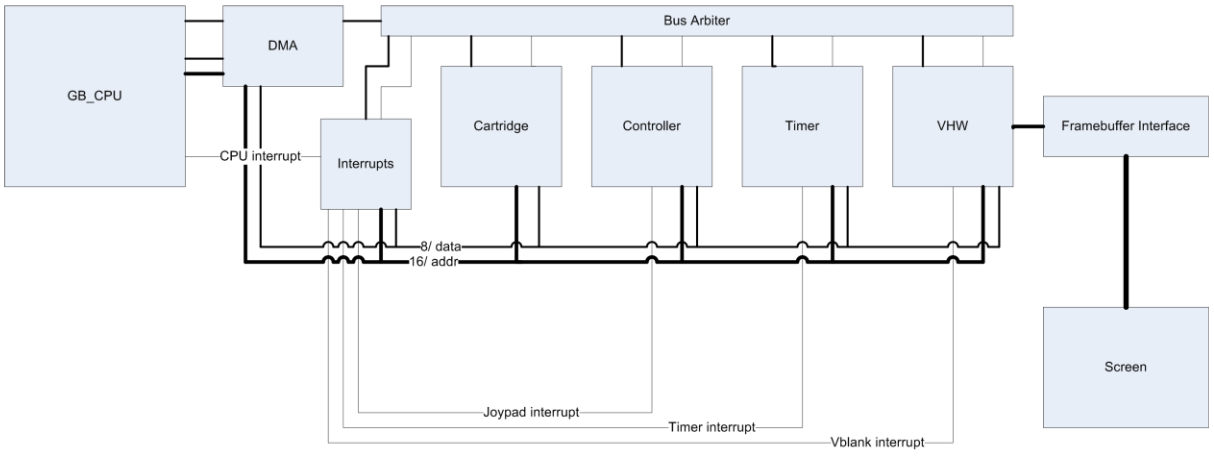
Original Goal

Implement a full GameBoy Color including link cable for multi-player capability.

Accomplished Goal

Implement an Original GameBoy sufficient to support Tetris without sound or link cable. And with some graphical corruption (due to incomplete DMA implementation).

Top Level Design Overview



CPU
Game Boy CPU is a stripped version of the Zilog's Z80 processor (or an enhanced Intel 8080, depending on your point-of-view) running at 4.19 MHz, with all instructions taking 4, 8, 12, 16, or 24 cycles (it was not an out-of-order or pipelined design). It has 7 8-bit registers, 6 of which can be referred to in 16-bit register pairs by some instructions. The registers are not general-purpose, and many instructions only support some of the registers.

Video Hardware
The Game Boy has a 160 x 144 pixel display. Each pixel supports 4 shades of gray. The screen is composed of background tiles and sprites. The background and sprites are made up of 8x8 tiles. The 160x144 pixel background is selected from a 32x32 tile background map. The background map can be scrolled and a window display can be displayed over it. There are a maximum of 40 sprites per screen and a maximum of 10 sprites per line due to a limitation in the video hardware.

Cartridge
A Game Boy cartridge is given 32 KiB of address space for the ROM, but most games banked the second 16 KiB of it in order to support 64 KiB - 512 KiB of rom. A cartridge also could use 8 KiB of address space for additional SRAM, which was usually backed by a watch battery and used to save games.

SRAM
The Game Boy had 8 KiB of directly-addressable SRAM. It also had 128 bytes of "High Memory" which was typically used for the stack, and was even jumped to and executed from during an OAM DMA.

Controller

The Game Boy had 8 buttons - 4 for the directional pad, and A, B, Start, and Select. The controller generated an interrupt on a button press, but the state of the buttons needs to be polled.

Timer
The Game Boy had a timer module which could be programmed to periodically trigger an interrupt.

Link Cable
The Game Boy could be directly connected to another gameboy for multiplayer, with a link cable. A simple full-duplex serial protocol was used over the link cable, and one gameboy would set the clock for the link.

Sound
The Game Boy had sound hardware which was controlled by programming 4 different sound channels which produced different sounds or sound effects.

CPU

Specification:

The Game Boy CPU is very similar to a Zilog Z80, except the Game Boy strips a lot of the Z80's functionality, adds a few instructions, changes the opcodes of a couple of instructions, and changes the semantics of HALT.

The Game Boy CPU has 7 8-bit regsiters and a flags register. Some of the registers can be used together as 16-bit register pairs.

Registers:
A - accumulator
F - flags

B	C
D	E
H(igh)	L(ow)

All 8-bit arithmetic instructions involved A and some other register, and saved the results to A. Most 8-bit loads to and from memory transferred to or from A. Many instructions involving a memory access used H and L together as a memory address (the High byte and the Low byte of the address). Some memory instructions, however, used BC or DE pairs as an address, and there were also a few other instructions involving these pairs.

Some general instruction types supported included add, add with carry, subtract, subtract with carry, increment, decrement, 8-bit move or load, 16 bit load, compare, call, return, jump, jump if condition (zero flag, carry flag), call if condition, return if condition, jump relative, jump relative if condition, test a single bit, set a single bit, reset a single bit, shift left arithmetic, shift right arithmetic, shift right logical, rotate left, rotate right, rotate left through carry, rotate right through carry, indirect load based on address in HL and increment or decrement HL, halt, stop, enable interrupts, disable interrupts, complement the carry flag...

Process and Implementation:

We found a z80 core written in verilog on opencores.org, and tried to adapt it. This z80 design had a two-stage pipeline such that the simplest instructions back to back would finish at a rate of one per clock. Rather than have some sort of microcode-like scheme, each part of the z80 would detect what instruction was in the inst register, and then adjust its behavior. Effectively, the handling of each instruction was spread out through the entire design. The z80 design defined the opcodes as parameters to make the instruction-matching statements more elegant and understandable, but it also took advantage of the specific bit patterns of these instructions in other areas. For example, the difference between an increment and a decrement of an 8-bit register was a single bit, so the control signals to the ALU had one setting which was used for both increments and decrements, and the ALU checked that single bit of the inst register itself. This sort of design style made it so that one couldn't even search for all the places in the code a particular instruction was handled, because some places just checked bits directly instead of using the opcode parameters. Overall, this z80 design proved to be efficient, but very difficult to modify as we needed. Also, we didn't need the efficiency, because the Game Boy cpu used 4 cycles for some ops that took this z80 design a single cycle (averaged, due to the pipeline).

We decided to start a new CPU design from scratch. We reused the opcode parameters from the z80 design because they were 90% correct for the Game Boy CPU. However we made the rest of our design completely different. This new design has a single "decode" module, the contents of which look like microcode, so that any odd instruction could be implemented by adding a section to the decode module, and sometimes adding support for a new op or a new input to the ALU or the Bus sections of the cpu core. We tried to make the decode module the only module which inspects the bits of the instruction, but in a couple of cases (CB prefix single-bit instructions, and RST) we let the alu access three bits from the instruction opcode, which were used as an index. A single section of the cpu core updated registers on the rising clock edge, and all the other parts of the cpu core were combinational. The decode unit mentioned earlier was combinational, and used for its input only the current sub-cycle, the instruction byte, and sometimes the next byte as well (CB prefix instructions). The decode unit had a variety of outputs to control the Bus unit and the ALU. The Bus unit would pick what registers should drive the address output, what register should drive the data output, and whether to raise the write-enable output. The ALU would use the decode module's control lines to select the registers or register pairs that feed it, what operation to apply to them, and what flags to affect. The register-update section would look at the decode module's control lines, the value on the memory read input, and the output of the ALU, and decide what value to save to each register at the end of each clock cycle.

The z80 design we started with (and scrapped) used combinational-read memory (asynchronous read), so that's how we designed the peripherals and the new cpu core to interact over the bus. We found that the Xilinx synthesis process would not use BRAMs for our large SRAM and ROM sections unless we made them synchronous-read, but we worked around that by putting these memories on a faster clock, so they still looked like asynchronous-read memories to the cpu.

Interrupt handling is spread throughout the cpu core. On a high level, interrupt handling works together with the interrupt device, which stores both the interrupt enable mask memory-mapped register, as well as the interrupt flags memory-mapped register, and handles the specific interrupt lines. When a bit of the interrupt flags is set, and that bit is also set in the interrupt enable mask, the interrupt device raises the interrupt line to the cpu core. If the cpu core has interrupt enabled, then at the start of the next instruction, it instead switches into interrupt-handling state. In the first cycle it loads the interrupt-enable register, and in the second it loads the flags register. In the third it both stores the flags register back with the bit of the interrupt it decided to run unset (it consults both the interrupt-enable register, and the priorities of the interrupt flag bits, which are defined by their order) as well as generates an RST instruction (a call to a jump table slot) and switches back to normal instruction execution mode to run the generated instruction. It applies little tweaks to make this work correctly. The Bus control part of the core changes its behavior when it detects that it is in interrupt mode in order to load and store to the correct addresses. A special single-bit register is set so the ALU knows that the next RST -related jump address it generates should be shifted into the interrupt jump table range. This whole mechanism could probably have been made more elegant. We are also unsure if it would have even been possible to tweak the z80 core we abandoned to have this behavior.

Video Hardware

Specification:

The GPU is assigned 8 KB of space from 0x8000 to 0x9FFF. Within this region are a series of registers used to control and display the state of the video hardware, the background maps, and the tile data. In our implementation we also considered the sprite attribute table 0xFE00 0xFE9F part of the video hardware.

Registers:

FF40 - LCDC - LCD Control (R/W)
-The LCDC register is used for a number of functions.
Bit 7 - LCD Display Enable (0=Off, 1=On)
Bit 6 - Window Tile Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
Bit 5 - Window Display Enable (0=Off, 1=On)
Bit 4 - BG & Window Tile Data Select (0=8800-97FF, 1=8000-8FFF)
Bit 3 - BG Tile Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
Bit 2 - OBJ (Sprite) Size (0=8x8, 1=8x16)
Bit 1 - OBJ (Sprite) Display Enable (0=Off, 1=On)
Bit 0 - BG Display (for CGB see below) (0=Off, 1=On)

FF41 - STAT - LCDC Status (R/W)
-The STAT register is used to display the current state of the video hardware and trigger interrupts when appropriate.
-The Gameboy has 4 modes.
-Mode 0: Video hardware is in h-blank. CPU can access RAM and OAM
-Mode 1: Video hardware is in v-blank. CPU can access RAM and OAM
-Mode 2: Video hardware is accessing OAM. CPU cannot read or write OAM at this time
-Mode 3: Video hardware is acessing OAM and VRAM. CPU cannot read or write either

FF42 - SCY - Scroll Y (R/W)
-The SCY register controls what line the background map starts displaying from.

FF43 - SCX - Scroll X (R/W)
-The SCX register controls what column the background map starts displaying from.

FF44 - LY - LCDC Y-Coordinate (R)
-The LY register states what line the video hardware is currently rendering.

FF45 - LYC - LY Compare (R/W)
-The LYC register is used to trigger an interrupt when LY=LYC

FF4A - WY - Window Y Position (R/W)
-WY is the line that the window overlay starts on.

FF4B - WX - Window X Position minus 7 (R/W)
-WX is the column that the window overlay starts on

FF47 - BGP - BG Palette Data (R/W) - Non CGB Mode Only
-BGP assigns the four possible indexes to colors.
- Bit 7-6 - Shade for Color Number 3
- Bit 5-4 - Shade for Color Number 2
- Bit 3-2 - Shade for Color Number 1
- Bit 1-0 - Shade for Color Number 0

The four possible gray shades are:

0	White
1	Light gray
2	Dark gray
3	Black

FF48 - OBP0 - Object Palette 0 Data (R/W) - Non CGB Mode Only
-Controls Sprite Palette 0 similar to BGP

FF49 - OBP1 - Object Palette 1 Data (R/W) - Non CGB Mode Only
-Controls Sprite Palette 1 similar to BGP

FF68 - BCPS/BGPI - CGB Mode Only - Background Palette Index
-This register is used to index into background color palettes in the Gameboy Color

FF69 - BCPD/BGPD - CGB Mode Only - Background Palette Data
-This register is used to write background color data to the palette index

FF6A - OCPS/OBPI - CGB Mode Only - Sprite Palette Index
-This register is used to index sprite color palettes in the Gameboy Color

FF6B - OCPD/OBPD - CGB Mode Only - Sprite Palette Data
-This register is used to write sprite color data to the palette index

FF4F - VBK - CGB Mode Only - VRAM Bank
-This register selects the current Video Ram bank.

Memory:

VRAM tile data 0x8000 - 0x97FF

In this area of memory 8x8 pixel tiles. Each tile is 16 bytes where each 2 bytes describe a line in this fashion:

Byte 0: 0010_**0**110
Byte 1: 0110_**1**100

The first line describes the least significant bit while the second line describes the most significant bit. So for the highlighted pixel 5 in this tile it would point to the color 1.

There are two ways to address this tile data based on the LCDC register bit 4. If bit 4 is 0 background and window tiles will start at 0x9000 and be signed from 0x8800-0x97FF. If bit 4 is 1 background and window tiles will start at 0x8000 and be unsigned from 0x8000-0x8FFF.

VRAM background map 0x9800-0x9FFF

There are two background maps in this region. Each can be used for backgrounds or window overlays. The map is chosen with bit 3 of the LCDC register. If bit 3 is 0 the map is 0x9800-0x9BFF. If bit 3 is 1 the map is 0x9C00-0x9FFF.

Each byte of this map represents a number of a tile. Depending on which tile data table is being used the number is translated into an address and used to index the appropriate tile.

In the Gameboy Color in bank 1 at the same memory location as the background map, each byte corresponded to a set of attributes that helped described the palettes used for that tile and whether or not it was displayed.

Bit 0-2 Background Palette number (BGP0-7)
Bit 3 Tile VRAM Bank number (0=Bank 0, 1=Bank 1)
Bit 4 Not used
Bit 5 Horizontal Flip (0=Normal, 1=Mirror horizontally)
Bit 6 Vertical Flip (0=Normal, 1=Mirror vertically)
Bit 7 BG-to-OAM Priority (0=Use OAM priority bit, 1=BG Priority)

Sprite Attribute Table 0xFE00-0xFE9F

There are a total of 40 sprites per screen. In this region each sprite is made up of 4 bytes. These bytes correspond with the following:

Byte0 - Y Position
Specifies the sprites vertical position on the screen

Byte1 - X Position
Specifies the sprites horizontal position on the screen

Byte2 - Tile/Pattern Number
Specifies the sprites Tile Number (00-FF). This (unsigned) value selects a tile from memory at 8000h-8FFFh. In CGB Mode this could be either in VRAM Bank 0 or 1, depending on Bit 3 of the following byte.

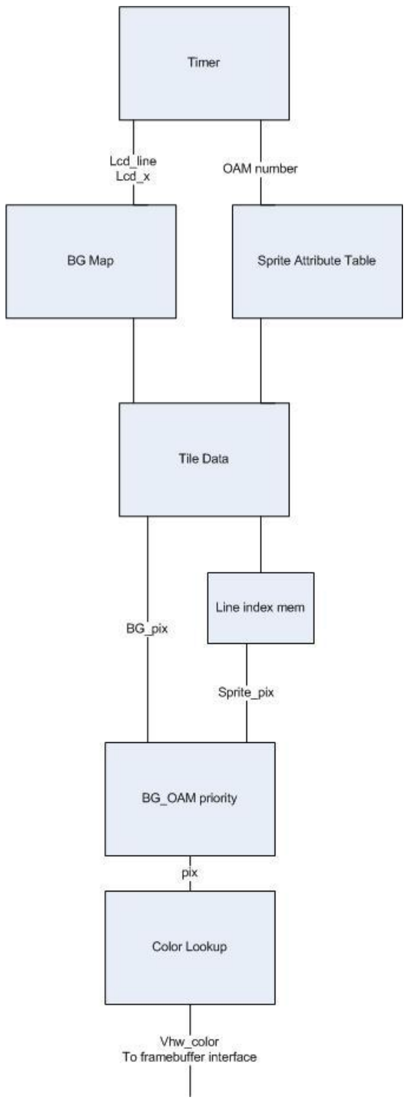
Byte3 - Attributes/Flags:
Bit7 OBJ-to-BG Priority (0=OBJ Above BG, 1=OBJ Behind BG color 1-3)
Bit6 Y flip (0=Normal, 1=Vertically mirrored)
Bit5 X flip (0=Normal, 1=Horizontally mirrored)
Bit4 Palette number **Non CGB Mode Only** (0=OBP0, 1=OBP1)

Bit3 Tile VRAM-Bank **CGB Mode Only** (0=Bank 0, 1=Bank 1)
Bit2-0 Palette number **CGB Mode Only** (OBP0-7)

Implementation:

In our implementation, we used a timer to count the cycles between modes. We would proceed from OAM mode 2 for about 80 cycles to BG mode 3 for about 160 cycles to h-blank mode 0 for about 200 cycles and then proceed to the next line. V-blank mode 1 takes about 4560 cycles and this is when most of the work from the CPU is done in VRAM.

In mode 2 every other clock cycle would be used to look up the next OAM, find out if it was on the current line and then place it in a line buffer if it was. In mode 3 the timer would be used to find the correct background map location to index into the correct tile. Once the tile was found the final background color was then calculated. Using the line buffer there would be a lookup if there was a sprite at the current pixel and if so the sprite would be written rather than the background pixel.



Framebuffer Interface

In order to place the pixel data on the DRAM for the framebuffer, we needed a way to properly transfer our data to DRAM. Our initial approach was to use PLB bus to get an access to the DRAM directly. However, we were not able to implement a module that works on PLB bus after many trials. Therefore, we decided to use the software to communicate with DRAM.

Our framebuffer interface gets the pixel data from the GPU and places them in the double buffer, which can contain a line in each. The double buffer was essential to communicate with the software since the OPB bus transfer to the software is not guaranteed. If we did not use the buffer, the software arbitrarily misses the pixel data and there would be many glitches on the output. Therefore, the framebuffer interface sends a line to the software and the line is placed in the DRAM so that the Xilinx's framebuffer module can send it out to the screen.

Results:

The video hardware in our implementation was partially working. The most notable missing entity was sprite support. There was not sufficient time to debug our sprite implementation and a lot of changes were made earlier to ensure that our sprite implementation was not interfering with our background results. Once the background results were confirmed we attempted to go back and implement sprites but failed to complete this task.

We were able to confirm that background maps were working correctly with our own tests and by running   the tetris ROM. In our own background map we wrote a custom gameboy assembly program which wrote a tile to VRAM tile data memory. The program then cycled through each byte in the background map and pointed the byte to that tile. The results were a repeating tile pattern on the screen.

When we ran the Tetris ROM we saw some of the correct tiles being displayed in the correct location. We believe that the incorrect tiles were a result of DMA transfers not having enough time to complete. We did not notice this problem in our own gameboy tests because we wrote to memory using the CPU rather than a DMA transfer. We were able to make the problem minimal by extending the V-blank period as long as possible in order to let the DMA transfer complete. This cut down on the number of tiles that looked incorrect.

When we ran the Mario Tennis ROM the background map and tile corruption was significant but we were able to confirm that the video hardware correctly scrolled.

Other Modules

Cartridge

For the simplest implementation possible, we only considered games which fit in a single 32 KiB contiguous ROM cartridge (as opposed to using banking) and didn't need any extra cartridge RAM. We also wrote our tests to use just unbanked ROM. This ROM was treated as asynchronous-read ROM, but as mentioned above, we had to use synchronous-read ROM and a faster clock for it to simulate asychronous-read ROM.

We prepared our cartridge ROMs in a couple of different ways in order to use them in simulation or synthesis. At first we used a z80 assembler which generated hex-encoded output, and a perl script which converted that into a text format appropriate for the readmemh() function supported in verilog by modelsim, which loaded the data into a byte array. We then switched to a different assembler (tniasm) which supported both z80 and gameboy instruction sets, but produced binary output. A different script using the 'hexdump' unix command was written to convert that to the readmemh() format. When we wanted to start running our own tests on the FPGA, we wrote another 'hexdump' based script to convert the binary into lines of verilog case statements, so it could be included in the middle of something like this:

```
always@* begin
    case(addr) begin
        `include rom_cart.inc
        default: data_out = 8'bxxxx_xxxx;
    endcase
end
```

This was sufficient for small programs, but wouldn't be very efficient for large actual roms. To allow large roms to be synthesized into BRAMs, we wrote yet another script to convert the binary to another ascii based form using eight 0 or 1 characters on a line, and wrote a module in vhdl to load it into a ram. Search google for "XST.pdf" for details on this technique.

Timer

The Timer device can generate periodic interrupts, and is controlled by a few memory-mapped registers.

FF04 - DIV - Divider Register
This register is incremented at rate of 16384Hz. Any write resets it to zero.

FF05 - TIMA - Timer counter
This register is incremented at a rate defined by TAC (below), and when it overflows it generates an interrupt and is reset to TMA (below).

FF06 - TMA - Timer Modulo
TIMA is reset to the value in this register when it overflows.

FF07 - TAC - Timer Control

```
Bit 2 - Timer Stop (0=Stop, 1=Start)
Bits 1-0 - TIMA rate select
00: 4096 Hz
01: 262144 Hz
10: 65536 Hz
11: 16384 Hz
```

This module was implemented with a cycle counter, which incremented each clock cycle in which the timer was started (TAC[2]). At a maximum value defined by TAC[1:0] this cycle counter incremented TIMA and restarted. The rest of the design followed from the specifications pretty directly.

SRAM

This module was implemented as a rather simple verilog byte array, plus some logic to interface with our bus. As mentioned above, we ended up changing the SRAM from asynchronous-read to synchronous-read-but-with-a-faster-clock for synthesis reasons.

DMA Device

This module has some complex behavior, and we didn't fully implement it. If we had fully implemented it, we might have a much more fluid result when running tetris or one of the other 32KiB Game Boy ROMs. Anyway, the DMA device had a few modes, and in each mode it copied two bytes per cycle. The way we implemented it (for simplicity), it was only possible to copy one byte in two cycles. However, when we switched to the synchronous memory with a faster clock, we could have used this faster clock to make the DMA meet the timing requirements from the perspective of the cpu core clock. We didn't get around to doing that.

The DMA device was inserted directly between the CPU and the rest of the bus. The DMA device either passed through the bus signals directly both ways, or if it was in the middle of a DMA operation, it disabled the CPU using the 'bus_en' signal, and used the bus as if it was the CPU. We made it this way because our bus was designed with only one master, the CPU. Actually, the DMA device didn't always pause the CPU when it was working, because during one type of DMA (OAM DMA), the CPU could still access "High Memory". So, we put the "High Memory" in the DMA device, and let the CPU's bus lines only work with this address space while the DMA device controlled the real bus lines.

We supported OAM DMA almost entirely correctly (we think), but we supported a non-hsync VRAM DMA at quarter speed, and we didn't support hsync VRAM DMA at all. We don't think Tetris used hsync VRAM DMA, but we think it suffered from slow non-hsync VRAM DMA. In both of the cases we supported, when we recieved a write to the memory-mapped register which initiated the DMA, three internal registers were written: src_addr, dst_addr, and bytes_left. The rest of the DMA device switched operation based on whether bytes_left was zero or not: doing DMA if bytes_left was non-zero, and letting the CPU use the bus if bytes_left was zero.

The whole thing is a bit complicated, and as described above, we didn't fully implement it. We think, however, that it would have been possible to implement a fully functional DMA device with same topology we used (described above and in the overview diagram at the beginning of this report), if we additionally gave the DMA device the faster memory clock.

Controller

Gameboy joypad has eight buttons/direction keys that are arranged in form of a 2x4 matrix. By writing to the designated register, we can decide either button or direction key is selected.

Bit 7 - Not used		
Bit 6 - Not used		
Bit 5 - P15 Select Button Keys	(0=Select)	
Bit 4 - P14 Select Direction Keys	(0=Select)	
Bit 3 - P13 Input Down or Start	(0=Pressed)	(Read Only)
Bit 2 - P12 Input Up or Select	(0=Pressed)	(Read Only)
Bit 1 - P11 Input Left or Button B	(0=Pressed)	(Read Only)
Bit 0 - P10 Input Right or Button A	(0=Pressed)	(Read Only)

We used PC keyboard to simulate the joypad functionality. The Xilinx has PS2 controller module which can interpret the keyboard input. Whenever a key is pressed, scancode is generated by the PS2 Controller. We needed to change the existing PS2 controller to send out the scancode to our joypad module so that we can decode the scancode and send an appropriate interrupt to CPU. Whenever the key is released, the scancode is repeated followed by another scancode 'F0'. In this way, we are able to recognize which key is currently pressed and released. Whenever there is change in the register, the interrupt is sent to CPU, and CPU asks whether if the key was button or directional. Depending on bit 4/5, CPU can recognize which button/directional key is pressed.

Result

We were able to get an original Gameboy ROM to run on our CPU displaying with our video hardware and responding to input from our keyboard. We were very pleased with these results seeing how we were having issues running an original ROM and could not get any screen to appear. Although we were able to show that our individual modules were working based on the tests we wrote in assembly, this would not have achieved the same effect as running an original ROM.

We did fall short on the multiplayer requirement of the project. With some more time we may have been able to connect the output pins of the board to the link cable of another Gameboy. We purchased a Gameboy for this reason but due to time constraints were not able to start this part of the project.

Sound was also a module we were not able to incorporate in our project. Sound is interesting part of the Gameboy and we would have liked to try an get it working but we felt that the CPU and display were more important and spent more time solving their issues rather than working on sound.

Conclusion

We were able to accomplish a lot with this project. Many of our accomplishments occurred closer to the deadline then we would have liked. This is what happens though when 3 determined people work under time constraints together. If we could have kept the same productivity throughout the semester we might have been able to accomplish additional modules.

The design decision to switch to the original Gameboy came a bit too late. If this decision would have been made earlier we would have been able to assume a lot of simplifications when designing both the CPU and the Video Hardware. On top of simplifying the project this would have cut down our synthesis time throughout the project.

If we were able to do this project again we would have our design decisions hammered out by the first couple weeks. We would be sure to schedule times for all of us to meet up, discuss the issues we are having, and integrate our changes together.

Individual Reports

Pierce Lopez

The Game Boy project was a bit daunting, and I'm amazed when I look at our FPGA running the real Tetris ROM, even though I had hoped for more (Game Boy Color, link cable) when the project started. Having traced through the first few thousand cycles of running Tetris while debugging, I might be more amazed than my partners :)

We primarily used a compilation of GameBoy specifications by emulator and homebrew writers, called "pandocs". I think that my work on the CPU was the only work which made significant use of other documentation besides pandocs. For working on the CPU I found the Z80 User Manual useful (it described Z80 assembly instructions in great detail, including every bit in the opcodes). I also used a couple of other GameBoy assembly references for those instructions which differed from the Z80 and for which pandocs was not specific enough. I was surprised by a few obvious technical errors in the Z80 User Manual in the bit rotation ops section, because the Z80 was very widely used and the official manual looked rather professional and official.

I did all the work on the cpu core. I started with a Z80 design which Wan found on opencores.org. However, the efficient but extremely difficult to read and modify design inclined me to procrastinate. At a point about half way through the project, I had dived to in really get the needed modifications done, and decided that I would really have to write my own core from scratch. At that point I had a pretty good feel for the instruction set and the performance requirements (and I had taken Computer Architecture the previous semester) so I was able to complete the new core (except for interrupts) in two weeks. Very few bugs were found in it later on... maybe three of over 256 instructions. I was really careful to get the microcode correct when I added a new instruction, because I knew that we would want to be able to trust that at least the CPU was working correctly when we were debugging the rest of the design.

Because I was the CPU designer, I was also the most familiar with Game Boy assembly, so I ended up writing most of the tests. Our initial tests just involved ROM and registers and were checked by register contents printouts in simulation. Later tests were designed to use RAM, run on the FPGA, take input from the input module, and demonstrate results by working with the video hardware to show simple blocks on the screen.

When the end of the project was coming up and we were still having issues with BRAMs and we hadn't written a banked cartridge module yet, I convinced my partners that we should switch to the original Game Boy from the Game Boy Color because some original Game Boy games had very small ROMs with no banking. Unfortunately this threw a wrench in Sean's video hardware design, and it took us some time to debug it after he was forced to hack it back to normal-gameboy-only. It's unclear exactly how much of the problems with the video hardware were caused by this last minute switch I pushed for, but I'm afraid (and sorry) that it might have had a detrimental effect on his designs.

Both Sean and Wan were very competent and great as partners, and I felt I could trust them to do their parts, and to generally be very reasonable.

Sean Moorman

I initially pitched the idea for doing the Gameboy as a project. I knew there would be plenty of documentation and that we could use existing software emulators to help us in the process. I helped to divide the project into parts and we decided that Pierce would do the CPU, I would do the video hardware, and Wan would do the framebuffer interface as well as a number of other miscellaneous modules.

After the pitch my contribution to the project was designing and implementing the Video Hardware. My initial implementation of the Video Hardware was for the Gameboy Color. When we became concerned that synthesis time was taking way too long and that we would not have enough LUTs to put the entire design on the board we switched over to the original Gameboy.

At this point I had to clip the extra banks of memory and registers that were not relevant to the Gameboy Color. We then had a working Video Hardware module for the Gameboy. With Pierce's help with the assembly language I was able to write a handful of tests that tested the functionality of the tile data region and the background map.

After these tests were confirmed I started to write tests to check the sprite functionality. When I was trying to get the background map to work I made some changes that eliminated the functionality of sprites to ensure the background map was correct. With a combination of these changes and the switch from Gameboy Color to Gameboy original I was not able to get sprites to work.

Besides the Video Hardware I assisted in the administration of the project. I also helped verified other modules and brainstorm design ideas for our project.

I am pleased with what we got to work on this project. With more time we probably would have been able to resolve the issues with the DMA transfers. I also could have had time to get sprites to work and ensure that window overlay was working correctly.

Wan Lee

In the beginning of the project, I was not familiar with the FPGA and did not know how the project would turn out. Implementing the GameBoy Color sorely in hardware seemed a bit overwhelming, but I was excited to work with great two teammates for a big scale project. Since I did not know the general way to approach GameBoy hardware, the teammates helped me a lot on the way.

My concentration for the project was the framebuffer interface and the other peripheral modules such as controller and sound. I spent most time trying to implement the framebuffer on the PLB bus. The documentation for implementing an IP on PLB bus was hard to find, I had hard time with trying to figure out how I can get an access to the DRAM. Since synthesizing the hardware took about 30 to 40 minutes at that time, I ended up spending half the time on the synthesis. The forum on the internet was helpful, but there were not many answers to my question (there were many unanswered questions on the forum). After giving up on the PLB, Pierce and I were able to implement the framebuffer interface with the software support. I was impressed with the animation on the screen and the fact that we connected all the GameBoy modules to make up the animation.

The challenge came again with the controller. I had to interpret how the Xilinx's PS2 Controller recognizes the signal from the keyboard. The codes for the PS2 Controller were more complicated than what I expected, but eventually I was able to find out where the scancode is generated. If I had better documentation on the PS2 Controller, it would have been possible to save much time.

Throughout the semester, I've learned so much from this project. I did not have much knowledge about hardware and this became a huge opportunity for me to work with the hardware implementation. Although I know that time constraint was tight for us to make the full-functioning GameBoy, I'm still very impressed with what we have accomplished. In general, I feel sorry for the team since my knowledge was very limited for this project. However, I thank my teammates for helping me a lot on the way and that they never hesitated to teach me anything I needed to know.