# Contents

List of Figures ........................................ 3

List of Tables ......................................... 4

Introduction ........................................... 5

## Design and Implementation .......................... 7

- System Design ........................................ 7
- Memory Map .......................................... 9
- GB-Z80 CPU .......................................... 9
- Instructions ......................................... 12
- Interrupts ............................................ 13
- Pixel Processing Unit (PPU) ......................... 13
- Video Timing ........................................ 16
- OAM DMA ............................................ 17
- Timer .................................................. 18
- Sound ................................................. 19
- Joypad ................................................. 20
- Serial .................................................. 21
- Cartridge ............................................ 24
- Header ............................................... 24
List of Figures

1 The Game Boy (Left) and its cartridge (Right) ........................................ 5
2 Top-level System Block Diagram .......................................................... 7
3 The Game Boy Memory Map [2] ............................................................. 10
4 State Diagram of our GB-Z80 ................................................................. 11
5 The CALL instruction [4] (left) and the decoded RISC instructions (right) . 12
6 The RISC instructions for interrupt handling ........................................ 13
7 The Background scrolling [6] ................................................................. 14
8 The Window [6] ..................................................................................... 15
9 The PPU Timing ................................................................................... 16
10 OAM DMA ............................................................................................ 17
11 Timer block diagram 1 [7] ................................................................. 18
12 Timer block diagram 2 [7] ................................................................... 19
13 Timer block diagram 3 [7] ................................................................... 20
14 Joypad schematic [6, 7] ....................................................................... 21
15 Serial I/O registers [6] .......................................................................... 22
16 Serial timing chart [6] .......................................................................... 23
18 MBC1 Memory Map [6] ......................................................................... 28
19 MBC3 Memory Map [6] ......................................................................... 30
20 MBC5 Memory Map [6] ......................................................................... 31
List of Tables

1. Cartridge header information [6, 7] ................................................. 25
2. MBC1 memory map and register description [4, 6] .............................. 27
3. MBC3 memory map and register description [6] ................................... 29
4. MBC5 memory map and register description [6] ................................... 31
5. List of Game Boy games emulated ...................................................... 32
6. Project contribution ........................................................................... 33
Introduction

The Game Boy is an 8-bit handheld console created by Nintendo. It was released in North America on July 31, 1989. The Game Boy and Game Boy Color combined have sold 118 million units worldwide, making it the 3rd most popular video-game console in history [8].

On the front, there is a black and white dot matrix display, capable of displaying $160 \times 144$ dots in 4 different gray scales. There is also a directional pad (D-pad) as well as A, B, SELECT and START buttons. On the top there is an on-off power switch and a slot for Game Boy
cartridges. On the right side, there is a volume control dial and a serial port that supports multi-player games or external peripherals such as the Game Boy Printer. There is another dial on the left side that adjusts contrast.

Inside the Game Boy, there is a custom 8-bit Sharp LR35902 SoC. The CPU in the SoC is usually referred to as the GB-Z80. Its internal registers are similar to the Intel 8080 but it has some instructions that were introduced in the Zilog Z80. The Game Boy has 8kB of SRAM as work RAM and 8kB of SRAM as video RAM. There is also a built-in 256-byte bootstrap ROM that validates the cartridge header, scrolls the Nintendo logo, and plays the boot sound. The console can support up to 64 MB of ROM with the help of memory bank controllers (MBCs) inside cartridges.

In this project, our goal is to make a cycle-accurate Game Boy hardware emulator on the Terasic DE1-SoC Board capable of smoothly running games. We implemented the LR35902 SoC and SRAMs on the Cyclone-V FPGA, game cartridge ROM and RAM banks on the SDRAM, and joypad controller on Linux running on the ARM core. We implemented the video display using our own VGA core on the FPGA, and displayed the Game Boy video output on a 1280×1024-resolution LCD monitor. We streamed the audio signal out using the Intel University Program IP for the Wolfson WM8731 CODEC on the DE1-SoC board and played it out with a pair of speakers.
Design

System Design

The top-level system block diagram is shown in Figure 2. This block diagram is based on our implementation of the Game Boy, which differs from the real Game Boy but achieves the same functionality. The \textbf{DATA} double arrows represent bidirectional (reads and writes) data flow between a bus master and a bus slave. The \textbf{MMIO} double arrows represent memory mapped I/O (MMIO) style data flow between a bus master and a bus slave. The \textbf{MMIO/INT} double arrows represent MMIO style data flow as well as interrupts from a bus slave to a bus master. The \textbf{AVALON-MM} double arrows represent the Intel Avalon Memory Mapped
Interface. The **AVALON-ST** double arrow represents the Intel Avalon Streaming Interface. The **VGA** double arrow represents the Video Graphics Array (VGA) interface.

Orange blocks are implemented on the FPGA using SystemVerilog. The GB-Z80 is the CPU of the Game Boy. The pixel processing unit (PPU) is used to generate graphics. The Timer module is used by the CPU to keep track of clock cycles. The Joypad block emulates the buttons on the Game Boy. The Serial module emulates the serial interface on the Game Boy.

There are 3 main data buses on the Game Boy. One bus connects to the work RAM and cartridge, another connects to the video RAM and the last one connects to the OAM. Only one master can write to a bus at one time. In a real Game Boy, reading on a bus that is being written by another master results in undefined behavior, depending on the tri-state bus behavior of the device. But in our implementation, reading while the other master is writing results in reading **8'hFF**.

To better support modern FPGAs, tri-state buses and latches in the original Game Boy are not used. Instead, we use separate data lines for read and write. We use multiplexers to address individual peripherals. We also use edge-sensitive flip-flops instead of latches.

When the CPU wants to address its peripherals, it sends the request Address, Read/Write flag, and Data to the Memory Management Unit (MMU). The MMU then decides which peripheral to enable or which bus should this address be put onto. The MMU also performs arbitration: when 2 devices want to access the same address, it will give access to the devices with the higher priority. The MMU can also perform direct memory access (DMA) so that it can bypass the CPU and perform OAM memory copy itself. Also, when a peripheral requests an interrupt, the MMU will relay the interrupt to the CPU based on the interrupt enable flag and save the interrupt request in the interrupt flag register.

The Qsys system is shown in Appendix A.
Memory Map

The Game Boy CPU uses memory mapped I/O to access all its peripherals. The memory map is shown in Figure 3. At the beginning when the Game Boy boots up, the area 0000-00FF is mapped to the internal bootstrap ROM. It automatically unmaps itself after the bootstrap completes and this area is then mapped to cartridge ROM. 0000-7FFF is the cartridge ROM area, which is 32kB in total. 8000-9FFF is the video RAM area. The region A000-BFFF is reserved for the RAM on the cartridge, usually used for saving game data. C000-DFFF is the internal work RAM area. E000-FDFF is called the echo RAM; accessing this area is equivalent to accessing C000-DDFF. FE00-FE9F is the Object Attribute Memory (OAM), which contains 160 bytes of RAM used to store sprite information. FF00-FF7F contains the MMIO registers for the joypad, serial, CPU interrupt flag, timer, PPU and sound. The region FF80-FFEF is usually called high RAM, a 127-byte program stack used by the CPU. Address FFFF contains the interrupt enable register used by the CPU to specify which interrupts are active at the time.

GB-Z80 CPU

The data line of the GB-Z80 is 8 bits wide while the address line is 16 bits wide. There are 6 general purpose registers, namely B, C, D, E, H, and L. They can be combined in pairs creating 3 pairs of 16-bit registers BC, DE, and HL. There is also an 8-bit register A, used for arithmetic logic unit (ALU) results, and a 4-bit register F used for ALU computation flags. For instance, when there is a overflow or the result after computation is 0, certain flags will be set in the F register. There are also two 16-bit registers: the stack pointer SP and Program Counter PC. SP is used to keep track of the current stack address and PC is used to keep track of the address of the next command to be fetched. The stack pointer does not necessarily point to the dedicated 127-byte high RAM stack; it can also point to work RAM.
or some other regions.

The GB-Z80 is a complex instruction set computer (CISC) CPU, which means its instructions can take variable clock cycles. Compared to the Intel 8080, bit-manipulation instructions from the Z80 were included. While instructions include the parity flag, half of the conditional jumps and I/O operations were removed. I/O is performed through memory load/store instructions.

For the GB-Z80, all instructions are executed in multiples of 4 clock cycles. It can either be 4, 8, 12, 16, 20 or 24 clock cycles long. There can be 512 possible instructions or opcodes; 256 are typical instructions, the other 256 are CB-prefixed instructions. When the CPU fetches a CB-prefixed instruction, it will fetch for another opcode right after.
There are 5 interrupt lines to the CPU, namely V-Blank interrupt, LCD controller interrupt, Timer interrupt, Serial interrupt and Joypad interrupt.

To simplify the design and meet the time requirement of this project, we did not try to reproduce the original structure of the Intel 8080 or Z80. We used a simple 2-stage reduced instruction set computer (RISC) CPU with a front-end decoder to perform the equivalent operations.
CALL nn
Unconditional function call to the absolute address specified by the operand nn.

<table>
<thead>
<tr>
<th>Opcode + data</th>
<th>0b11001101 + LSb of nn + MSb of nn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>3 bytes</td>
</tr>
<tr>
<td>Duration</td>
<td>6 machine cycles</td>
</tr>
<tr>
<td>Flags</td>
<td>-</td>
</tr>
<tr>
<td>Timing</td>
<td></td>
</tr>
</tbody>
</table>
| Pseudocode    | opcode = read(PC++)  
|               | if opcode == 0x03D:  
|               | nn = unsigned_16(lsb=read(PC++), msb=read(PC++))  
|               | write(−SP, msb(PC))  
|               | write(−SP, lsb(PC))  
|               | PC = nn                             |

Figure 5: The CALL instruction [4] (left) and the decoded RISC instructions (right)

Instructions

Take the CALL nn instruction as an example, illustrated in Figure 5. It takes 24 cycles to complete. It reads 2 consecutive bytes from addresses pointed by PC, stores the current PC in to stack, and loads the 2 bytes previously read into PC. For our GB-Z80, first the front-end reads and decodes the instruction using 2 clock cycles. The CALL nn instruction is subsequently decoded into 7 RISC instructions, each taking 2 clock cycles to complete. Thats only 16 cycles in total so the rest of opcodes are filled with NOP (not shown in the figure). X and T are two registers I introduced in my GB-Z80 to store temporary data. The first RISC opcode LD_XPC means “load the data at address equal to PC to register X”. In the Memory Access stage, the GB-Z80 requests for a memory read at the PC, and in the Execute/Write Back stage, it gets the requested data and saves it into X. Also it increments the PC by 1. The DEC_SP opcode means “Decrement the SP register by 1”. In our GB-Z80, it does nothing in the Memory Access stage, and performs a 16-bit subtraction on the SP register. The LD_SPPCh command means “Load the high byte of PC onto the Stack”. In the Memory Access stage, our GB-Z80 asks for a write at the address equal to the SP, and in the Write Back stage the value in the high byte of the PC is written to the stack. The JP_TX opcode means “Jump to the address equal to the register pair TX”. In our GB-Z80, it does...
nothing in the Memory Access stage, and performs a 16-bit load from the TX to the PC.

Interrupts

Interrupts are checked at the end of an instruction. If the IME is set and there is an interrupt flag in the IF register, and the interrupt is also set in the IE register, that interrupt routine is then served. Each interrupt takes 20 clock cycles to complete and the instructions are shown in Figure 6. These are based on the interrupt test set. The GB-Z80 will first clear the IME, preventing any other interrupt. Then it will push the high byte of PC on to stack. Afterwards, it saves the current interrupt request in a temporary location. It then resets the interrupt flag and pushes the low byte of PC on to stack, and finally it jumps to the interrupt vector. The LATCH_INTQ opcode saves the current interrupt in a temporary register. The RST_IF opcode asks for a memory write, and clears the current interrupt in the IF register.

```
define DECORDER_INTR(addr)
begin
    RISC_OPCODE[0] = DI;
    RISC_OPCODE[1] = DEC_SP;
    RISC_OPCODE[2] = LD_SPPCh;
    RISC_OPCODE[3] = LATCH_INTQ;
    RISC_OPCODE[4] = RST_IF;
    RISC_OPCODE[5] = DEC_SP;
    RISC_OPCODE[6] = LD_SPPCl;
    RISC_OPCODE[7] = RSTאני addr;
    NUM_Tcnt = 6'd20;
end
```

Figure 6: The RISC instructions for interrupt handling

Pixel Processing Unit (PPU)

The Game Boys visible screen area is 160×144 pixels. Like most other consoles of that era, the Game Boy did not have enough memory or bandwidth to hold a framebuffer in
memory. Instead, a tile system is employed. A set of bit maps is held in memory and a map is built using references to these bitmaps. The advantage is that one tile can be used repeatedly through the map, simply by using its reference. The Game Boys tiled graphics system supports tiles of 8×8 pixels for Background and Window, and tiles of 8x8 or 8x16 pixels for sprites.

Figure 7 shows that the background map can support 256×256 pixels whereas the display only uses 160×144 pixels, so there is a scope for the background to be moved relative to the screen. The PPU achieves this by defining a point in the background that corresponds to the top left of the screen. By moving this point between frames, the background can scroll on the screen. For this reason, definition of the top left corner is held by 2 PPU registers,
SCX and SCY.

Figure 8 depicts another drawing layer called Window in the Game Boy. It is very similar to the background but it can not scroll. The programmer can only set the starting top left corner of the window, and it will be drawn all the way to the bottom right of the screen. Most games use it as a head-up display or menu display.

![Window Diagram](image)

Figure 8: The Window [6]

There are 2 maps of 32×32 tiles that can be held in memory, and only one can be used for display at a time. There is space in the Game Boy for 384 tiles, so half of them are shared between the maps. One map uses tile numbers 0 to 255 and the other uses numbers between -128 and 127 for its tiles.

The Game Boy can handle 4 shades of gray. Representing one of these four colors in the tile data takes 2 bits, so each tile in the tile data set is held in 8×8×2 bits or 16 bytes. One additional complication for the Game Boy is that each of the four possible values can correspond to any of the four colors. The palettes are mainly used to create easy color changes for the tile set.

In our Game Boy design, we put a framebuffer in place of the Game Boy LCD display. The framebuffer is a 160×144×2-bit dual-port dual-clock SRAM. The Game Boy can write to it
at its own clock speed and the VGA core can read pixels out at the VGA clock speed.

**Video Timing**

The Game Boy video hardware simulates a cathode-ray tube (CRT) display in its timing. It also has the H-Blank and V-blank period. However, since the Game Boy uses an LCD display, it can do something a CRT display cannot: it can stop clocking the LCD whenever it wants to. This occurs when the PPU is not ready to send out a pixel yet, resulting in variable length in the rendering period and H-blank period. As seen in Figure 9, the PPU always spends 456 clock cycles on a scan line and 4560 clock cycles for a whole screen refresh. At the beginning of a scan line, there is a fixed 80-clock cycle OAM search area. This area is used to determine whether there are any sprites on the current scan line and help sprite fetching in the rendering area. During OAM search, it will iterate through the OAM and find the first 10 sprites on the current scan line. When it finds a sprite, it will take a note of its X position and its position in the OAM, storing them in a local OAM.

![Figure 9: The PPU Timing](image)
After the OAM search comes the rendering area. In the rendering area, pixels are shifted out from the PPU to the LCD (or for our design, a framebuffer). The rendering area can take 174 to 291 clock cycles to complete, depending on the SCX, sprite location, and number of sprites on the scan line. At the end of a scan line is the H-Blank period. H-Blank varies from 85 to 202 dots; in this area, the PPU will not accessing any memory. After 144 scan lines, the PPU enters V-Blank; in this area, it will not access any memory either. Notice that the PPU only spends less than 4 clock cycles in line 153, and it spends the rest of the clock cycles in line 0 in V-Blank.

Details of rendering timing is described in [5] and the presentation slides. We did not implement the extra clock cycles when there is a sprite at X=0.

OAM DMA

![OAM DMA Diagram](image)

Figure 10: OAM DMA

The OAM is usually filled by a DMA. CPU can write the source address into the DMA register. The MMU will move 160 bytes from the source address to the OAM. The source can be on the work RAM bus or the video RAM bus. While DMA is running, the CPU
cannot access the bus that is the source of the DMA or the OAM. In the original Game Boy, the OAM is a slave of the PPU and the OAM is a part of the PPU logic; however, we implemented the OAM as a peripheral of the CPU. This makes memory management easier since the CPU does not have to go through the PPU to access the OAM.

Timer

Figure 11: Timer block diagram 1 [7]

The timer peripheral is basically a big counter with automatic reload on overflow. It works as a timer as you can ask it to send an interrupt after a certain number of clocks. Some games like Tetris use the counter in it as a time seed for random number generation. The logic block diagram is shown in Figures 11, 12, and 13.
The Game Boy has four sound channels. Two square waves with adjustable duty cycle, a programmable wave table, and a noise generator. Each has some kind of frequency or pitch control. The first square channel also has an automatic frequency sweep unit to help with sound effects. The square and noise channels each have a volume envelope unit to help with fade-in or fade-out sound effects. On the other hand, the wave channel only has limited manual volume control. Each channel has a length counter that can silence the channel after a preset time to handle note duration. Each channel can be individually mapped to the left, right, or both audio outputs. There is also a master volume control register that can independently adjust left and right outputs.

We used the Intel University Program CODEC IP to configure the CODEC and sent the
samples out through the Avalon Streaming interface to the CODEC.

**Joypad**

The Game Boy joypad I/O register is located at CPU address FF00. As shown in Figure 14, the eight keys are arranged in the form of a $2 \times 4$ matrix, where P10-P13 are input ports connected to pull-up resistors and P14-P15 are output ports. The CPU regularly sets P14-P15 low to read which keys were pressed. During this time, whichever key was pressed closes
the signal path and the corresponding input port is pulled low by the diode. For example, if P15 is set to 0 and button A is pressed, then P10 will be 0 (P11-13 stay at logic 1); if the RIGHT key was pressed, then P10 will be logic 1.

Our joypad module was fulfilled by a USB keyboard and a device driver that communicates with a joypad peripheral on the DE1-SoC. The peripheral was generated in Qsys and added to the Device Tree. Our user space program (Appendix C.2) can configure any USB keyboard keys as joypad keys (except ESC, SPACE, and modifiers); the SPACE key is reserved for enabling double speed. Whenever any configured joypad keys are pressed, the joypad status is sent to the kernel module.

Serial

The serial interface allows two Game Boy devices to transfer data with one another, conventionally via a link cable. For example, players can trade Pokemon and battle each other when playing compatible Pokemon games, or play 2-person Tetris.

As shown in Figure 15, serial I/O is controlled by the SB and SC registers, located at CPU addresses FF01 and FF02. The MSB of the SC register controls the serial transfer and the LSB selects the clock used. One Game Boy acts as the master and uses its internal clock.
at 8.192kHz, while the second one acts as the slave and uses an external clock (typically supplied by the first Game Boy, but can go up to 500kHz).

The timing chart and block diagram are illustrated in Figures 16 and 17, respectively. Data is first set in the SB register. Then, the MSB of the SC register is set to 1, initiating the transfer; during this time, read and write access to the SB register is disabled. Sending and receiving 8-bit data occur simultaneously. The data in the SB register is shifted leftward by a bit at every falling edge of the clock and the SOUT port outputs the highest bit; input data from the SIN port is shifted in the LSB of the SB register at every rising edge of the clock. After 8 clock counts of a 3-bit counter, the MSB of the SC register is set to 0 and an interrupt is sent to the CPU, signaling the completion of a byte transfer.

Figure 15: Serial I/O registers [6]
Figure 16: Serial timing chart [6]

Figure 17: Serial block diagram [6]
Cartridge

In the interests of time, money, and convenience, we developed a virtual cartridge module that loads ROM files downloaded from online onto the DE1-SOC (as opposed to buying physical cartridges and soldering wires onto GPIOs) to play various games. The user space program (Appendix C.2) reads a specified binary ROM file and loads its contents onto the on-board SDRAM. The Unix system call `mmap(2)` maps the virtual addresses used by the program to physical addresses of the SDRAM, enabling direct data manipulation from user space. Information such as ROM and RAM size, MBC type are read from the cartridge header region and used to configure the cartridge module.

Certain games such as *Pokemon Yellow* and *Legend of Zelda: Link’s Awakening* include an internal battery (with an expected lifespan of 10 years) in their physical cartridges to save game progress in RAM. This enables the player to continue where they left off in the game even after powering off the Game Boy. Virtual cartridges store game data in binary files with .sav extensions. After our user space program loads the game ROM into SDRAM, any SAV file with the same name is automatically loaded into a dedicated RAM region in SDRAM. Upon pressing the ESC keyboard key, Game Boy emulation stops running, game data in SDRAM is read by the program, and the SAV file is overwritten.

Header

The internal information of the cartridge is contained in a header region located at addresses 0100-014F. Table 1 summarizes the values in this region.
<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100-0103</td>
<td>Entry point</td>
<td>The program jumps to this execution point after displaying the Nintendo logo, and then starts the main program</td>
</tr>
<tr>
<td>0104-0133</td>
<td>Nintendo logo</td>
<td>Defines the bitmap of the Nintendo logo displayed when the Game Boy is powered on; will not run if these bytes are incorrect</td>
</tr>
<tr>
<td>0134-0143</td>
<td>Game title</td>
<td>Title of the game in UPPER CASE ASCII; remaining bytes are filled with 00’s if it is less than 16 characters</td>
</tr>
<tr>
<td>0143</td>
<td>CGB flag</td>
<td>Denotes Game Boy Color compatibility</td>
</tr>
<tr>
<td>0144-0145</td>
<td>New licensee code</td>
<td>Two character ASCII licensee code that specifies the company or publisher of the game; only used for games released after the Super Game Boy (SGB)</td>
</tr>
<tr>
<td>0146</td>
<td>SGB flag</td>
<td>Indicates if the game supports SGB functions</td>
</tr>
<tr>
<td>0147</td>
<td>Cartridge type</td>
<td>Specifies the hardware used in the cartridge (MBC, battery, rumble, etc.)</td>
</tr>
<tr>
<td>0148</td>
<td>ROM size</td>
<td>ROM size and number of banks</td>
</tr>
<tr>
<td>0149</td>
<td>RAM size</td>
<td>RAM size and number of banks (if any)</td>
</tr>
<tr>
<td>014A</td>
<td>Destination code</td>
<td>Indicates where the product is intended to be marketed</td>
</tr>
<tr>
<td>014B</td>
<td>Old licensee code</td>
<td>Specifies the company or publisher of older games</td>
</tr>
<tr>
<td>014C</td>
<td>Mask ROM version number</td>
<td>The version number of the game</td>
</tr>
<tr>
<td>014D</td>
<td>Complement check</td>
<td>Contains the checksum across header bytes in addresses 0134-014C; game will not run if this is incorrect</td>
</tr>
<tr>
<td>014E-014F</td>
<td>Global checksum</td>
<td>Contains a checksum across the entire cartridge (except for two bytes); the Game Boy in reality ignores this value</td>
</tr>
</tbody>
</table>
SDRAM

Most games with multiple banks of ROM are too big to be fit onto the DE1-SoC’s on-chip RAM. Hence, we had to use the SDRAM to store them. We ran the SDRAM at 16 times the speed of the Game Boy clock and since the column access strobe (CAS) latency is 3, we thought this was fast enough for the Game Boy to read it, but it did not work. We found that the automatic refresh used by the SDRAM might be the culprit. When we want to access the SDRAM and if its refreshing, it would take much longer to get valid data. But the Game Boy expects SRAM behavior; it always wants valid data after a fixed delay. Because we used a 16-times faster clock, we knew exactly when the Game Boy clock will rise. So when the Game Boy clock is about to rise and the data is still not ready yet, we stop the Game Boy clock and wait for the data. So from the Game Boy’s point of view, data is always ready before the next clock cycle. The intermittent stop of the clock is not perceivable to the human eye.

Memory Bank Controller (MBC)

Bank switching is a technique that increases the amount of usable memory beyond what the processor can address at a time. This allows a system to be configured differently at different times based on need by switching between various banks of memory. For example, in the context of video games, the ROM bank that contains the bitmap of the start screen can be switched out once the game is underway.

Many Game Boy games embed MBC chips in their cartridges to expand the available address space and store larger game content. The most common MBC chips for the Game Boy are MBC1, MBC3, and MBC5, which we all implemented in the project.
MBC1

MBC1 is the first MBC chip for the Game Boy and the foundation of newer MBC chips. It contains four registers that control the behavior of the MBC. The ROM bank number is controlled by two registers, so the effective ROM bank number is the concatenation of the 2-bit BANK2 and 5-bit BANK1. Table 2 and Figure 18 depict the memory map and register functions of the MBC1.

<table>
<thead>
<tr>
<th>Address</th>
<th>Read/Write</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-3FFF</td>
<td>Read</td>
<td>ROM Bank 00</td>
<td>Contains the 16kB of ROM bank 00 in ROM banking mode or of ROM bank 32\times BANK2 in RAM banking mode*</td>
</tr>
<tr>
<td>4000-7FFF</td>
<td>Read</td>
<td>ROM Bank 01-7F</td>
<td>Contains any of the other 16kB banks of ROM; however, banks 20h, 40h, 60h cannot be selected</td>
</tr>
<tr>
<td>A000-BFFF</td>
<td>Read/Write</td>
<td>RAM Bank 00-03 (if any)</td>
<td>Addresses an external RAM bank (up to 8 kB)</td>
</tr>
<tr>
<td>0000-1FFF</td>
<td>Write</td>
<td>RAM Enable</td>
<td>Writing 0Ah enables RAM; 00h disables RAM</td>
</tr>
<tr>
<td>2000-3FF</td>
<td>Write</td>
<td>ROM Bank Number</td>
<td>Selects lower 5 bits of ROM (BANK1); if 20h, 40h, 60h are written, 21h, 41h, 61h are selected respectively</td>
</tr>
<tr>
<td>4000-5FF</td>
<td>Write</td>
<td>RAM Bank Number/Upper Bits of ROM Bank Number</td>
<td>Selects a RAM bank (00-03) or upper two bits of ROM bank number (BANK2), depending on the ROM/RAM mode</td>
</tr>
<tr>
<td>6000-7FF</td>
<td>Write</td>
<td>ROM/RAM Mode Select</td>
<td>00 = ROM banking mode (max 8kB RAM, 2MB ROM); 01 = RAM banking mode (max 32kB RAM, 512kB ROM)</td>
</tr>
</tbody>
</table>

* As described in [4]; other documentation say this area contains ROM bank 00 only

Table 2: MBC1 memory map and register description [4, 6]
MBC3

MBC3 includes a built-in Real Time Clock (RTC) to track time in Game Boy games such as *Harvest Moon* or Game Boy Color games such as *Pokemon Crystal*. The RTC requires an external 32.768 kHz quartz oscillator and battery to tick even when the Game Boy is powered off. There are also four registers that control the data interaction between the cartridge and the Game Boy. Unlike MBC1, MBC3 has independent registers to address ROM and RAM banks in addition to RTC clock counters. Table 3 and Figure 19 illustrate the memory map and register functions of the MBC3.
<table>
<thead>
<tr>
<th>Address</th>
<th>Read/Write</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-3FFF</td>
<td>Read</td>
<td>ROM Bank 00</td>
<td>Always contains the 16kB of ROM bank 00</td>
</tr>
<tr>
<td>0000-3FFF</td>
<td>Read</td>
<td>ROM Bank 01-7F</td>
<td>Contains any of the other 16kB banks of ROM; unlike that of MBC1, accessing banks 20h, 40h, 60h is supported</td>
</tr>
<tr>
<td>A000-BFFF</td>
<td>Read/Write</td>
<td>RAM Bank 00-03 (if any) or RTC Register 08-0C</td>
<td>Addresses an external 8kB RAM bank or RTC register</td>
</tr>
<tr>
<td>0000-1FFF</td>
<td>Write</td>
<td>RAM and Timer Enable</td>
<td>Writing 0Ah enables RAM and RTC registers; 00h disables both</td>
</tr>
<tr>
<td>2000-3FFF</td>
<td>Write</td>
<td>ROM Bank Number</td>
<td>All 7 bits written form the bank number; however, writing 00 will select bank 01 instead</td>
</tr>
<tr>
<td>4000-5FFF</td>
<td>Write</td>
<td>RAM Bank Number/RTC Register Select</td>
<td>Writing 00-07 selects a RAM bank (if any); writing 08-0C will map the RTC register into memory</td>
</tr>
<tr>
<td>6000-7FFF</td>
<td>Write</td>
<td>Latch Clock Data</td>
<td>Writing 00 and then 01 will latch the current time into the RTC registers; latched data will not change until 00→01 is written again</td>
</tr>
</tbody>
</table>

Table 3: MBC3 memory map and register description [6]

MBC5

MBC5 supports games with up to 8MB ROM and 128kB RAM. Due to this, its ROM bank number requires 9 bits so two of the four control registers collectively address this. Unlike MBC1, it has separate registers to control RAM and ROM banking. Table 4 and Figure 20 outlines the memory map and register functions of the MBC5.
Figure 19: MBC3 Memory Map [6]
<table>
<thead>
<tr>
<th>Address</th>
<th>Read/Write</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-3FFF</td>
<td>Read</td>
<td>ROM Bank 00</td>
<td>Always contains the 16kB of ROM bank 00</td>
</tr>
<tr>
<td>4000-7FFF</td>
<td>Read</td>
<td>ROM Bank 00-7F</td>
<td>Contains any of the other 16kB banks of ROM, including bank 00</td>
</tr>
<tr>
<td>A000-BFFF</td>
<td>Read/Write</td>
<td>RAM Bank 00-03 (if any)</td>
<td>Addresses an external 8kB RAM bank</td>
</tr>
<tr>
<td>0000-1FFF</td>
<td>Write</td>
<td>RAM Enable</td>
<td>Writing 0Ah enables RAM; 00h disables RAM</td>
</tr>
<tr>
<td>2000-2FFF</td>
<td>Write</td>
<td>Low 8 bits of ROM Bank Number</td>
<td>Writes the lower 8 bits of the bank number; writing 00 is allowed</td>
</tr>
<tr>
<td>3000-3FFF</td>
<td>Write</td>
<td>MSB of ROM Bank Number</td>
<td>Writes the 9th bit of the bank number</td>
</tr>
<tr>
<td>4000-5FFF</td>
<td>Write</td>
<td>RAM Bank Number</td>
<td>Writing a value between 00-0F selects the corresponding RAM bank</td>
</tr>
</tbody>
</table>

Table 4: MBC5 memory map and register description [6]

Figure 20: MBC5 Memory Map [6]
Results

Accuracy Test ROMs

Gekkio [3] and Blargg [1] developed test ROMs from running them with real Game Boy devices. Our test results are listed in Appendix B. To the best of our knowledge, VerilogBoy [9] is the most recent attempt (apart from ours) to emulate the Game Boy with an FPGA. The other emulators were all written in software.

Game ROMs

The ultimate goal of the project is to successfully run games of various ROM and RAM sizes with either MBC1, MBC3, or MBC5 chips. Our Game Boy emulator has ran (but is not limited to) the following game ROMs in Table 5 without any noticeable problem:

<table>
<thead>
<tr>
<th>Name</th>
<th>ROM</th>
<th>RAM</th>
<th>MBC</th>
<th>External Battery</th>
</tr>
</thead>
<tbody>
<tr>
<td>OH DEMO</td>
<td>128kB</td>
<td>0</td>
<td>MBC1</td>
<td>No</td>
</tr>
<tr>
<td>POCKET-DEMO</td>
<td>128kB</td>
<td>0</td>
<td>MBC1</td>
<td>No</td>
</tr>
<tr>
<td>POKEMON YELLOW (INT)</td>
<td>1MB</td>
<td>32kB</td>
<td>MBC5</td>
<td>Yes</td>
</tr>
<tr>
<td>TETRIS</td>
<td>32kB</td>
<td>0</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>POKEMON RED (JP)</td>
<td>1MB</td>
<td>32kB</td>
<td>MBC3</td>
<td>Yes</td>
</tr>
<tr>
<td>DMG AGING TEST</td>
<td>32kB</td>
<td>0</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>LEGEND OF ZELDA: LINK’S AWAKENING</td>
<td>1MB</td>
<td>32kB</td>
<td>MBC5</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5: List of Game Boy games emulated
Evaluation

Contribution

Table 6 lists the modules each member contributed to.

<table>
<thead>
<tr>
<th>Member</th>
<th>Contributed Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nanyu</td>
<td>GB-Z80, MMU, Sound, Timer, PPU, OAM, Cartridge, system integration</td>
</tr>
<tr>
<td>Justin</td>
<td>Joypad, Serial, Cartridge, MBC, all software (user space program, joypad device driver)</td>
</tr>
</tbody>
</table>

Table 6: Project contribution

Future Work

The serial module was planned to be tested by exporting the SIN, SOUT, and SCK ports to GPIOs on the DE1-SoC board, and physically connecting the GPIO pins of both devices with jumper wires. However, this experiment was not carried out, mainly due to not having an extra DE1-SoC board to pair with.

We will improve the accuracy of the Game Boy and pass all the test ROMs. Once that is achieved, we will upgrade it to a Game Boy Color.
References


Appendices
Appendix A

Qsys System
Appendix B

Accuracy Tests
## Blargg's tests

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu instrs</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>dmg sound</td>
<td>✗</td>
<td>✔️</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✗</td>
</tr>
<tr>
<td>instr timing</td>
<td>✔️</td>
<td>✔️</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✔️</td>
</tr>
<tr>
<td>interrupt time</td>
<td>N/A</td>
<td>✗</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✗</td>
</tr>
<tr>
<td>mem timing</td>
<td>N/A</td>
<td>✔️</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✔️</td>
</tr>
<tr>
<td>mem timing 2</td>
<td>✔️</td>
<td>✔️</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✔️</td>
</tr>
<tr>
<td>oam bug</td>
<td>✗</td>
<td>✗</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✗</td>
</tr>
</tbody>
</table>
## Mooneye GB acceptance tests

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>add sp e timing</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>call timing</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>call timing2</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>call cc_timing</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>call cc_timing2</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>di timing GS</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>div timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ei sequence</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ei timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>halt ime0 ei</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>halt ime0 nointr_timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>halt ime1 timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>halt ime1 timing2 GS</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>if ie registers</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>intr timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>jp timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>jp cc timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ld hl sp e timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>oam dma_restart</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>oam dma start</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>oam dma timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>pop timing</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>push timing</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>rapid di ei</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ret timing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Instructions

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret cc timing</td>
<td>🔹</td>
<td>✗</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
</tr>
<tr>
<td>reti timing</td>
<td>🔹</td>
<td>✗</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
</tr>
<tr>
<td>reti intr timing</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
</tr>
<tr>
<td>rst timing</td>
<td>🔹</td>
<td>✗</td>
<td>✗</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
</tr>
</tbody>
</table>

### Interrupt handling

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ie push</td>
<td>🔹</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>🔹</td>
<td>🔹</td>
</tr>
</tbody>
</table>

### OAM DMA

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
</tr>
<tr>
<td>reg_read</td>
<td>🔹</td>
<td>🔹</td>
<td>🔹</td>
<td>✗</td>
<td>✗</td>
<td>🔹</td>
<td>🔹</td>
</tr>
<tr>
<td>sources dmgABCmgbS</td>
<td>🔹</td>
<td>🔹</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>🔹</td>
</tr>
</tbody>
</table>

### Serial

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot sclk align dmgABCmgb</td>
<td>✗</td>
<td>🔹</td>
<td>🔹</td>
<td>✗</td>
<td>✗</td>
<td>🔹</td>
<td>🔹</td>
</tr>
<tr>
<td>Test</td>
<td>mooneye-gb</td>
<td>BGB</td>
<td>Gambatte</td>
<td>Higan</td>
<td>MESS</td>
<td>VerilogBoy</td>
<td>Ours</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------</td>
<td>-----</td>
<td>----------</td>
<td>-------</td>
<td>-------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>hblank ly scx timing GS</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>intr 1 2 timing GS</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>intr 2 0 timing</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>intr 2 mode0 timing</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>intr 2 mode3 timing</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>intr 2 oam ok timing</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>intr 2 mode0 timing sprites</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>lcdon timing dmgABCmgbS</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>lcdon write timing GS</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>stat irq blocking</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>stat lyc onoff</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>vblank stat intr GS</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
## Timer

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>div write</td>
<td>👍</td>
<td>👍</td>
<td>X</td>
<td>👍</td>
<td>👍</td>
<td>👍</td>
<td>👍</td>
</tr>
<tr>
<td>rapid toggle</td>
<td>👍</td>
<td>👍</td>
<td>X</td>
<td>X</td>
<td>👍</td>
<td>👍</td>
<td>👍</td>
</tr>
<tr>
<td>tim00 div trigger</td>
<td>👍</td>
<td>👍</td>
<td>X</td>
<td>X</td>
<td>👍</td>
<td>👍</td>
<td>👍</td>
</tr>
<tr>
<td>tim00</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tim01 div trigger</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tim01</td>
<td>👍</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tim10 div trigger</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tim10</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tim11 div trigger</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tim11</td>
<td>👍</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tima reload</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tima write reloading</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>tma write reloading</td>
<td>👍</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

## MBC

<table>
<thead>
<tr>
<th>Test</th>
<th>mooneye-gb</th>
<th>BGB</th>
<th>Gambatte</th>
<th>Higan</th>
<th>MESS</th>
<th>VerilogBoy</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBC1</td>
<td>N/A</td>
<td>✔</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✔</td>
</tr>
<tr>
<td>MBC5</td>
<td>N/A</td>
<td>✔</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>✔</td>
</tr>
</tbody>
</table>

Note: MBC3 test ROM was not created at the time of testing.
Appendix C

Source Code

C.1 Hardware

```verbatim
/*timescale 1ns / 1ns*/

//
// This is the GB-Z80 CPU
// All modules in a single file for simulation
//
#include "GB_Z80_ALU.vh"
#include "GB_Z80_CPU.vh"
#include "GB_Z80_DECODER.vh"

#define NO_BOOT 0

module GB_Z80_SINGLE

```
input logic clk,
input logic rst,
output logic [15:0] ADDR, // Memory Address Bus
input logic [7:0] DATA_in, // Input Data Bus
output logic [7:0] DATA_out, // Output Data Bus
output logic RD, // CPU wants to read data from Memory or IO, active high
output logic WR, // CPU holds valid data to be stored in Memory or IO, active high
output logic CPU_HALT, // CPU has executed a HALT instruction and is awaiting an interrupt, active high
input logic [4:0] INTQ, // Interrupt Request, Interrupt will be honored at the end of the current instruction
input logic [4:0] IE // Interrupt Enable
);

GB_Z80_REG CPU_REG, CPU_REG_NEXT;
logic [15:0] ADDR_NEXT;

/* Decoder */
logic [7:0] INST, INST_NEXT; // Instruction Register
logic [4:0] INTQ_INT, INTQ_INT_NEXT;
GB_Z80_RISC_OPCODE RISC_OPCODE [0:10];
logic [5:0] NUM_Tcnt;
logic isCB, isCB_NEXT;
logic isINT, isINT_NEXT;
logic isPCMEM [0:10];
logic [4:0] T_CNT, T_CNT_NEXT;
logic [2:0] M_CNT, M_CNT_NEXT;
byte cur_risc_num;

GB_Z80_DECODER CPU_DECODER(.CPU_OPCODE(INST), .INTQ(INTQ_INT), .isCB(isCB)
    , .isINT(isINT), .RISC_OPCODE(RISC_OPCODE), .NUM_Tcnt(NUM_Tcnt), .
isPCMEM(isPCMEM),
    .FLAG(CPU_REG.F));

/* ALU */
logic [7:0] ALU_OPD1_L, ALU_OPD2_L, ALU_STATUS, ALU_RESULT_L, ALU_RESULT_H;

GB_Z80_ALU_OPCODE ALU_OPCODE;

GB_Z80_ALU_CPU_ALU(.OPD1_L(ALU_OPD1_L), .OPD2_L(ALU_OPD2_L), .OPCODE(
    ALU_OPCODE), .FLAG(CPU_REG.F), .STATUS(ALU_STATUS),
    .RESULT_L(ALU_RESULT_L), .RESULT_H(ALU_RESULT_H));

/* Main FSMD */
// Main 4 Stages are IF -> DE -> EX -> (MEM)WB
// Each takes 1 T cycle

typedef enum {CPU_IF, CPU_DE, CPU_DE_CB, CPU_EX_RISC, CPU_WB_RISC}
    CPU_STATE_t;

CPU_STATE_t CPU_STATE, CPU_STATE_NEXT;

logic RD_NEXT, WR_NEXT;
logic EX_done;

logic IME, IME_NEXT; // Interrupt Master Enable

always_ff @(posedge clk)
begin
    /* Power On Reset */
    if (rst)
        begin
            CPU_STATE <= CPU_IF;
            CPU_REG.PC <= 0;
            CPU_REG.F <= 0;
        end
CPU_REG.T <= 0;
ADDR <= 0;
if ('NO_BOOT')
begin
    CPU_REG.A <= 8'h01;
    CPU_REG.F <= 8'hB0;
    CPU_REG.B <= 8'h00;
    CPU_REG.C <= 8'h13;
    CPU_REG.D <= 8'h00;
    CPU_REG.E <= 8'hD8;
    CPU_REG.H <= 8'h01;
    CPU_REG.L <= 8'h4D;
    CPU_REG.SP<0> <= 8'hFF;
    CPU_REG.SP<1> <= 8'hFE;
    CPU_REG.PC <= 16'h0100;
    ADDR <= 16'h0100;
end
RD <= 1; WR <= 0;
T_CNT <= 0; //M_CNT <= 0;
isCB <= 0;
isINT <= 0;
IME <= 0;
INTQ_INT <= 0;
end
else
begin
    CPU_STATE <= CPU_STATE_NEXT;
    CPU_REG <= CPU_REG_NEXT;
    ADDR <= ADDR_NEXT;
    RD <= RD_NEXT; WR<= WR_NEXT;
    T_CNT <= T_CNT_NEXT; //M_CNT <= M_CNT_NEXT;
isCB <= isCB_NEXT;
isINT <= isINT_NEXT;
INST <= INST_NEXT;
IME <= IME_NEXT;
INTQ_INT <= INTQ_INT_NEXT;

end

assign M_CNT = (T_CNT - 1) >> 2; // 1 M Cycle for every 4 T cycles
assign cur_risc_num = (T_CNT >> 1) - 1 - (isCB << 1) + isINT;

always_comb
begin
    CPU_STATE_NEXT = CPU_STATE;
    CPU_REG_NEXT = CPU_REG;
    ADDR_NEXT = CPU_REG.PC;
    isCB_NEXT = isCB;
    isINT_NEXT = isINT;
    IME_NEXT = IME;
    INTQ_INT_NEXT = INTQ_INT;
    RD_NEXT = 0; WR_NEXT = 0;
    DATA_out = 0;
    T_CNT_NEXT = T_CNT + 1;
    INST_NEXT = INST;
    ALU_OPD1_L = 0;
    ALU_OPD2_L = 0;
    ALU_OPCODE = ALU_NOP;
    CPU_HALT = 0;
    unique case (CPU_STATE)
        // Instruction Fetch From Memory at PC
        CPU_IF : begin
            RD_NEXT = 0;
            INST_NEXT = DATA_in;
        end
    endcase
end
T_CNT_NEXT = T_CNT + 1;
CPU_STATE_NEXT = CPU_DE;
CPU_REG_NEXT.PC = CPU_REG.PC + 1;
end

CPU_DE :
begin
if ((INST) == 8'hCB && !isCB)
begin
CPU_STATE_NEXT = CPU_DE_CB;
isCB_NEXT = 1;
T_CNT_NEXT = T_CNT + 1;
end
else
begin
CPU_STATE_NEXT = CPU_EX_RISC;
T_CNT_NEXT = T_CNT + 1;
end
end

CPU_DE_CB :
begin
if (T_CNT != 3) // CB fetch delay
begin
T_CNT_NEXT = T_CNT + 1;
CPU_STATE_NEXT = CPU_DE_CB;
end
else
begin
T_CNT_NEXT = T_CNT + 1;
CPU_STATE_NEXT = CPU_IF;
RD_NEXT = 1;
end
end
CPU_EX_RISC:
begin
    T_CNT_NEXT = T_CNT + 1;
    CPU_STATE_NEXT = CPU_WB_RISC;

    if (isPCMEM[cur_risc_num])
        CPU_REG_NEXT.PC = CPU_REG.PC + 1;
    //if (!IME && (RISC_OPCODE[cur_risc_num] == HALT)) // HALT "skip" behaviour
        // CPU_REG_NEXT.PC = CPU_REG.PC + 1;

    case (RISC_OPCODE[cur_risc_num])
        NOP: ; // no operations
        LD_APC, LD_BPC, LD_CPC, LD_DPC, LD_EPC, LD_HPC, LD_LPC,
        LD_TPC, LD_XPC, LD_SP1PC, LD_SPhPC, JP_R8, JP_NZR8, JP_ZR8, JP_NCR8,
        JP_CR8: RD_NEXT = 1;
        LD_ABC: `RD_nn(B, C)
        LD_ADE: `RD_nn(D, E)
        LD_AHL, LD_BHL, LD_CHL, LD_DHL, LD_EHL, LD_HHL, LD_LHL,
        LD_THL, ADD_AHL, ADC_AHL, SUB_AHL, SBC_AHL, AND_AHL, XOR_AHL, OR_AHL,
        CP_AHL: `RD_nn(H, L)
        LD_ATX: `RD_nn(T, X)
        LD_PClSP, LD_PChSP, LD_BSP, LD_CSP, LD_DSP, LD_ESP, LD_HSP,
        , LD_LSP, LD_ASP, LD_FSP: `RD_nn(SPh, SP1)
        LD_AHT: `RD_FFn(T)
        LD_AHC: `RD_FFn(C)

        LD_PCSPl, LD_PCSPh: WR_NEXT = 1;
        LD_BCA: `WR_nn(B, C)
        LD_DEA: `WR_nn(D, E)
        LD_HLA, LD_HLB, LD_HLC, LD_HLD, LD_HLE, LD_HLH, LD_HLL,
        LD_HLT: `WR_nn(H, L)
        LD_TXA, LD_TXSPh, LD_TXSPl: `WR_nn(T, X)
LD_SPA, LD_SPB, LD_SPC, LD_SPD, LD_SPE, LD_SPH, LD_SPL,
LD_SPF, LD_SPPCh, LD_SPPCl : `WR_nn(SPh, SPL)
LD_HTA : `WR_FIFO(T)
LD_HCA : `WR_FIFO(C)
DI: IME_NEXT = 0;
LATCH_INTQ: INTQ_INT_NEXT = INTQ;
RST_IF: begin ADDR_NEXT = 16'hFF0F; WR_NEXT = 1; RD_NEXT = 1; end
default: endcase
end
CPU_WB_RISC :
begin
if (T_CNT - (isCB << 2) == NUM_Tcnt - 1)
begin
T_CNT_NEXT = 0;
CPU_STATE_NEXT = CPU_IF;

isCB_NEXT = 0;
RD_NEXT = 1;
isINT_NEXT = 0;
INTQ_INT_NEXT = 0;
if (IME && (INTQ != 5'b00))
begin
CPU_STATE_NEXT = CPU_EX_RISC;
isINT_NEXT = 1;
end
else if ((INTQ == 5'b00) && (RISC_OPCODE[cur_risc_num] == HALT)) // Handle HALT
begin
CPU_STATE_NEXT = CPU_WB_RISC;
T_CNT_NEXT = T_CNT;
CPU_HALT = 1;
else
begin

T_CNT_NEXT = T_CNT + 1;
CPU_STATE_NEXT = CPU_EX_RISC;

end

case (RISC_OPCODE[cur_risc_num])

  NOP: ;
  LD_AA: `LD_n_n(A, A)
  LD_AB: `LD_n_n(A, B)
  LD_AC: `LD_n_n(A, C)
  LD_AD: `LD_n_n(A, D)
  LD_AE: `LD_n_n(A, E)
  LD_AH: `LD_n_n(A, H)
  LD_AL: `LD_n_n(A, L)

  LD_BA: `LD_n_n(B, A)
  LD_BB: `LD_n_n(B, B)
  LD_BC: `LD_n_n(B, C)
  LD_BD: `LD_n_n(B, D)
  LD_BE: `LD_n_n(B, E)
  LD_BH: `LD_n_n(B, H)
  LD_BL: `LD_n_n(B, L)

  LD_CA: `LD_n_n(C, A)
  LD_CB: `LD_n_n(C, B)
  LD_CC: `LD_n_n(C, C)
  LD_CD: `LD_n_n(C, D)
  LD_CE: `LD_n_n(C, E)
  LD_CH: `LD_n_n(C, H)
  LD_CL: `LD_n_n(C, L)
LD_DA: `LD_n_n(D, A)
LD_DB: `LD_n_n(D, B)
LD_DC: `LD_n_n(D, C)
LD_DD: `LD_n_n(D, D)
LD_DE: `LD_n_n(D, E)
LD_DH: `LD_n_n(D, H)
LD_DL: `LD_n_n(D, L)

LD_EA: `LD_n_n(E, A)
LD_EB: `LD_n_n(E, B)
LD_EC: `LD_n_n(E, C)
LD_ED: `LD_n_n(E, D)
LD_EE: `LD_n_n(E, E)
LD_EH: `LD_n_n(E, H)
LD_EL: `LD_n_n(E, L)

LD_HA: `LD_n_n(H, A)
LD_HB: `LD_n_n(H, B)
LD_HC: `LD_n_n(H, C)
LD_HD: `LD_n_n(H, D)
LD_HE: `LD_n_n(H, E)
LD_HH: `LD_n_n(H, H)
LD_HL: `LD_n_n(H, L)

LD_LA: `LD_n_n(L, A)
LD_LB: `LD_n_n(L, B)
LD_LC: `LD_n_n(L, C)
LD_LD: `LD_n_n(L, D)
LD_LE: `LD_n_n(L, E)
LD_LH: `LD_n_n(L, H)
LD_LL: `LD_n_n(L, L)
LD_PCHL: CPU_REG_NEXT.PC = {CPU_REG.H, CPU_REG.L};

LD_SPHL: {CPU_REG_NEXT.SP_h, CPU_REG_NEXT.SP_l} = {CPU_REG.H, CPU_REG.L};

LD_APC, LD_AHL, LD_ABC, LD_ADE, LD_ASP, LD_AHT, LD_AHC, LD_ATX: CPU_REG_NEXT.A = DATA_in;
LD_BPC, LD_BHL, LD_BSP: CPU_REG_NEXT.B = DATA_in;
LD_CPC, LD_CHL, LD_CSP: CPU_REG_NEXT.C = DATA_in;
LD_DPC, LD_DHL, LD_DSP: CPU_REG_NEXT.D = DATA_in;
LD_EPC, LD_EHL, LD_ESP: CPU_REG_NEXT.E = DATA_in;
LD_HPC, LD_HHL, LD_HSP: CPU_REG_NEXT.H = DATA_in;
LD_LPC, LD_LHL, LD_LSP: CPU_REG_NEXT.L = DATA_in;
LD_FSP: CPU_REG_NEXT.F = DATA_in;
LD_TPC: CPU_REG_NEXT.T = DATA_in;
LD_XPC: CPU_REG_NEXT.X = DATA_in;
LD_SPI1PC: CPU_REG_NEXT.SP_l = DATA_in;
LD_SPhPC: CPU_REG_NEXT.SP_h = DATA_in;
LD_THLPC: CPU_REG_NEXT.T = DATA_in;
LD_PC1SP: CPU_REG_NEXT.PC = {CPU_REG.PC[15:8], DATA_in};
LD_PChSP: CPU_REG_NEXT.PC = {DATA_in, CPU_REG.PC[7:0]};

LD_BCA, LD_DEA, LD_HLA, LD_SPA, LD_HTA, LD_HCA, LD_TXA:
DATA_out = CPU_REG.A;
LD_HLB, LD_SPB: DATA_out = CPU_REG.B;
LD_HLC, LD_SPC: DATA_out = CPU_REG.C;
LD_HLD, LD_SPD: DATA_out = CPU_REG.D;
LD_HLE, LD_SPE: DATA_out = CPU_REG.E;
LD_HLH, LD_SPH: DATA_out = CPU_REG.H;
LD_HLL, LD_SPL: DATA_out = CPU_REG.L;
LD_SPF: DATA_out = CPU_REG.F;
LD_HLT: DATA_out = CPU_REG.T;
LD_PCSPl, LD_TXSPl: DATA_out = CPU_REG.SP1;
LD_PCSPh, LD_TXSPh: DATA_out = CPU_REG.SPPh;
LD_SPPCh: DATA_out = CPU_REG.PC[15:8];
LD_SPPC1: DATA_out = CPU_REG.PC[7:0];

LD_HL_SPR8: 'LD_HL_SPR8

INC_BC : `INC_nn(B, C)
DEC_BC : `DEC_nn(B, C)
INC_DE : `INC_nn(D, E)
DEC_DE : `DEC_nn(D, E)
INC_HL : `INC_nn(H, L)
DEC_HL : `DEC_nn(H, L)
INC_TX : `INC_nn(T, X)
DEC_TX : `DEC_nn(T, X)
INC_SP : `INC_nn(SP, SP1)
DEC_SP : `DEC_nn(SP, SP1)
INC_A : `INC_n(A)
DEC_A : `DEC_n(A)
INC_B : `INC_n(B)
DEC_B : `DEC_n(B)
INC_C : `INC_n(C)
DEC_C : `DEC_n(C)
INC_D : `INC_n(D)
DEC_D : `DEC_n(D)
INC_E : `INC_n(E)
DEC_E : `DEC_n(E)
INC_H : `INC_n(H)
DEC_H : `DEC_n(H)
INC_L : `INC_n(L)
DEC_L : `DEC_n(L)
INC_T : `INC_n(T)
DEC_T : 'DEC_n(T)

RLC_A : 'SHIFTER_op_n(RLC, A)
RLC_B : 'SHIFTER_op_n(RLC, B)
RLC_C : 'SHIFTER_op_n(RLC, C)
RLC_D : 'SHIFTER_op_n(RLC, D)
RLC_E : 'SHIFTER_op_n(RLC, E)
RLC_H : 'SHIFTER_op_n(RLC, H)
RLC_L : 'SHIFTER_op_n(RLC, L)
RLC_T : 'SHIFTER_op_n(RLC, T)

RRC_A : 'SHIFTER_op_n(RRC, A)
RRC_B : 'SHIFTER_op_n(RRC, B)
RRC_C : 'SHIFTER_op_n(RRC, C)
RRC_D : 'SHIFTER_op_n(RRC, D)
RRC_E : 'SHIFTER_op_n(RRC, E)
RRC_H : 'SHIFTER_op_n(RRC, H)
RRC_L : 'SHIFTER_op_n(RRC, L)
RRC_T : 'SHIFTER_op_n(RRC, T)

RR_A : 'SHIFTER_op_n(RR, A)
RR_B : 'SHIFTER_op_n(RR, B)
RR_C : 'SHIFTER_op_n(RR, C)
RR_D : 'SHIFTER_op_n(RR, D)
RR_E : 'SHIFTER_op_n(RR, E)
RR_H : 'SHIFTER_op_n(RR, H)
RR_L : 'SHIFTER_op_n(RR, L)
RR_T : 'SHIFTER_op_n(RR, T)

RL_A : 'SHIFTER_op_n(RL, A)
RL_B : 'SHIFTER_op_n(RL, B)
RL_C : 'SHIFTER_op_n(RL, C)
RL_D : 'SHIFTER_op_n(RL, D)
RL_E : `SHIFTER_op_n(RL, E)
RL_H : `SHIFTER_op_n(RL, H)
RL_L : `SHIFTER_op_n(RL, L)
RL_T : `SHIFTER_op_n(RL, T)

SRA_A : `SHIFTER_op_n(SRA, A)
SRA_B : `SHIFTER_op_n(SRA, B)
SRA_C : `SHIFTER_op_n(SRA, C)
SRA_D : `SHIFTER_op_n(SRA, D)
SRA_E : `SHIFTER_op_n(SRA, E)
SRA_H : `SHIFTER_op_n(SRA, H)
SRA_L : `SHIFTER_op_n(SRA, L)
SRA_T : `SHIFTER_op_n(SRA, T)

SLA_A : `SHIFTER_op_n(SLA, A)
SLA_B : `SHIFTER_op_n(SLA, B)
SLA_C : `SHIFTER_op_n(SLA, C)
SLA_D : `SHIFTER_op_n(SLA, D)
SLA_E : `SHIFTER_op_n(SLA, E)
SLA_H : `SHIFTER_op_n(SLA, H)
SLA_L : `SHIFTER_op_n(SLA, L)
SLA_T : `SHIFTER_op_n(SLA, T)

SWAP_A : `SHIFTER_op_n(SWAP, A)
SWAP_B : `SHIFTER_op_n(SWAP, B)
SWAP_C : `SHIFTER_op_n(SWAP, C)
SWAP_D : `SHIFTER_op_n(SWAP, D)
SWAP_E : `SHIFTER_op_n(SWAP, E)
SWAP_H : `SHIFTER_op_n(SWAP, H)
SWAP_L : `SHIFTER_op_n(SWAP, L)
SWAP_T : `SHIFTER_op_n(SWAP, T)

SRL_A : `SHIFTER_op_n(SRL, A)
SRL_B : `SHIFTER_op_n(SRL, B)
SRL_C : `SHIFTER_op_n(SRL, C)
SRL_D : `SHIFTER_op_n(SRL, D)
SRL_E : `SHIFTER_op_n(SRL, E)
SRL_H : `SHIFTER_op_n(SRL, H)
SRL_L : `SHIFTER_op_n(SRL, L)
SRL_T : `SHIFTER_op_n(SRL, T)

ADD_AA : `ALU_A_op_n(ADD, A)
ADD_AB : `ALU_A_op_n(ADD, B)
ADD_AC : `ALU_A_op_n(ADD, C)
ADD_AD : `ALU_A_op_n(ADD, D)
ADD_AE : `ALU_A_op_n(ADD, E)
ADD_AH : `ALU_A_op_n(ADD, H)
ADD_AL : `ALU_A_op_n(ADD, L)
ADD_AT : `ALU_A_op_n(ADD, T)
ADD_AHL : `ALU_A_op_Data_in(ADD)

ADD_SPT : `ADD_SPT

ADC_AA : `ALU_A_op_n(ADC, A)
ADC_AB : `ALU_A_op_n(ADC, B)
ADC_AC : `ALU_A_op_n(ADC, C)
ADC_AD : `ALU_A_op_n(ADC, D)
ADC_AE : `ALU_A_op_n(ADC, E)
ADC_AH : `ALU_A_op_n(ADC, H)
ADC_AL : `ALU_A_op_n(ADC, L)
ADC_AT : `ALU_A_op_n(ADC, T)
ADC_AHL : `ALU_A_op_Data_in(ADC)

SUB_AA : `ALU_A_op_n(SUB, A)
SUB_AB: `ALU_A_op_n(SUB, B)
SUB_AC: `ALU_A_op_n(SUB, C)
SUB_AD: `ALU_A_op_n(SUB, D)
SUB_AE: `ALU_A_op_n(SUB, E)
SUB_AH: `ALU_A_op_n(SUB, H)
SUB_AL: `ALU_A_op_n(SUB, L)
SUB_AT: `ALU_A_op_n(SUB, T)
SUB_AHL: `ALU_A_op_Data_in(SUB)

SBC_AA: `ALU_A_op_n(SBC, A)
SBC_AB: `ALU_A_op_n(SBC, B)
SBC_AC: `ALU_A_op_n(SBC, C)
SBC_AD: `ALU_A_op_n(SBC, D)
SBC_AE: `ALU_A_op_n(SBC, E)
SBC_AH: `ALU_A_op_n(SBC, H)
SBC_AL: `ALU_A_op_n(SBC, L)
SBC_AT: `ALU_A_op_n(SBC, T)
SBC_AHL: `ALU_A_op_Data_in(SBC)

AND_AA: `ALU_A_op_n(AND, A)
AND_AB: `ALU_A_op_n(AND, B)
AND_AC: `ALU_A_op_n(AND, C)
AND_AD: `ALU_A_op_n(AND, D)
AND_AE: `ALU_A_op_n(AND, E)
AND_AH: `ALU_A_op_n(AND, H)
AND_AL: `ALU_A_op_n(AND, L)
AND_AT: `ALU_A_op_n(AND, T)
AND_AHL: `ALU_A_op_Data_in(AND)

XOR_AA: `ALU_A_op_n(XOR, A)
XOR_AB: `ALU_A_op_n(XOR, B)
XOR_AC: `ALU_A_op_n(XOR, C)
XOR_AD: `ALU_A_op_n(XOR, D)
XOR_AE: `ALU_A_op_n(XOR, E)
XOR_AH: `ALU_A_op_n(XOR, H)
XOR_AL: `ALU_A_op_n(XOR, L)
XOR_AT: `ALU_A_op_n(XOR, T)
XOR_AHL: `ALU_A_op_Data_in(XOR)

OR_AA: `ALU_A_op_n(OR, A)
OR_AB: `ALU_A_op_n(OR, B)
OR_AC: `ALU_A_op_n(OR, C)
OR_AD: `ALU_A_op_n(OR, D)
OR_AE: `ALU_A_op_n(OR, E)
OR_AH: `ALU_A_op_n(OR, H)
OR_AL: `ALU_A_op_n(OR, L)
OR_AT: `ALU_A_op_n(OR, T)
OR_AHL: `ALU_A_op_Data_in(OR)

CP_AA: `ALU_op_n(CP, A)
CP_AB: `ALU_op_n(CP, B)
CP_AC: `ALU_op_n(CP, C)
CP_AD: `ALU_op_n(CP, D)
CP_AE: `ALU_op_n(CP, E)
CP_AH: `ALU_op_n(CP, H)
CP_AL: `ALU_op_n(CP, L)
CP_AT: `ALU_op_n(CP, T)
CP_AHL: `ALU_op_Data_in(CP)

ADD_LC: `ADDL_n(C)
ADD_LE: `ADDL_n(E)
ADD_LL: `ADDL_n(L)
ADD_LSP1: `ADDL_n(Sp1)
ADC_HB: `ADCH_n(B)
ADC_HD: `ADCH_n(D)
ADC_HH: `ADCH_n(H)
ADC_HSPh: `ADCH_n(SPh)

DAA: `DAA

CPL: begin
CPU_REG_NEXT.A = CPU_REG.A ^ 8'hFF;
CPU_REG_NEXT.F = CPU_REG.F | 8'b0110_0000;
end // invert all bits in A

SCF: CPU_REG_NEXT.F = {CPU_REG.F[7], 3'b001, CPU_REG.F[3:0]}; // set carry flag

CCF: CPU_REG_NEXT.F = {CPU_REG.F[7], 2'b00, ~CPU_REG.F[4],
CPU_REG.F[3:0]}; // compliment carry flag

JP_R8: CPU_REG_NEXT.PC = `DO_JPR8;

`DO_JPR8;

CPU_REG.PC;

`DO_JPR8;

CPU_REG.PC;

JP_TX: CPU_REG_NEXT.PC = {CPU_REG.T, CPU_REG.X};

CPU_REG.X}; CPU_REG.PC ;


CPU_REG.X}; CPU_REG.PC ;


RST_00: CPU_REG_NEXT.PC = {8'h00, 8'h00};
RST_08: CPU_REG_NEXT.PC = {8'h00, 8'h08};
RST_10: CPU_REG_NEXT.PC = {8'h00, 8'h10};
RST_18 : CPU_REG_NEXT.PC = {8'h00, 8'h18};
RST_20 : CPU_REG_NEXT.PC = {8'h00, 8'h20};
RST_28 : CPU_REG_NEXT.PC = {8'h00, 8'h28};
RST_30 : CPU_REG_NEXT.PC = {8'h00, 8'h30};
RST_38 : CPU_REG_NEXT.PC = {8'h00, 8'h38};
RST_40 : CPU_REG_NEXT.PC = {8'h00, 8'h40};
RST_48 : CPU_REG_NEXT.PC = {8'h00, 8'h48};
RST_50 : CPU_REG_NEXT.PC = {8'h00, 8'h50};
RST_58 : CPU_REG_NEXT.PC = {8'h00, 8'h58};
RST_60 : CPU_REG_NEXT.PC = {8'h00, 8'h60};

BIT0_A : `ALU_BIT_b_n(0, A)
BIT1_A : `ALU_BIT_b_n(1, A)
BIT2_A : `ALU_BIT_b_n(2, A)
BIT3_A : `ALU_BIT_b_n(3, A)
BIT4_A : `ALU_BIT_b_n(4, A)
BIT5_A : `ALU_BIT_b_n(5, A)
BIT6_A : `ALU_BIT_b_n(6, A)
BIT7_A : `ALU_BIT_b_n(7, A)

BIT0_B : `ALU_BIT_b_n(0, B)
BIT1_B : `ALU_BIT_b_n(1, B)
BIT2_B : `ALU_BIT_b_n(2, B)
BIT3_B : `ALU_BIT_b_n(3, B)
BIT4_B : `ALU_BIT_b_n(4, B)
BIT5_B : `ALU_BIT_b_n(5, B)
BIT6_B : `ALU_BIT_b_n(6, B)
BIT7_B : `ALU_BIT_b_n(7, B)

BIT0_C : `ALU_BIT_b_n(0, C)
BIT1_C : `ALU_BIT_b_n(1, C)
BIT2_C : `ALU_BIT_b_n(2, C)
BIT3_C : `ALU_BIT_b_n(3, C)
BIT4_C : `ALU_BIT_b_n(4, C)
BIT5_C : `ALU_BIT_b_n(5, C)
BIT6_C : `ALU_BIT_b_n(6, C)
BIT7_C : `ALU_BIT_b_n(7, C)

BIT0_D : `ALU_BIT_b_n(0, D)
BIT1_D : `ALU_BIT_b_n(1, D)
BIT2_D : `ALU_BIT_b_n(2, D)
BIT3_D : `ALU_BIT_b_n(3, D)
BIT4_D : `ALU_BIT_b_n(4, D)
BIT5_D : `ALU_BIT_b_n(5, D)
BIT6_D : `ALU_BIT_b_n(6, D)
BIT7_D : `ALU_BIT_b_n(7, D)

BIT0_E : `ALU_BIT_b_n(0, E)
BIT1_E : `ALU_BIT_b_n(1, E)
BIT2_E : `ALU_BIT_b_n(2, E)
BIT3_E : `ALU_BIT_b_n(3, E)
BIT4_E : `ALU_BIT_b_n(4, E)
BIT5_E : `ALU_BIT_b_n(5, E)
BIT6_E : `ALU_BIT_b_n(6, E)
BIT7_E : `ALU_BIT_b_n(7, E)

BIT0_H : `ALU_BIT_b_n(0, H)
BIT1_H : `ALU_BIT_b_n(1, H)
BIT2_H : `ALU_BIT_b_n(2, H)
BIT3_H : `ALU_BIT_b_n(3, H)
BIT4_H : `ALU_BIT_b_n(4, H)
BIT5_H : `ALU_BIT_b_n(5, H)
BIT6_H : `ALU_BIT_b_n(6, H)
BIT7_H : `ALU_BIT_b_n(7, H)

BIT0_L : `ALU_BIT_b_n(0, L)
BIT1_L: `ALU_BIT_b_n(1, L)
BIT2_L: `ALU_BIT_b_n(2, L)
BIT3_L: `ALU_BIT_b_n(3, L)
BIT4_L: `ALU_BIT_b_n(4, L)
BIT5_L: `ALU_BIT_b_n(5, L)
BIT6_L: `ALU_BIT_b_n(6, L)
BIT7_L: `ALU_BIT_b_n(7, L)

BIT0_T: `ALU_BIT_b_n(0, T)
BIT1_T: `ALU_BIT_b_n(1, T)
BIT2_T: `ALU_BIT_b_n(2, T)
BIT3_T: `ALU_BIT_b_n(3, T)
BIT4_T: `ALU_BIT_b_n(4, T)
BIT5_T: `ALU_BIT_b_n(5, T)
BIT6_T: `ALU_BIT_b_n(6, T)
BIT7_T: `ALU_BIT_b_n(7, T)

RES0_A: `ALU_SETRST_op_b_n(RES, 0, A)
RES1_A: `ALU_SETRST_op_b_n(RES, 1, A)
RES2_A: `ALU_SETRST_op_b_n(RES, 2, A)
RES3_A: `ALU_SETRST_op_b_n(RES, 3, A)
RES4_A: `ALU_SETRST_op_b_n(RES, 4, A)
RES5_A: `ALU_SETRST_op_b_n(RES, 5, A)
RES6_A: `ALU_SETRST_op_b_n(RES, 6, A)
RES7_A: `ALU_SETRST_op_b_n(RES, 7, A)

RES0_B: `ALU_SETRST_op_b_n(RES, 0, B)
RES1_B: `ALU_SETRST_op_b_n(RES, 1, B)
RES2_B: `ALU_SETRST_op_b_n(RES, 2, B)
RES3_B: `ALU_SETRST_op_b_n(RES, 3, B)
RES4_B: `ALU_SETRST_op_b_n(RES, 4, B)
RES5_B: `ALU_SETRST_op_b_n(RES, 5, B)
RES6_B: `ALU_SETRST_op_b_n(RES, 6, B)
RES7_B: `ALU_SETSTRST_op_b_n(RES, 7, B)
RES0_C: `ALU_SETSTRST_op_b_n(RES, 0, C)
RES1_C: `ALU_SETSTRST_op_b_n(RES, 1, C)
RES2_C: `ALU_SETSTRST_op_b_n(RES, 2, C)
RES3_C: `ALU_SETSTRST_op_b_n(RES, 3, C)
RES4_C: `ALU_SETSTRST_op_b_n(RES, 4, C)
RES5_C: `ALU_SETSTRST_op_b_n(RES, 5, C)
RES6_C: `ALU_SETSTRST_op_b_n(RES, 6, C)
RES7_C: `ALU_SETSTRST_op_b_n(RES, 7, C)
RES0_D: `ALU_SETSTRST_op_b_n(RES, 0, D)
RES1_D: `ALU_SETSTRST_op_b_n(RES, 1, D)
RES2_D: `ALU_SETSTRST_op_b_n(RES, 2, D)
RES3_D: `ALU_SETSTRST_op_b_n(RES, 3, D)
RES4_D: `ALU_SETSTRST_op_b_n(RES, 4, D)
RES5_D: `ALU_SETSTRST_op_b_n(RES, 5, D)
RES6_D: `ALU_SETSTRST_op_b_n(RES, 6, D)
RES7_D: `ALU_SETSTRST_op_b_n(RES, 7, D)
RES0_E: `ALU_SETSTRST_op_b_n(RES, 0, E)
RES1_E: `ALU_SETSTRST_op_b_n(RES, 1, E)
RES2_E: `ALU_SETSTRST_op_b_n(RES, 2, E)
RES3_E: `ALU_SETSTRST_op_b_n(RES, 3, E)
RES4_E: `ALU_SETSTRST_op_b_n(RES, 4, E)
RES5_E: `ALU_SETSTRST_op_b_n(RES, 5, E)
RES6_E: `ALU_SETSTRST_op_b_n(RES, 6, E)
RES7_E: `ALU_SETSTRST_op_b_n(RES, 7, E)
RES0_H: `ALU_SETSTRST_op_b_n(RES, 0, H)
RES1_H: `ALU_SETSTRST_op_b_n(RES, 1, H)
RES2_H: `ALU_SETSTRST_op_b_n(RES, 2, H)
RES3_H: `ALU_SETSTRST_op_b_n(RES, 3, H)
RES4_H: `ALU_SETST_op_b_n(RES, 4, H)
RES5_H: `ALU_SETST_op_b_n(RES, 5, H)
RES6_H: `ALU_SETST_op_b_n(RES, 6, H)
RES7_H: `ALU_SETST_op_b_n(RES, 7, H)

RES0_L: `ALU_SETST_op_b_n(RES, 0, L)
RES1_L: `ALU_SETST_op_b_n(RES, 1, L)
RES2_L: `ALU_SETST_op_b_n(RES, 2, L)
RES3_L: `ALU_SETST_op_b_n(RES, 3, L)
RES4_L: `ALU_SETST_op_b_n(RES, 4, L)
RES5_L: `ALU_SETST_op_b_n(RES, 5, L)
RES6_L: `ALU_SETST_op_b_n(RES, 6, L)
RES7_L: `ALU_SETST_op_b_n(RES, 7, L)

RES0_T: `ALU_SETST_op_b_n(RES, 0, T)
RES1_T: `ALU_SETST_op_b_n(RES, 1, T)
RES2_T: `ALU_SETST_op_b_n(RES, 2, T)
RES3_T: `ALU_SETST_op_b_n(RES, 3, T)
RES4_T: `ALU_SETST_op_b_n(RES, 4, T)
RES5_T: `ALU_SETST_op_b_n(RES, 5, T)
RES6_T: `ALU_SETST_op_b_n(RES, 6, T)
RES7_T: `ALU_SETST_op_b_n(RES, 7, T)

SET0_A: `ALU_SETST_op_b_n(SET, 0, A)
SET1_A: `ALU_SETST_op_b_n(SET, 1, A)
SET2_A: `ALU_SETST_op_b_n(SET, 2, A)
SET3_A: `ALU_SETST_op_b_n(SET, 3, A)
SET4_A: `ALU_SETST_op_b_n(SET, 4, A)
SET5_A: `ALU_SETST_op_b_n(SET, 5, A)
SET6_A: `ALU_SETST_op_b_n(SET, 6, A)
SET7_A: `ALU_SETST_op_b_n(SET, 7, A)

SET0_B: `ALU_SETST_op_b_n(SET, 0, B)
SET1_B: `ALU_SETRST_op_b_n(SET, 1, B)
SET2_B: `ALU_SETRST_op_b_n(SET, 2, B)
SET3_B: `ALU_SETRST_op_b_n(SET, 3, B)
SET4_B: `ALU_SETRST_op_b_n(SET, 4, B)
SET5_B: `ALU_SETRST_op_b_n(SET, 5, B)
SET6_B: `ALU_SETRST_op_b_n(SET, 6, B)
SET7_B: `ALU_SETRST_op_b_n(SET, 7, B)

SET0_C: `ALU_SETRST_op_b_n(SET, 0, C)
SET1_C: `ALU_SETRST_op_b_n(SET, 1, C)
SET2_C: `ALU_SETRST_op_b_n(SET, 2, C)
SET3_C: `ALU_SETRST_op_b_n(SET, 3, C)
SET4_C: `ALU_SETRST_op_b_n(SET, 4, C)
SET5_C: `ALU_SETRST_op_b_n(SET, 5, C)
SET6_C: `ALU_SETRST_op_b_n(SET, 6, C)
SET7_C: `ALU_SETRST_op_b_n(SET, 7, C)

SET0_D: `ALU_SETRST_op_b_n(SET, 0, D)
SET1_D: `ALU_SETRST_op_b_n(SET, 1, D)
SET2_D: `ALU_SETRST_op_b_n(SET, 2, D)
SET3_D: `ALU_SETRST_op_b_n(SET, 3, D)
SET4_D: `ALU_SETRST_op_b_n(SET, 4, D)
SET5_D: `ALU_SETRST_op_b_n(SET, 5, D)
SET6_D: `ALU_SETRST_op_b_n(SET, 6, D)
SET7_D: `ALU_SETRST_op_b_n(SET, 7, D)

SET0_E: `ALU_SETRST_op_b_n(SET, 0, E)
SET1_E: `ALU_SETRST_op_b_n(SET, 1, E)
SET2_E: `ALU_SETRST_op_b_n(SET, 2, E)
SET3_E: `ALU_SETRST_op_b_n(SET, 3, E)
SET4_E: `ALU_SETRST_op_b_n(SET, 4, E)
SET5_E: `ALU_SETRST_op_b_n(SET, 5, E)
SET6_E: `ALU_SETRST_op_b_n(SET, 6, E)
SET7_E: `ALU_SET_RST_op_b_n(SET, 7, E)

SET0_H: `ALU_SET_RST_op_b_n(SET, 0, H)
SET1_H: `ALU_SET_RST_op_b_n(SET, 1, H)
SET2_H: `ALU_SET_RST_op_b_n(SET, 2, H)
SET3_H: `ALU_SET_RST_op_b_n(SET, 3, H)
SET4_H: `ALU_SET_RST_op_b_n(SET, 4, H)
SET5_H: `ALU_SET_RST_op_b_n(SET, 5, H)
SET6_H: `ALU_SET_RST_op_b_n(SET, 6, H)
SET7_H: `ALU_SET_RST_op_b_n(SET, 7, H)

SET0_L: `ALU_SET_RST_op_b_n(SET, 0, L)
SET1_L: `ALU_SET_RST_op_b_n(SET, 1, L)
SET2_L: `ALU_SET_RST_op_b_n(SET, 2, L)
SET3_L: `ALU_SET_RST_op_b_n(SET, 3, L)
SET4_L: `ALU_SET_RST_op_b_n(SET, 4, L)
SET5_L: `ALU_SET_RST_op_b_n(SET, 5, L)
SET6_L: `ALU_SET_RST_op_b_n(SET, 6, L)
SET7_L: `ALU_SET_RST_op_b_n(SET, 7, L)

SET0_T: `ALU_SET_RST_op_b_n(SET, 0, T)
SET1_T: `ALU_SET_RST_op_b_n(SET, 1, T)
SET2_T: `ALU_SET_RST_op_b_n(SET, 2, T)
SET3_T: `ALU_SET_RST_op_b_n(SET, 3, T)
SET4_T: `ALU_SET_RST_op_b_n(SET, 4, T)
SET5_T: `ALU_SET_RST_op_b_n(SET, 5, T)
SET6_T: `ALU_SET_RST_op_b_n(SET, 6, T)
SET7_T: `ALU_SET_RST_op_b_n(SET, 7, T)

EI: IME_NEXT = 1;

RST_IF:
begin
DATA_out = DATA_in;
for (int i = 0; i < 5; i++)
begin
    if (INTQ_INT[i])
    begin
        DATA_out[i] = 0;
        break;
    end
end
default:
endcase

// Patch
if ((INST == 8'hC1 || INST == 8'hD1 || INST == 8'hE1 || INST == 8'hF1) && !isCB && !isINT && cur_risc_num == 4)
    `INC_nn(SPh, SP1)

if ((RISC_OPCODE[cur_risc_num] == RLC_A || RISC_OPCODE[cur_risc_num] == RL_A || RISC_OPCODE[cur_risc_num] == RRC_A || RISC_OPCODE[cur_risc_num] == RR_A) && !isCB && !isINT)
    begin
        CPU_REG_NEXT.F = ALU_STATUS & 8'b0001_1111;
    end

if ((INST == 8'h09 || INST == 8'h19 || INST == 8'h29 || INST == 8'h39) && !isCB && !isINT)
    begin
        CPU_REG_NEXT.F = (ALU_STATUS & 8'b0111_1111) | (CPU_REG.F & 8'b1000_0000); // Don't change Zero Flag
    end
CPU_REG_NEXT.F = CPU_REG_NEXT.F & 8'b1111_0000;

if (CPU_STATE_NEXT == CPU_IF) ADDR_NEXT = CPU_REG_NEXT.PC; // When PC is update at the last cycle, ADDR won't change in time, fix this

e ndcase

e nd

d endmodule

module GB_Z80_DECODER
(
  input logic [7:0] CPU_OPCODE,
  input logic [4:0] INTQ,
  input logic isCB,
  input logic isINT,
  input logic [7:0] FLAG,
  output GB_Z80_RISC_OPCODE RISC_OPCODE[0:10],
  output logic [5:0] NUM_Tcnt, // How many RISC opcodes in total (1-5)
  output logic isPCMEM [0:10]
);

always_comb
begin
  for (int i = 0; i <= 10; i++)
    begin
      RISC_OPCODE[i] = NOP;
isPCMEM[i] = 0;

end

NUM_Tcnt = 6'd4;

if (!isINT)
begin
unique case ( {isCB, CPU_OPCODE} )
  9'h000: RISC_OPCODE[0] = NOP;
  9'h001: `DECODER_LDnn_d16(B, C)
  9'h002: `DECODER_LDnn_A(BC)
  9'h003: `DECODER_INC_nn(BC)
  9'h004: RISC_OPCODE[0] = INC_B;
  9'h005: RISC_OPCODE[0] = DEC_B;
  9'h006: `DECODER_LDn_d8(B)
  9'h007: RISC_OPCODE[0] = RLC_A;
  9'h008: `DECODER_LD_a16_SP
  9'h009: `DECODER_ADDHL_nn(B, C)
  9'h00A: `DECODER_LDA_nn(BC)
  9'h00B: `DECODER_DEC_nn(BC)
  9'h00C: RISC_OPCODE[0] = INC_C;
  9'h00D: RISC_OPCODE[0] = DEC_C;
  9'h00E: `DECODER_LDn_d8(C)
  9'h00F: RISC_OPCODE[0] = RRC_A;
  9'h010: // STOP 0
        begin
            RISC_OPCODE[0] = NOP; // STOP not implemented yet
        end
  9'h011: `DECODER_LDnn_d16(D, E)
  9'h012: `DECODER_LDnn_A(DE)
  9'h013: `DECODER_INC_nn(DE)
  9'h014: RISC_OPCODE[0] = INC_D;
  9'h015: RISC_OPCODE[0] = DEC_D;
9'h016: `DECODER_LDn_d8(D)
9'h017: RISC_OPCODE[0] = RL_A;
9'h018: // JR r8
begin
    NUM_Tcnt = 6'd12;
end
9'h019: `DECODER_ADDHL_nn(D, E)
9'h01A: `DECODER_LDA_nn(DE)
9'h01B: `DECODER_DEC_nn(DE)
9'h01C: RISC_OPCODE[0] = INC_E;
9'h01D: RISC_OPCODE[0] = DEC_E;
9'h01E: `DECODER_LDn_d8(E)
9'h01F: RISC_OPCODE[0] = RR_A;
9'h020: // JR NZ,r8
begin
    RISC_OPCODE[2] = JP_NZR8;
end
9'h021: `DECODER_LDnn_d16(H, L)
9'h022: `DECODER_LD_HL_INC_A
9'h023: `DECODER_INC_nn(HL)
9'h024: RISC_OPCODE[0] = INC_H;
9'h025: RISC_OPCODE[0] = DEC_H;
9'h026: `DECODER_LDn_d8(H)
9'h027: RISC_OPCODE[0] = DAA;
9'h028: // JR Z,r8
begin
end
9'h029: `DECODER_ADDHL_nn(H, L)
9'h02A: `DECODER_LD_A_HL_INC
9'h02B: `DECODER_DEC_nn(HL)
9'h02C: RISC_OPCODE[0] = INC_L;
9'h02D: RISC_OPCODE[0] = DEC_L;
9'h02E: `DECODER_LDn_d8(L)
9'h02F: RISC_OPCODE[0] = CPL;
9'h030:

begin
end

9'h031: `DECODER_LDnn_d16(SPh, SP1)
9'h032: `DECODER_LD_HL_DEC_A
9'h033: `DECODER_INC_nn(SP)
9'h034: `DECODER_INC_MEM_HL
9'h035: `DECODER_DEC_MEM_HL
9'h036: `DECODER_LD_MEM_HL_d8
9'h037: RISC_OPCODE[0] = SCF;
9'h038:

begin
end

9'h039: `DECODER_ADDHL_nn(SPh, SP1)
9'h03A: `DECODER_LD_A_HL_DEC
9'h03B: `DECODER_DEC_nn(SP)
9'h03C: RISC_OPCODE[0] = INC_A;
9'h03D: RISC_OPCODE[0] = DEC_A;
9'h03E: `DECODER_LDn_d8(A)
9'h03F: RISC_OPCODE[0] = CCF;
9'h040: RISC_OPCODE[0] = LD_BB;
9'h041: RISC_OPCODE[0] = LD_BC;
9'h042: RISC_OPCODE[0] = LD_BD;
9'h043: RISC_OPCODE[0] = LD_BE;
9'h044: RISC_OPCODE[0] = LD_BH;
9'h045: RISC_OPCODE[0] = LD_BL;
9'h046: `DECODER_LD_n_MEM_HL(B)
9'h047: RISC_OPCODE[0] = LD_BA;
9'h048: RISC_OPCODE[0] = LD_CB;
9'h049: RISC_OPCODE[0] = LD_CC;
9'h04A: RISC_OPCODE[0] = LD_CD;
9'h04B: RISC_OPCODE[0] = LD_CE;
9'h04C: RISC_OPCODE[0] = LD_CH;
9'h04D: RISC_OPCODE[0] = LD_CL;
9'h04E: `DECODER_LD_n_MEM_HL(C)
9'h04F: RISC_OPCODE[0] = LD_CA;
9'h050: RISC_OPCODE[0] = LD_DB;
9'h051: RISC_OPCODE[0] = LD_DC;
9'h052: RISC_OPCODE[0] = LD_DD;
9'h053: RISC_OPCODE[0] = LD_DE;
9'h054: RISC_OPCODE[0] = LD_DH;
9'h055: RISC_OPCODE[0] = LD_DL;
9'h056: `DECODER_LD_n_MEM_HL(D)
9'h057: RISC_OPCODE[0] = LD_DA;
9'h058: RISC_OPCODE[0] = LD_EB;
9'h059: RISC_OPCODE[0] = LD_EC;
9'h05A: RISC_OPCODE[0] = LD_ED;
9'h05B: RISC_OPCODE[0] = LD_EE;
9'h05C: RISC_OPCODE[0] = LD_EH;
9'h05D: RISC_OPCODE[0] = LD_EL;
9'h05E: `DECODER_LD_n_MEM_HL(E)
9'h05F: RISC_OPCODE[0] = LD_EA;
9'h060: RISC_OPCODE[0] = LD_HB;
9'h061: RISC_OPCODE[0] = LD_HC;
9'h062: RISC_OPCODE[0] = LD_HD;
9'h063: RISC_OPCODE[0] = LD_HE;
9'h064: RISC_OPCODE[0] = LD_HH;
h065: RISC_OPCODE[0] = LD_HL;
h066: `DECODER_LD_n_MEM_HL(H)
h067: RISC_OPCODE[0] = LD_HA;
h068: RISC_OPCODE[0] = LD_LB;
h069: RISC_OPCODE[0] = LD_LC;
h06A: RISC_OPCODE[0] = LD_LD;
h06B: RISC_OPCODE[0] = LD_LE;
h06C: RISC_OPCODE[0] = LD_LH;
h06D: RISC_OPCODE[0] = LD_LL;
h06E: `DECODER_LD_n_MEM_HL(L)
h06F: RISC_OPCODE[0] = LD_LA;
h070: `DECODER_LD_MEM_HL_n(B)
h071: `DECODER_LD_MEM_HL_n(C)
h072: `DECODER_LD_MEM_HL_n(D)
h073: `DECODER_LD_MEM_HL_n(E)
h074: `DECODER_LD_MEM_HL_n(H)
h075: `DECODER_LD_MEM_HL_n(L)
h076: RISC_OPCODE[0] = HALT;
h077: `DECODER_LD_MEM_HL_n(A)
h078: RISC_OPCODE[0] = LD_AB;
h079: RISC_OPCODE[0] = LD_AC;
h07A: RISC_OPCODE[0] = LD_AD;
h07B: RISC_OPCODE[0] = LD_AE;
h07C: RISC_OPCODE[0] = LD_AH;
h07D: RISC_OPCODE[0] = LD_AL;
h07E: `DECODER_LD_n_MEM_HL(A)
h07F: RISC_OPCODE[0] = LD_AA;
h080: `DECODER_ALU_op_n(ADD, B)
h081: `DECODER_ALU_op_n(ADD, C)
h082: `DECODER_ALU_op_n(ADD, D)
h083: `DECODER_ALU_op_n(ADD, E)
h084: `DECODER_ALU_op_n(ADD, H)
h085: `DECODER_ALU_op_n(ADD, L)
9'h086: `DECODER_ALU_op_MEM_HL (ADD)
9'h087: `DECODER_ALU_op_n (ADD, A)
9'h088: `DECODER_ALU_op_n (ADC, B)
9'h089: `DECODER_ALU_op_n (ADC, C)
9'h08A: `DECODER_ALU_op_n (ADC, D)
9'h08B: `DECODER_ALU_op_n (ADC, E)
9'h08C: `DECODER_ALU_op_n (ADC, H)
9'h08D: `DECODER_ALU_op_n (ADC, L)
9'h08E: `DECODER_ALU_op_MEM_HL (ADC)
9'h08F: `DECODER_ALU_op_n (ADC, A)
9'h090: `DECODER_ALU_op_n (SUB, B)
9'h091: `DECODER_ALU_op_n (SUB, C)
9'h092: `DECODER_ALU_op_n (SUB, D)
9'h093: `DECODER_ALU_op_n (SUB, E)
9'h094: `DECODER_ALU_op_n (SUB, H)
9'h095: `DECODER_ALU_op_n (SUB, L)
9'h096: `DECODER_ALU_op_MEM_HL (SUB)
9'h097: `DECODER_ALU_op_n (SUB, A)
9'h098: `DECODER_ALU_op_n (SBC, B)
9'h099: `DECODER_ALU_op_n (SBC, C)
9'h09A: `DECODER_ALU_op_n (SBC, D)
9'h09B: `DECODER_ALU_op_n (SBC, E)
9'h09C: `DECODER_ALU_op_n (SBC, H)
9'h09D: `DECODER_ALU_op_n (SBC, L)
9'h09E: `DECODER_ALU_op_MEM_HL (SBC)
9'h09F: `DECODER_ALU_op_n (SBC, A)
9'h0A0: `DECODER_ALU_op_n (AND, B)
9'h0A1: `DECODER_ALU_op_n (AND, C)
9'h0A2: `DECODER_ALU_op_n (AND, D)
9'h0A3: `DECODER_ALU_op_n (AND, E)
9'h0A4: `DECODER_ALU_op_n (AND, H)
9'h0A5: `DECODER_ALU_op_n (AND, L)
9'h0A6: `DECODER_ALU_op_MEM_HL (AND)
9'hoA7: 'DECODER_ALU_op_n(AND, A)
9'hoA8: 'DECODER_ALU_op_n(XOR, B)
9'hoA9: 'DECODER_ALU_op_n(XOR, C)
9'hoaA: 'DECODER_ALU_op_n(XOR, D)
9'hoaB: 'DECODER_ALU_op_n(XOR, E)
9'hoaC: 'DECODER_ALU_op_n(XOR, H)
9'hoaD: 'DECODER_ALU_op_n(XOR, L)
9'hoaE: 'DECODER_ALU_op_MEM_HL(XOR)
9'hoaF: 'DECODER_ALU_op_n(XOR, A)
9'h0B0: 'DECODER_ALU_op_n(OR, B)
9'h0B1: 'DECODER_ALU_op_n(OR, C)
9'h0B2: 'DECODER_ALU_op_n(OR, D)
9'h0B3: 'DECODER_ALU_op_n(OR, E)
9'h0B4: 'DECODER_ALU_op_n(OR, H)
9'h0B5: 'DECODER_ALU_op_n(OR, L)
9'h0B6: 'DECODER_ALU_op_MEM_HL(OR)
9'h0B7: 'DECODER_ALU_op_n(OR, A)
9'h0B8: 'DECODER_ALU_op_n(CP, B)
9'h0B9: 'DECODER_ALU_op_n(CP, C)
9'h0BA: 'DECODER_ALU_op_n(CP, D)
9'h0BB: 'DECODER_ALU_op_n(CP, E)
9'h0BC: 'DECODER_ALU_op_n(CP, H)
9'h0BD: 'DECODER_ALU_op_n(CP, L)
9'h0BE: 'DECODER_ALU_op_MEM_HL(CP)
9'h0BF: 'DECODER_ALU_op_n(CP, A)
9'h0C0: 'DECODER_RET_NZ
9'h0C1: 'DECODER_POP_nn(B, C)
9'h0C2: 'DECODER_JP_NZ_a16
9'h0C3: 'DECODER_JP_a16
9'h0C4: 'DECODER_CALL_NZ_a16
9'h0C5: 'DECODER_PUSH_nn(B, C)
9'h0C6: 'DECODER_ALU_op_d8(ADD)
9'h0C7: 'DECODER_RST(00)
9'h0C8: `DECODER_RET_Z
9'h0C9: `DECODER_RET
9'h0CA: `DECODER_JP_Z_a16
9'h0CB: ; // CB Prefix
9'h0CC: `DECODER_CALL_Z_a16
9'h0CD: `DECODER_CALL_a16
9'h0CE: `DECODER_ALU_op_d8(ADC)
9'h0CF: `DECODER_RST(08)
9'h0D0: `DECODER_RET_NC
9'h0D1: `DECODER_POP_nn(D, E)
9'h0D2: `DECODER_JP_NC_a16
9'h0D3: ; // Undefined
9'h0D4: `DECODER_CALL_NC_a16
9'h0D5: `DECODER_PUSH_nn(D, E)
9'h0D6: `DECODER_ALU_op_d8(SUB)
9'h0D7: `DECODER_RST(10)
9'h0D8: `DECODER_RET_C
9'h0D9: `DECODER_RETI
9'h0DA: `DECODER_JP_C_a16
9'h0DB: ; // Undefined
9'h0DC: `DECODER_CALL_C_a16
9'h0DD: ; // Undefined
9'h0DE: `DECODER_ALU_op_d8(SBC)
9'h0DF: `DECODER_RST(18)
9'h0E0: `DECODER_LDH_a8_A
9'h0E1: `DECODER_POP_nn(H, L)
9'h0E2: `DECODER_LDH_C_A
9'h0E3: ; // Undefined
9'h0E4: ; // Undefined
9'h0E5: `DECODER_PUSH_nn(H, L)
9'h0E6: `DECODER_ALU_op_d8(AND)
9'h0E7: `DECODER_RST(20)
9'h0E8: `DECODER_ADD_SP_R8
9'h0E9: RISC_OPCODE[0] = LD_PCHL;
9'h0EA: `DECODER_LD_a16_A
9'h0EB: ; // Undefined
9'h0EC: ; // Undefined
9'h0ED: ; // Undefined
9'h0EE: `DECODER_ALU_op_d8(XOR)
9'h0EF: `DECODER_RST(28)
9'h0F0: `DECODER_LDH_A_a8
9'h0F1: `DECODER_POP_nn(A, F)
9'h0F2: `DECODER_LDH_A_C
9'h0F3: RISC_OPCODE[0] = DI;
9'h0F4: ; // Undefined
9'h0F5: `DECODER_PUSH_nn(A, F)
9'h0F6: `DECODER_ALU_op_d8(OR)
9'h0F7: `DECODER_RST(30)
9'h0F8: `DECODER_LD_HL_SPR8
9'h0F9: begin RISC_OPCODE[2] = LD_SPHL; NUM_Tcnt = 6'd8; end
9'h0FA: `DECODER_LD_A_a16
9'h0FB: RISC_OPCODE[0] = EI;
9'h0FC: ; // Undefined
9'h0FD: ; // Undefined
9'h0FE: `DECODER_ALU_op_d8(CP)
9'h0FF: `DECODER_RST(38)

/* CB Commands */
9'h100: RISC_OPCODE[0] = RLC_B;
9'h101: RISC_OPCODE[0] = RLC_C;
9'h102: RISC_OPCODE[0] = RLC_D;
9'h103: RISC_OPCODE[0] = RLC_E;
9'h104: RISC_OPCODE[0] = RLC_H;
9'h105: RISC_OPCODE[0] = RLC_L;
9'h106: `DECODER_CB_ALU_op_MEM_HL(RLC)
9'h107: RISC_OPCODE[0] = RLC_A;
9'h108: RISC_OPCODE[0] = RRC_B;
9'h109: RISC_OPCODE[0] = RRC_C;
9'h10A: RISC_OPCODE[0] = RRC_D;
9'h10B: RISC_OPCODE[0] = RRC_E;
9'h10C: RISC_OPCODE[0] = RRC_H;
9'h10D: RISC_OPCODE[0] = RRC_L;
9'h10E: `DECODER_CB_ALU_op_MEM_HL(RRC)
9'h10F: RISC_OPCODE[0] = RRC_A;
9'h110: RISC_OPCODE[0] = RL_B;
9'h111: RISC_OPCODE[0] = RL_C;
9'h112: RISC_OPCODE[0] = RL_D;
9'h113: RISC_OPCODE[0] = RL_E;
9'h114: RISC_OPCODE[0] = RL_H;
9'h115: RISC_OPCODE[0] = RL_L;
9'h116: `DECODER_CB_ALU_op_MEM_HL(RL)
9'h117: RISC_OPCODE[0] = RL_A;
9'h118: RISC_OPCODE[0] = RR_B;
9'h119: RISC_OPCODE[0] = RR_C;
9'h11A: RISC_OPCODE[0] = RR_D;
9'h11B: RISC_OPCODE[0] = RR_E;
9'h11C: RISC_OPCODE[0] = RR_H;
9'h11D: RISC_OPCODE[0] = RR_L;
9'h11E: `DECODER_CB_ALU_op_MEM_HL(RR)
9'h11F: RISC_OPCODE[0] = RR_A;
9'h120: RISC_OPCODE[0] = SLA_B;
9'h121: RISC_OPCODE[0] = SLA_C;
9'h122: RISC_OPCODE[0] = SLA_D;
9'h123: RISC_OPCODE[0] = SLA_E;
9'h124: RISC_OPCODE[0] = SLA_H;
9'h125: RISC_OPCODE[0] = SLA_L;
9'h126: `DECODER_CB_ALU_op_MEM_HL(SLA)
9'h127: RISC_OPCODE[0] = SLA_A;
9'h128: RISC_OPCODE[0] = SRA_B;
9'h129: RISC_OPCODE[0] = SRA_C;
9'h12A: RISC_OPCODE[0] = SRA_D;
9'h12B: RISC_OPCODE[0] = SRA_E;
9'h12C: RISC_OPCODE[0] = SRA_H;
9'h12D: RISC_OPCODE[0] = SRA_L;
9'h12E: `DECODER_CB_ALU_op_MEM_HL(SRA)
9'h12F: RISC_OPCODE[0] = SRA_A;
9'h130: RISC_OPCODE[0] = SWAP_B;
9'h131: RISC_OPCODE[0] = SWAP_C;
9'h132: RISC_OPCODE[0] = SWAP_D;
9'h133: RISC_OPCODE[0] = SWAP_E;
9'h134: RISC_OPCODE[0] = SWAP_H;
9'h135: RISC_OPCODE[0] = SWAP_L;
9'h136: `DECODER_CB_ALU_op_MEM_HL(SWAP)
9'h137: RISC_OPCODE[0] = SWAP_A;
9'h138: RISC_OPCODE[0] = SRL_B;
9'h139: RISC_OPCODE[0] = SRL_C;
9'h13A: RISC_OPCODE[0] = SRL_D;
9'h13B: RISC_OPCODE[0] = SRL_E;
9'h13C: RISC_OPCODE[0] = SRL_H;
9'h13D: RISC_OPCODE[0] = SRL_L;
9'h13E: `DECODER_CB_ALU_op_MEM_HL(SRL)
9'h13F: RISC_OPCODE[0] = SRL_A;
9'h140: `DECODER_CB_BIT_op_b_n(BIT, 0, B)
9'h141: `DECODER_CB_BIT_op_b_n(BIT, 0, C)
9'h142: `DECODER_CB_BIT_op_b_n(BIT, 0, D)
9'h143: `DECODER_CB_BIT_op_b_n(BIT, 0, E)
9'h144: `DECODER_CB_BIT_op_b_n(BIT, 0, H)
9'h145: `DECODER_CB_BIT_op_b_n(BIT, 0, L)
9'h146: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 0)
9'h147: `DECODER_CB_BIT_op_b_n(BIT, 0, A)
9'h148: `DECODER_CB_BIT_op_b_n(BIT, 1, B)
9'h149: `DECODER_CB_BIT_op_b_n(BIT, 1, C)
9'h14A: `DECODER_CB_BIT_op_b_n(BIT, 1, D)
9'h14B: `DECODER_CB_BIT_op_b_n(BIT, 1, E)
9'h14C: `DECODER_CB_BIT_op_b_n(BIT, 1, H)
9'h14D: `DECODER_CB_BIT_op_b_n(BIT, 1, L)
9'h14E: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 1)
9'h14F: `DECODER_CB_BIT_op_b_n(BIT, 1, A)
9'h150: `DECODER_CB_BIT_op_b_n(BIT, 2, B)
9'h151: `DECODER_CB_BIT_op_b_n(BIT, 2, C)
9'h152: `DECODER_CB_BIT_op_b_n(BIT, 2, D)
9'h153: `DECODER_CB_BIT_op_b_n(BIT, 2, E)
9'h154: `DECODER_CB_BIT_op_b_n(BIT, 2, H)
9'h155: `DECODER_CB_BIT_op_b_n(BIT, 2, L)
9'h156: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 2)
9'h157: `DECODER_CB_BIT_op_b_n(BIT, 2, A)
9'h158: `DECODER_CB_BIT_op_b_n(BIT, 3, B)
9'h159: `DECODER_CB_BIT_op_b_n(BIT, 3, C)
9'h15A: `DECODER_CB_BIT_op_b_n(BIT, 3, D)
9'h15B: `DECODER_CB_BIT_op_b_n(BIT, 3, E)
9'h15C: `DECODER_CB_BIT_op_b_n(BIT, 3, H)
9'h15D: `DECODER_CB_BIT_op_b_n(BIT, 3, L)
9'h15E: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 3)
9'h15F: `DECODER_CB_BIT_op_b_n(BIT, 3, A)
9'h160: `DECODER_CB_BIT_op_b_n(BIT, 4, B)
9'h161: `DECODER_CB_BIT_op_b_n(BIT, 4, C)
9'h162: `DECODER_CB_BIT_op_b_n(BIT, 4, D)
9'h163: `DECODER_CB_BIT_op_b_n(BIT, 4, E)
9'h164: `DECODER_CB_BIT_op_b_n(BIT, 4, H)
9'h165: `DECODER_CB_BIT_op_b_n(BIT, 4, L)
9'h166: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 4)
9'h167: `DECODER_CB_BIT_op_b_n(BIT, 4, A)
9'h168: `DECODER_CB_BIT_op_b_n(BIT, 5, B)
9'h169: `DECODER_CB_BIT_op_b_n(BIT, 5, C)
9'h16A: `DECODER_CB_BIT_op_b_n(BIT, 5, D)
9'h16B: `DECODER_CB_BIT_op_b_n(BIT, 5, E)
9'h16C: `DECODER_CB_BIT_op_b_n(BIT, 5, H)
9'h16D: `DECODER_CB_BIT_op_b_n(BIT, 5, L)
9'h16E: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 5)
9'h16F: `DECODER_CB_BIT_op_b_n(BIT, 5, A)
9'h170: `DECODER_CB_BIT_op_b_n(BIT, 6, B)
9'h171: `DECODER_CB_BIT_op_b_n(BIT, 6, C)
9'h172: `DECODER_CB_BIT_op_b_n(BIT, 6, D)
9'h173: `DECODER_CB_BIT_op_b_n(BIT, 6, E)
9'h174: `DECODER_CB_BIT_op_b_n(BIT, 6, H)
9'h175: `DECODER_CB_BIT_op_b_n(BIT, 6, L)
9'h176: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 6)
9'h177: `DECODER_CB_BIT_op_b_n(BIT, 6, A)
9'h178: `DECODER_CB_BIT_op_b_n(BIT, 7, B)
9'h179: `DECODER_CB_BIT_op_b_n(BIT, 7, C)
9'h17A: `DECODER_CB_BIT_op_b_n(BIT, 7, D)
9'h17B: `DECODER_CB_BIT_op_b_n(BIT, 7, E)
9'h17C: `DECODER_CB_BIT_op_b_n(BIT, 7, H)
9'h17D: `DECODER_CB_BIT_op_b_n(BIT, 7, L)
9'h17E: `DECODER_CB_BIT_op_b_MEM_HL(BIT, 7)
9'h17F: `DECODER_CB_BIT_op_b_n(BIT, 7, A)
9'h180: `DECODER_CB_BIT_op_b_n(RES, 0, B)
9'h181: `DECODER_CB_BIT_op_b_n(RES, 0, C)
9'h182: `DECODER_CB_BIT_op_b_n(RES, 0, D)
9'h183: `DECODER_CB_BIT_op_b_n(RES, 0, E)
9'h184: `DECODER_CB_BIT_op_b_n(RES, 0, H)
9'h185: `DECODER_CB_BIT_op_b_n(RES, 0, L)
9'h186: `DECODER_CB_RES_SET_op_b_MEM_HL(RES, 0)
9'h187: `DECODER_CB_BIT_op_b_n(RES, 0, A)
9'h188: `DECODER_CB_BIT_op_b_n(RES, 1, B)
9'h189: `DECODER_CB_BIT_op_b_n(RES, 1, C)
9'h18A: `DECODER_CB_BIT_op_b_n(RES, 1, D)
9'h18B: `DECODER_CB_BIT_op_b_n(RES, 1, E)
9'h18C: `DECODER_CB_BIT_op_b_n(RES, 1, H)
9'h18D: `DECODER_CB_BIT_op_b_n(RES, 1, L)
9'h18E: `DECODER_CB_RES_SET_op_b_MEM_HL(RES, 1)
9'h18F: `DECODER_CB_BIT_op_b_n(RES, 1, A)
9'h190: `DECODER_CB_BIT_op_b_n(RES, 2, B)
9'h191: `DECODER_CB_BIT_op_b_n(RES, 2, C)
9'h192: `DECODER_CB_BIT_op_b_n(RES, 2, D)
9'h193: `DECODER_CB_BIT_op_b_n(RES, 2, E)
9'h194: `DECODER_CB_BIT_op_b_n(RES, 2, H)
9'h195: `DECODER_CB_BIT_op_b_n(RES, 2, L)
9'h196: `DECODER_CB_RES_SET_op_b_MEM_HL(RES, 2)
9'h197: `DECODER_CB_BIT_op_b_n(RES, 2, A)
9'h198: `DECODER_CB_BIT_op_b_n(RES, 3, B)
9'h199: `DECODER_CB_BIT_op_b_n(RES, 3, C)
9'h19A: `DECODER_CB_BIT_op_b_n(RES, 3, D)
9'h19B: `DECODER_CB_BIT_op_b_n(RES, 3, E)
9'h19C: `DECODER_CB_BIT_op_b_n(RES, 3, H)
9'h19D: `DECODER_CB_BIT_op_b_n(RES, 3, L)
9'h19E: `DECODER_CB_RES_SET_op_b_MEM_HL(RES, 3)
9'h19F: `DECODER_CB_BIT_op_b_n(RES, 3, A)
9'h1A0: `DECODER_CB_BIT_op_b_n(RES, 4, B)
9'h1A1: `DECODER_CB_BIT_op_b_n(RES, 4, C)
9'h1A2: `DECODER_CB_BIT_op_b_n(RES, 4, D)
9'h1A3: `DECODER_CB_BIT_op_b_n(RES, 4, E)
9'h1A4: `DECODER_CB_BIT_op_b_n(RES, 4, H)
9'h1A5: `DECODER_CB_BIT_op_b_n(RES, 4, L)
9'h1A6: `DECODER_CB_RES_SET_op_b_MEM_HL(RES, 4)
9'h1A7: `DECODER_CB_BIT_op_b_n(RES, 4, A)
9'h1A8: `DECODER_CB_BIT_op_b_n(RES, 5, B)
9'h1A9: `DECODER_CB_BIT_op_b_n(RES, 5, C)
9'h1AA: `DECODER_CB_BIT_op_b_n(RES, 5, D)
9'h1AB: `DECODER_CB_BIT_op_b_n(RES, 5, E)
9'h1AC: `DECODER_CB_BIT_op_b_n(RES, 5, H)
9'h1AD: `DECODER_CB_BIT_op_b_n(RES, 5, L)
9'h1AE: `DECODER_CB_RES_SET_op_b_MEM_HL(REs, 5)
9'h1AF: `DECODER_CB_BIT_op_b_n(REs, 5, A)
9'h1B0: `DECODER_CB_BIT_op_b_n(REs, 6, B)
9'h1B1: `DECODER_CB_BIT_op_b_n(REs, 6, C)
9'h1B2: `DECODER_CB_BIT_op_b_n(REs, 6, D)
9'h1B3: `DECODER_CB_BIT_op_b_n(REs, 6, E)
9'h1B4: `DECODER_CB_BIT_op_b_n(REs, 6, H)
9'h1B5: `DECODER_CB_BIT_op_b_n(REs, 6, L)
9'h1B6: `DECODER_CB_RES_SET_op_b_MEM_HL(REs, 6)
9'h1B7: `DECODER_CB_BIT_op_b_n(REs, 6, A)
9'h1B8: `DECODER_CB_BIT_op_b_n(REs, 7, B)
9'h1B9: `DECODER_CB_BIT_op_b_n(REs, 7, C)
9'h1BA: `DECODER_CB_BIT_op_b_n(REs, 7, D)
9'h1BB: `DECODER_CB_BIT_op_b_n(REs, 7, E)
9'h1BC: `DECODER_CB_BIT_op_b_n(REs, 7, H)
9'h1BD: `DECODER_CB_BIT_op_b_n(REs, 7, L)
9'h1BE: `DECODER_CB_RES_SET_op_b_MEM_HL(REs, 7)
9'h1BF: `DECODER_CB_BIT_op_b_n(REs, 7, A)
9'h1C0: `DECODER_CB_BIT_op_b_n(SET, 0, B)
9'h1C1: `DECODER_CB_BIT_op_b_n(SET, 0, C)
9'h1C2: `DECODER_CB_BIT_op_b_n(SET, 0, D)
9'h1C3: `DECODER_CB_BIT_op_b_n(SET, 0, E)
9'h1C4: `DECODER_CB_BIT_op_b_n(SET, 0, H)
9'h1C5: `DECODER_CB_BIT_op_b_n(SET, 0, L)
9'h1C6: `DECODER_CB_RES_SET_op_b_MEM_HL(SET, 0)
9'h1C7: `DECODER_CB_BIT_op_b_n(SET, 0, A)
9'h1C8: `DECODER_CB_BIT_op_b_n(SET, 1, B)
9'h1C9: `DECODER_CB_BIT_op_b_n(SET, 1, C)
9'h1CA: `DECODER_CB_BIT_op_b_n(SET, 1, D)
9'h1CB: `DECODER_CB_BIT_op_b_n(SET, 1, E)
9'h1CC: `DECODER_CB_BIT_op_b_n(SET, 1, H)
9'h1CD: `DECODER_CB_BIT_op_b_n(SET, 1, L)
9'h1CE: `DECODER_CB_RES_SET_op_b_MEM_HL(SET, 1)
9'h1CF: `DECODER_CB_BIT_op_b_n (SET, 1, A)
9'h1D0: `DECODER_CB_BIT_op_b_n (SET, 2, B)
9'h1D1: `DECODER_CB_BIT_op_b_n (SET, 2, C)
9'h1D2: `DECODER_CB_BIT_op_b_n (SET, 2, D)
9'h1D3: `DECODER_CB_BIT_op_b_n (SET, 2, E)
9'h1D4: `DECODER_CB_BIT_op_b_n (SET, 2, H)
9'h1D5: `DECODER_CB_BIT_op_b_n (SET, 2, L)
9'h1D6: `DECODER_CB_RES_SET_op_b_MEM_HL (SET, 2)
9'h1D7: `DECODER_CB_BIT_op_b_n (SET, 2, A)
9'h1D8: `DECODER_CB_BIT_op_b_n (SET, 3, B)
9'h1D9: `DECODER_CB_BIT_op_b_n (SET, 3, C)
9'h1DA: `DECODER_CB_BIT_op_b_n (SET, 3, D)
9'h1DB: `DECODER_CB_BIT_op_b_n (SET, 3, E)
9'h1DC: `DECODER_CB_BIT_op_b_n (SET, 3, H)
9'h1DD: `DECODER_CB_BIT_op_b_n (SET, 3, L)
9'h1DE: `DECODER_CB_RES_SET_op_b_MEM_HL (SET, 3)
9'h1DF: `DECODER_CB_BIT_op_b_n (SET, 3, A)
9'h1E0: `DECODER_CB_BIT_op_b_n (SET, 4, B)
9'h1E1: `DECODER_CB_BIT_op_b_n (SET, 4, C)
9'h1E2: `DECODER_CB_BIT_op_b_n (SET, 4, D)
9'h1E3: `DECODER_CB_BIT_op_b_n (SET, 4, E)
9'h1E4: `DECODER_CB_BIT_op_b_n (SET, 4, H)
9'h1E5: `DECODER_CB_BIT_op_b_n (SET, 4, L)
9'h1E6: `DECODER_CB_RES_SET_op_b_MEM_HL (SET, 4)
9'h1E7: `DECODER_CB_BIT_op_b_n (SET, 4, A)
9'h1E8: `DECODER_CB_BIT_op_b_n (SET, 5, B)
9'h1E9: `DECODER_CB_BIT_op_b_n (SET, 5, C)
9'h1EA: `DECODER_CB_BIT_op_b_n (SET, 5, D)
9'h1EB: `DECODER_CB_BIT_op_b_n (SET, 5, E)
9'h1EC: `DECODER_CB_BIT_op_b_n (SET, 5, H)
9'h1ED: `DECODER_CB_BIT_op_b_n (SET, 5, L)
9'h1EE: `DECODER_CB_RES_SET_op_b_MEM_HL (SET, 5)
9'h1EF: `DECODER_CB_BIT_op_b_n (SET, 5, A)
9'h1F0: `DECODER_CB_BIT_op_b_n (SET, 6, B)
9'h1F1: `DECODER_CB_BIT_op_b_n (SET, 6, C)
9'h1F2: `DECODER_CB_BIT_op_b_n (SET, 6, D)
9'h1F3: `DECODER_CB_BIT_op_b_n (SET, 6, E)
9'h1F4: `DECODER_CB_BIT_op_b_n (SET, 6, H)
9'h1F5: `DECODER_CB_BIT_op_b_n (SET, 6, L)
9'h1F6: `DECODER_CB_RES_SET_op_b_MEM_HL (SET, 6)
9'h1F7: `DECODER_CB_BIT_op_b_n (SET, 6, A)
9'h1F8: `DECODER_CB_BIT_op_b_n (SET, 7, B)
9'h1F9: `DECODER_CB_BIT_op_b_n (SET, 7, C)
9'h1FA: `DECODER_CB_BIT_op_b_n (SET, 7, D)
9'h1FB: `DECODER_CB_BIT_op_b_n (SET, 7, E)
9'h1FC: `DECODER_CB_BIT_op_b_n (SET, 7, H)
9'h1FD: `DECODER_CB_BIT_op_b_n (SET, 7, L)
9'h1FE: `DECODER_CB_RES_SET_op_b_MEM_HL (SET, 7)
9'h1FF: `DECODER_CB_BIT_op_b_n (SET, 7, A)
endcase
end
else
begin
if (INTQ == 0) `DECODER_INTR(00)
else
begin
for (int i = 0; i <= 4; i++)
begin
if (INTQ[i])
begin
unique case (i)
0: `DECODER_INTR(40)
1: `DECODER_INTR(48)
2: `DECODER_INTR(50)
3: `DECODER_INTR(58)
4: `DECODER_INTR(60)
endcase

break;
end
end
end

NUM_Tcnt = 6'd20;

for (int i = 0; i <= 10; i++)
begin

if (RISC_OPCODE[i] == LD_BPC || RISC_OPCODE[i] == LD_CPC ||
    RISC_OPCODE[i] == LD_DPC || RISC_OPCODE[i] == LD_EPC ||
    RISC_OPCODE[i] == LD_HPC || RISC_OPCODE[i] == LD_LPC ||
    RISC_OPCODE[i] == LD_TPC || RISC_OPCODE[i] == LD_XPC ||
    RISC_OPCODE[i] == LD_APC ||
    RISC_OPCODE[i] == LD_PCB || RISC_OPCODE[i] == LD_PCC ||
    RISC_OPCODE[i] == LD_PCD || RISC_OPCODE[i] == LD_PCE ||
    RISC_OPCODE[i] == LD_PCH || RISC_OPCODE[i] == LD_PCL ||
    RISC_OPCODE[i] == LD_PCT ||
    RISC_OPCODE[i] == LD_PCSPl || RISC_OPCODE[i] == LD_PCPh ||
    RISC_OPCODE[i] == LD_SPlPC || RISC_OPCODE[i] == LD_SPhPC ||
    RISC_OPCODE[i] == JP_R8 || RISC_OPCODE[i] == JP_NZR8 ||
    RISC_OPCODE[i] == JP_ZR8 || RISC_OPCODE[i] == JP_NCR8 ||
    RISC_OPCODE[i] == JP_CR8 )
begin

  isPCMEM[i] = 1;
end
end
end
endmodule
module GB_Z80_ALU
(
    input logic [7:0] OPD1_L,
    input logic [7:0] OPD2_L,
    input GB_Z80_ALU_OPCODE OPCODE,
    input logic [7:0] FLAG, // the F register
    output logic [7:0] STATUS, // updated flag
    output logic [7:0] RESULT_L,
    output logic [7:0] RESULT_H // Not used for 8-bit ALU
);

// int is signed 32 bit 2 state integer
int opd1h_int;
int opd2h_int;
int opd16_int;
int result_int;
logic [7:0] status_int;

assign RESULT_L = result_int[7:0];
assign RESULT_H = result_int[15:8];
assign STATUS = status_int;
assign opd1h_int = {1'b0, OPD1_L};
assign opd2h_int = {1'b0, OPD2_L};
assign opd16_int = {OPD2_L, OPD1_L};

always_comb
begin

result_int = 0;
status_int = FLAG;
unique case (OPCODE)

   ALU_NOP : ;
/* 8-bit Arithmetic */
ALU_ADD, ALU_ADC :
   begin
      result_int = opd1h_int + opd2h_int + ((OPCODE == ALU_ADC) & FLAG [4]);
      status_int[7] = RESULT_L == 0; // Zero Flag (Z)
      status_int[6] = 0; //Subtract Flag (N)
      status_int[5] = opd1h_int[3:0] + opd2h_int[3:0] + ((OPCODE == ALU_ADC) & FLAG[4]) > 5'h0F; // Half Carry Flag (H)
      status_int[4] = result_int[8]; // Carry Flag (C)
   end

ALU_SUB, ALU_SBC, ALU_CP : // SUB and CP are the same command to the ALU
   begin
      result_int = opd2h_int - opd1h_int - ((OPCODE == ALU_SBC) & FLAG [4]);
      status_int[7] = RESULT_L == 0;
      status_int[6] = 1;
      status_int[5] = {1'b0, opd2h_int[3:0]} < ({1'b0, opd1h_int[3:0]} + ((OPCODE == ALU_SBC) & FLAG[4]));
      status_int[4] = opd2h_int < (opd1h_int + ((OPCODE == ALU_SBC) & FLAG[4]));
   end

ALU_AND :
   begin
      result_int = opd1h_int & opd2h_int;
      status_int[7] = RESULT_L == 0;
      status_int[6] = 0;
      status_int[5] = 1;
      status_int[4] = 0;
end

ALU_OR :

begin
    result_int = opd1h_int | opd2h_int;
    status_int[7] = RESULT_L == 0;
    status_int[6] = 0;
    status_int[5] = 0;
    status_int[4] = 0;
end

ALU_XOR :

begin
    result_int = opd1h_int ^ opd2h_int;
    status_int[7] = RESULT_L == 0;
    status_int[6] = 0;
    status_int[5] = 0;
    status_int[4] = 0;
end

ALU_INC :

begin
    result_int = opd1h_int + 1;
    status_int[7] = RESULT_L == 0;
    status_int[6] = 0;
    status_int[5] = opd1h_int[3:0] == 4'hF;
    status_int[4] = FLAG[4];
end

ALU_DEC :

begin
    result_int = opd1h_int - 1;
    status_int[7] = RESULT_L == 0;
    status_int[6] = 1;
    status_int[5] = opd1h_int[3:0] == 4'h0;
    status_int[4] = FLAG[4];
end
ALU_CPL :
begin
for (int i = 0; i <= 7; i++)
    result_int[i] = ~opd1h_int[i];
status_int[7] = FLAG[7];
status_int[6] = 1;
status_int[5] = 1;
status_int[4] = FLAG[4];
end

ALU_BIT :
begin
status_int[7] = ~opd1h_int[opd2h_int];
status_int[6] = 0;
status_int[5] = 1;
status_int[4] = FLAG[4];
end

ALU_SET :
begin
result_int = opd1h_int;
result_int[opd2h_int] = 1;
end

ALU_RES :
begin
result_int = opd1h_int;
result_int[opd2h_int] = 0;
end

ALU_INC16 :
begin
result_int = opd16_int + 1;
end

ALU_DEC16 :
begin
result_int = opd16_int - 1;
end

ALU_DAA :

begin


status_int[4] = 0;
if (FLAG[5] || (!FLAG[6] && ((opd1h_int & 8'h0F) > 8'h09)))
result_int = result_int | 8'h06;
begin
  result_int = result_int | 8'h60;
  status_int[4] = 1;
end
result_int = FLAG[6] ? opd1h_int - result_int : opd1h_int + result_int;
status_int[7] = RESULT_L == 0;
status_int[5] = 0;
end

SHIFTER_SWAP:

begin
result_int = {opd1h_int[3:0], opd1h_int[7:4]};
status_int[7] = RESULT_L == 0;
status_int[6] = 0;
status_int[5] = 0;
status_int[4] = 0;
end

SHIFTER_RLC :

begin
result_int = {opd1h_int[6:0], opd1h_int[7]};
status_int[7] = RESULT_L == 0;
status_int[6] = 0;
status_int[5] = 0;
status_int[4] = opd1h_int[7];
end
SHIFTER_RL :
begin
  result_int = {opd1h_int[6:0], FLAG[4]};
  status_int[7] = RESULT_L == 0;
  status_int[6] = 0;
  status_int[5] = 0;
  status_int[4] = opd1h_int[7];
end

SHIFTER_RRC :
begin
  result_int = {opd1h_int[0], opd1h_int[7:1]};
  status_int[7] = RESULT_L == 0;
  status_int[6] = 0;
  status_int[5] = 0;
  status_int[4] = opd1h_int[0];
end

SHIFTER_RR :
begin
  result_int = {FLAG[4], opd1h_int[7:1]};
  status_int[7] = RESULT_L == 0;
  status_int[6] = 0;
  status_int[5] = 0;
  status_int[4] = opd1h_int[0];
end

SHIFTER_SLA :
begin
  result_int = {opd1h_int[6:0], 1'b0};
  status_int[7] = RESULT_L == 0;
  status_int[6] = 0;
  status_int[5] = 0;
  status_int[4] = opd1h_int[7];
end

SHIFTER_SRA :
begin
    result_int = {opd1h_int[7], opd1h_int[7:1]};
    status_int[7] = RESULT_L == 0;
    status_int[6] = 0;
    status_int[5] = 0;
    status_int[4] = opd1h_int[0];
end
endcase
end
endmodule

Listing C.1: GB_Z80_SINGLE.sv

/* Internal Registers */
ifndef GB_Z80_CPU_H
    define GB_Z80_CPU_H

typedef struct
{
    logic [7:0] A; logic [7:0] F; // AF, F for Flag
    logic [7:0] B; logic [7:0] C; // BC, nn
    logic [7:0] D; logic [7:0] E; // DE, nn
    logic [7:0] H; logic [7:0] L; // HL, nn
}
logic [7:0] T; logic [7:0] X; // Temp Result
logic [7:0] SPH, SPL; // Stack Pointer
logic [15:0] PC; // Program Counter
}
}

`define WR_nn(n1, n2) 
begin 
    WR_NEXT = 1; 
    ADDR_NEXT = {CPU_REG.``n1, CPU_REG.``n2}; 
end

`define WR_FFn(n) 
begin 
    WR_NEXT = 1; 
    ADDR_NEXT = {8'hFF, CPU_REG.``n}; 
end

`define RD_nn(n1, n2) 
begin 
    RD_NEXT = 1; 
    ADDR_NEXT = {CPU_REG.``n1, CPU_REG.``n2}; 
end

`define RD_FFn(n) 
begin 
    RD_NEXT = 1; 
    ADDR_NEXT = {8'hFF, CPU_REG.``n}; 
end

`define LD_n_n(n1, n2) 
begin 

CPU_REG_NEXT."`n1 = CPU_REG."`n2; \
    end

`define INC_n(n) \
    begin \
        ALU_OPCODE = ALU_INC; \
        ALU_OPD1_L = CPU_REG."`n; \
        CPU_REG_NEXT."`n = ALU_RESULT_L; \
        CPU_REG_NEXT.F = ALU_STATUS; \
    end

`define DEC_n(n) \
    begin \
        ALU_OPCODE = ALU_DEC; \
        ALU_OPD1_L = CPU_REG."`n; \
        CPU_REG_NEXT."`n = ALU_RESULT_L; \
        CPU_REG_NEXT.F = ALU_STATUS; \
    end

// {n1, n2}
`define INC_nn(n1, n2) \
    begin \
        ALU_OPCODE = ALU_INC16; \
        ALU_OPD1_L = CPU_REG."`n2; \
        ALU_OPD2_L = CPU_REG."`n1; \
        CPU_REG_NEXT."`n1 = ALU_RESULT_H; \
        CPU_REG_NEXT."`n2 = ALU_RESULT_L; \
    end

`define DEC_nn(n1, n2) \
    begin \
        ALU_OPCODE = ALU_DEC16; \
        ALU_OPD1_L = CPU_REG."`n2; \
        ALU_OPD2_L = CPU_REG."`n1; \

CPU_REG_NEXT.\`\n1 = ALU_RESULT_H; \
CPU_REG_NEXT.\`\n2 = ALU_RESULT_L; \

end

`define ADDL_n(n) \n
    begin \n    
    ALU_OPCODE = ALU_ADD; \n    ALU_OPD2_L = CPU_REG.L; \n    ALU_OPD1_L = CPU_REG.\`\n; \n    CPU_REG_NEXT.L = ALU_RESULT_L; \n    CPU_REG_NEXT.F = ALU_STATUS; \n    
    end

`define ADCH_n(n) \n
    begin \n    
    ALU_OPCODE = ALU_ADC; \n    ALU_OPD2_L = CPU_REG.H; \n    ALU_OPD1_L = CPU_REG.\`\n; \n    CPU_REG_NEXT.H = ALU_RESULT_L; \n    CPU_REG_NEXT.F = ALU_STATUS; \n    
    end

`define ALU_A_op_n(op, n) \n
    begin \n    
    ALU_OPCODE = ALU_\`\op; \n    ALU_OPD2_L = CPU_REG.A; \n    ALU_OPD1_L = CPU_REG.\`\n; \n    CPU_REG_NEXT.A = ALU_RESULT_L; \n    CPU_REG_NEXT.F = ALU_STATUS; \n    
    end

`define ALU_A_op_Data_in(op) \n
    begin \n
ALU_OPCODE = ALU_``op; \\
ALU_OPD2_L = CPU_REG.A; \\
ALU_OPD1_L = DATA_in; \\
CPU_REG_NEXT.A = ALU_RESULT_L; \\
CPU_REG_NEXT.F = ALU_STATUS; \\
end

`define ALU_op_n(op, n) \
begin \\
ALU_OPCODE = ALU_``op; \\
ALU_OPD2_L = CPU_REG.A; \\
ALU_OPD1_L = CPU_REG.``n; \\
//CPU_REG_NEXT.A = ALU_RESULT_L; \\
CPU_REG_NEXT.F = ALU_STATUS; \\
end

`define ALU_BIT_b_n(b, n) \
begin \\
ALU_OPCODE = ALU_BIT; \\
ALU_OPD2_L = ``b; \\
ALU_OPD1_L = CPU_REG.``n; \\
CPU_REG_NEXT.F = ALU_STATUS; \\
end

`define ALU_SET_RST_op_b_n(op, b, n) \
begin \\
ALU_OPCODE = ALU_``op; \\
ALU_OPD2_L = ``b; \\
ALU_OPD1_L = CPU_REG.``n; \\
CPU_REG_NEXT.``n = ALU_RESULT_L; \\
end
`define ALU_op_Data_in(op) \
    begin \
        ALU_OPCODE = ALU_"op; \
        ALU_OPD2_L = CPU_REG.A; \
        ALU_OPD1_L = DATA_in; \
        //CPU_REG_NEXT.A = ALU_RESULT_L; \
        CPU_REG_NEXT.F = ALU_STATUS; \
    end

`define SHIFTER_op_n(op, n) \
    begin \
        ALU_OPCODE = SHIFTER_"op; \
        ALU_OPD1_L = CPU_REG."n; \
        CPU_REG_NEXT."n = ALU_RESULT_L; \
        CPU_REG_NEXT.F = ALU_STATUS; \
    end

`define DAA \
    begin \
        ALU_OPCODE = ALU_DAA; \
        ALU_OPD1_L = CPU_REG.A; \
        CPU_REG_NEXT.A = ALU_RESULT_L; \
        CPU_REG_NEXT.F = ALU_STATUS; \
    end

`define DO_JPR8 {1'b0, CPU_REG_PC} + {3'b0, DATA_in[6:0]} - {1'b0, 
        DATA_in[7], 7'b000_0000}

    // H and C are based on Unsigned ! added to SP1

`define ADD_SPT \
    begin \
        {CPU_REG_NEXT.SPh, CPU_REG_NEXT.SPl} = {1'b0, CPU_REG.SPh, CPU_REG 
            .SPl} + {3'b0, CPU_REG.T[6:0]} - {1'b0, CPU_REG.T[7], 7'b000_0000}; \
        CPU_REG_NEXT.F = \

Listing C.2: GB_Z80_CPU.vh

```
{ \
   2'b00, \n   (({1'b0, CPU_REG.SP1[3:0]} + {1'b0, CPU_REG.T[3:0]}) > 5'h0F), \
   (({1'b0, CPU_REG.SP1[7:0]} + {1'b0, CPU_REG.T[7:0]}) > 9'h0FF) , \n   CPU_REG.F[3:0] \n}; \nend
```

```
`define LD_HL_SPR8 \nbegin \n{CPU_REG_NEXT.H, CPU_REG_NEXT.L} = {1'b0, CPU_REG.SP, CPU_REG.SP1 } + {3'b0, CPU_REG.T[6:0]} - {1'b0, CPU_REG.T[7], 7'b000_0000}; \n   CPU_REG_NEXT.F = \n   { \
      2'b00, \n      (({1'b0, CPU_REG.SP1[3:0]} + {1'b0, CPU_REG.T[3:0]}) > 5'h0F) , \n      (({1'b0, CPU_REG.SP1[7:0]} + {1'b0, CPU_REG.T[7:0]}) > 9'h0FF ), \n      CPU_REG.F[3:0] \n   }; \nend
```

```
`endif
```

```
`ifndef GB_Z80_DECODER_H
`define GB_Z80_DECODER_H

typedef enum
{

```
/* No Operation */

NOP,

HALT,
STOP,

/* 8-bit register operations */
/* LD r1 <- r2 */
LD_AA,  // 7F, Same as original
LD_AB,  // 78, Same as original
LD_AC,  // 7A, Same as original
LD_AD,
LD_AE,
LD_AH,
LD_AL,
LD_BB,
LD_BA,
LD_BC,
LD_BD,
LD_BE,
LD_BH,
LD_BL,
LD_CA,
LD_CB,
LD_CC,
LD_CD,
LD_CE,
LD_CH,
LD_CL,
LD_DA,
LD_DB,
LD_DC,
LD_DD,
LD_DE,
LD_DH,
LD_DL,
LD_EA,
LD_EB,
LD_EC,
LD_ED,
LD_EE,
LD_EH,
LD_EL,
LD_HA,
LD_HB,
LD_HC,
LD_HD,
LD_HE,
LD_HH,
LD_HL,
LD_LA,
LD_LB,
LD_LC,
LD_LD,
LD_LE,
LD_LL,
LD_LH,
LD_SPLL,  // low side of SP
LD_SPbH,  // high side of SP
LD_PCHL,
LD_SPbH,

LD_HL_SPR8,

/* LD r1 <- (nn) */
LD_APC,
LD_BPC,
LD_CPC,
LD_DPC,
LD_EPC,
LD_HPC,
LD_LPC,
LD_TPC,
LD_XPC,
LD_SP1PC,
LD_SP2PC,
LD_ABC,
LD_ADE,
LD_AHL,
LD_BHL,
LD_CHL,
LD_DHL,
LD_EHL,
LD_HHL,
LD_LHL,
LD_THL,
LD_BSP,
LD_CSP,
LD_DSP,
LD_ESP,
LD_HSP,
LD_LSP,
LD_ASP,
LD_FSP,
LD_PC1SP,
LD_PC2SP,
LD_AHT,
LD_AHC,
LD_ATX,

/* LD (nn) <- r1 */
LD_PCB,
LD_PCC,
LD_PCD,
LD_PCE,
LD_PCH,
LD_PCL,
LD_PCT,
LD_PCSP1,
LD_PCSPh,
LD_BCA,
LD_DEA,
LD_HLA,
LD_HLB,
LD_HLC,
LD_HLD,
LD_HLE,
LD_HLH,
LD_HLL,
LD_HLT,
LD_SPA,
LD_SPB,
LD_SPC,
LD_SPD,
LD_SPE,
LD_SPH,
LD_SPL,
LD_SPF,
LD_SPPCh,
LD_SPPCl,
LD_HTA,
LD_HCA,
LD_TXA,
LD_TXSP1,
LD_TXSPh,

/* Arithmetic Operations */

ADD_AA,  // Write back to A
ADD_AB,
ADD_AC,
ADD_AD,
ADD_AE,
ADD_AH,
ADD_AL,
ADD_AT,
ADD_AHL,
ADD_LC,
ADD_LE,  // 16-bit
ADD_LL,
ADD_LSP1,

ADD_SPT,

ADC_AA,
ADC_AB,
ADC_AC,
ADC_AD,
ADC_AE,
ADC_AH,
ADC_AL,
ADC_AT,
ADC_AHL,
ADC_HB,
ADC_HD,
ADC_HH,
ADC_HSPh,

SUB_AA,
SUB_AB,
SUB_AC,
SUB_AD,
SUB_AE,
SUB_AH,
SUB_AL,
SUB_AT,
SUB_AHL,

SBC_AA,
SBC_AB,
SBC_AC,
SBC_AD,
SBC_AE,
SBC_AH,
SBC_AL,
SBC_AT,
SBC_AHL,

AND_AA,
AND_AB,
AND_AC,
AND_AD,
AND_AE,
AND_AH,
AND_AL,
AND_AT,
AND_AHL,

OR_AA,
OR_AB,
OR_AC,
OR_AD,
OR_AE,
OR_AH,
OR_AL,
OR_AT,
OR_AHL,

XOR_AA,
XOR_AB,
XOR_AC,
XOR_AD,
XOR_AE,
XOR_AH,
XOR_AL,
XOR_AT,
XOR_AHL,

CP_AA,
CP_AB,
CP_AC,
CP_AD,
CP_AE,
CP_AH,
CP_AL,
CP_AT,
CP_AHL,
INC_A,
INC_B,
INC_C,
INC_D,
INC_E,
INC_H,
INC_L,
INC_T,

INC_BC, // 16-bit
INC_DE,
INC_HL,
INC_SP,
INC_TX,

DEC_A,
DEC_B,
DEC_C,
DEC_D,
DEC_E,
DEC_H,
DEC_L,
DEC_T,

DEC_BC, // 16-bit
DEC_DE,
DEC_HL,
DEC_SP,
DEC_TX,

RL_A,
RL_B,
RL_C,
RL_D,
RL_E,
RL_H,
RL_L,
RL_T,

RLC_A,
RLC_B,
RLC_C,
RLC_D,
RLC_E,
RLC_H,
RLC_L,
RLC_T,

RR_A,
RR_B,
RR_C,
RR_D,
RR_E,
RR_H,
RR_L,
RR_T,

RRC_A,
RRC_B,
RRC_C,
RRC_D,
RRC_E,
RRC_H,
RRC_L,
RRC_T,
SLA_A,
SLA_B,
SLA_C,
SLA_D,
SLA_E,
SLA_H,
SLA_L,
SLA_T,
SRA_A,
SRA_B,
SRA_C,
SRA_D,
SRA_E,
SRA_H,
SRA_L,
SRA_T,
SWAP_A,
SWAP_B,
SWAP_C,
SWAP_D,
SWAP_E,
SWAP_H,
SWAP_L,
SWAP_T,
SRL_A,
SRL_B,
SRL_C,
SRL_D,
SRL_E,
SRL_H,
SRL_L,
SRL_T,

DAA,
CPL,
SCF,
CCF,

JP_R8,
JP_NZR8,
JP_ZR8,
JP_NCR8,
JP_CR8,
JP_TX,
JP_Z_TX,
JP_NZ_TX,
JP_C_TX,
JP_NC_TX,

RST_00,
RST_08,
RST_10,
RST_18,
RST_20,
RST_28,
RST_30,
RST_38,
RST_40,
RST_48,
RST_50,
RST_58,
RST_60,

BIT0_A,
BIT1_A,
BIT2_A,
BIT3_A,
BIT4_A,
BIT5_A,
BIT6_A,
BIT7_A,

BIT0_B,
BIT1_B,
BIT2_B,
BIT3_B,
BIT4_B,
BIT5_B,
BIT6_B,
BIT7_B,

BIT0_C,
BIT1_C,
BIT2_C,
BIT3_C,
BIT4_C,
BIT5_C,
BIT6_C,
BIT7_C,

BIT0_D,
BIT1_D,
BIT2_D,
BIT3_D,
BIT4_D,
BIT5_D,
BIT6_D,
BIT7_D,
BIT0_E,
BIT1_E,
BIT2_E,
BIT3_E,
BIT4_E,
BIT5_E,
BIT6_E,
BIT7_E,
BIT0_H,
BIT1_H,
BIT2_H,
BIT3_H,
BIT4_H,
BIT5_H,
BIT6_H,
BIT7_H,
BIT0_L,
BIT1_L,
BIT2_L,
BIT3_L,
BIT4_L,
BIT5_L,
BIT6_L,
BIT7_L,
BIT0_T,
BIT1_T, BIT2_T, BIT3_T, BIT4_T, BIT5_T, BIT6_T, BIT7_T,
RES0_A, RES1_A, RES2_A, RES3_A, RES4_A, RES5_A, RES6_A, RES7_A,
RES0_B, RES1_B, RES2_B, RES3_B, RES4_B, RES5_B, RES6_B, RES7_B,
RES0_C, RES1_C, RES2_C, RES3_C, RES4_C, RES5_C, RES6_C,
RES7_C,
RES0_D,
RES1_D,
RES2_D,
RES3_D,
RES4_D,
RES5_D,
RES6_D,
RES7_D,
RES0_E,
RES1_E,
RES2_E,
RES3_E,
RES4_E,
RES5_E,
RES6_E,
RES7_E,
RES0_H,
RES1_H,
RES2_H,
RES3_H,
RES4_H,
RES5_H,
RES6_H,
RES7_H,
RES0_L,
RES1_L,
RES2_L,
RES3_L,
RES4_L,
RES5_L,
RES6_L,
RES7_L,
RES0_T,
RES1_T,
RES2_T,
RES3_T,
RES4_T,
RES5_T,
RES6_T,
RES7_T,
SET0_A,
SET1_A,
SET2_A,
SET3_A,
SET4_A,
SET5_A,
SET6_A,
SET7_A,
SET0_B,
SET1_B,
SET2_B,
SET3_B,
SET4_B,
SET5_B,
SET6_B,
SET7_B,
SET0_C,
SET1_C,
SET2_C,
SET3_C,
SET4_C,
SET5_C,
SET6_C,
SET7_C,

SET0_D,
SET1_D,
SET2_D,
SET3_D,
SET4_D,
SET5_D,
SET6_D,
SET7_D,

SET0_E,
SET1_E,
SET2_E,
SET3_E,
SET4_E,
SET5_E,
SET6_E,
SET7_E,

SET0_H,
SET1_H,
SET2_H,
SET3_H,
SET4_H,
SET5_H,
SET6_H,
SET7_H,
SET0_L,
SET1_L,
SET2_L,
SET3_L,
SET4_L,
SET5_L,
SET6_L,
SET7_L,
SET0_T,
SET1_T,
SET2_T,
SET3_T,
SET4_T,
SET5_T,
SET6_T,
SET7_T,

EI,
DI,
LATCH_INTQ,
RST_IF
}
)

#define DECODER_LDn_d8(n) \
begin \
RISC_OPCODE[1] = LD_``n``PC; \
NUM_Tcnt = 6`d8; \
end

'define DECODER_LDnn_d16(n1, n2) \
begin \
    RISC_OPCODE[1] = LD_``n2``PC; \
    RISC_OPCODE[3] = LD_``n1``PC; \
    NUM_Tcnt = 6'd12; \
end

'define DECODER_LDnn_A(nn) \
begin \
    RISC_OPCODE[1] = LD_``nn``A; \
    NUM_Tcnt = 6'd8; \
end

'define DECODER_LDA_nn(nn) \
begin \
    RISC_OPCODE[1] = LD_A``nn; \
    NUM_Tcnt = 6'd8; \
end

'define DECODER_ADDHL_nn(n1, n2) \
begin \
    RISC_OPCODE[1] = ADD_L``n2; \
    RISC_OPCODE[2] = ADC_H``n1; \
    NUM_Tcnt = 6'd8; \
end

'define DECODER_DEC_nn(nn) \
begin \
    RISC_OPCODE[1] = DEC_``nn; \
    NUM_Tcnt = 6'd8; \
end

'define DECODER_INC_nn(nn) \

begin \
  RISC_OPCODE[1] = INC_``nn; \n  NUM_Tcnt = 6'd8; \nend \
\n\n`define DECODER_LD_HL_INC_A \nbegin \n  RISC_OPCODE[1] = LD_HLA; \n  RISC_OPCODE[2] = INC_HL; \n  NUM_Tcnt = 6'd8; \nend \
``define DECODER_LD_HL_DEC_A \nbegin \n  RISC_OPCODE[1] = LD_HLA; \n  RISC_OPCODE[2] = DEC_HL; \n  NUM_Tcnt = 6'd8; \nend \
``define DECODER_LD_A_HL_INC \nbegin \n  RISC_OPCODE[1] = LD_AHL; \n  RISC_OPCODE[2] = INC_HL; \n  NUM_Tcnt = 6'd8; \nend \
``define DECODER_LD_A_HL_DEC \nbegin \n  RISC_OPCODE[1] = LD_AHL; \n  RISC_OPCODE[2] = DEC_HL; \n  NUM_Tcnt = 6'd8; \nend \
``define DECODER_INC_MEM_HL \nbegin \n  RISC_OPCODE[1] = LD_THL; \n  RISC_OPCODE[2] = INC_T; \n```
RISC_OPCODE[3] = LD_HLT; \
NUM_Tcnt = 6'd12; \
end \\
#define DECODER_DEC_MEM_HL \ 
begin \ 
  RISC_OPCODE[1] = LD_THL; \
  RISC_OPCODE[2] = DEC_T; \
  RISC_OPCODE[3] = LD_HLT; \
  NUM_Tcnt = 6'd12; \
end \\
#define DECODER_LD_MEM_HL_d8 \ 
begin \ 
  RISC_OPCODE[1] = LD_TPC; \
  RISC_OPCODE[3] = LD_HLT; \
  NUM_Tcnt = 6'd12; \
end \\
#define DECODER_LD_n_MEM_HL(n) \ 
begin \ 
  RISC_OPCODE[2] = LD_
\n  NUM_Tcnt = 6'd8; \
end \\
#define DECODER_LD_MEM_HL_n(n) \ 
begin \ 
  RISC_OPCODE[2] = LD_HL\n\n  NUM_Tcnt = 6'd8; \
end \\
#define DECODER_ALU_op_n (op, n) \ 
begin \ 
  RISC_OPCODE[0] = \op\_A\n\nend \\
#define DECODER_ALU_op_d8 (op) \ 
begin \
RISC_OPCODE[1] = LD_TPC; \
RISC_OPCODE[2] = `op`_AT; \
NUM_Tcnt = 6'd8; \
end
\n`define DECODER_ALU_op_MEM_HL(op) \nbegin \
   RISC_OPCODE[2] = `op`_A`HL; \
   NUM_Tcnt = 6'd8; \
end
\n`define DECODER_RET \nbegin \
   RISC_OPCODE[1] = LD_PC1SP; \n   RISC_OPCODE[2] = INC_SP; \n   RISC_OPCODE[3] = LD_PChSP; \n   RISC_OPCODE[4] = INC_SP; \n   NUM_Tcnt = 6'd16; \nend
\n`define DECODER_RETI \nbegin \
   RISC_OPCODE[1] = LD_PC1SP; \n   RISC_OPCODE[2] = INC_SP; \n   RISC_OPCODE[3] = LD_PChSP; \n   RISC_OPCODE[4] = INC_SP; \n   RISC_OPCODE[5] = EI; \n   NUM_Tcnt = 6'd16; \nend
\n`define DECODER_RET_NZ \nbegin \
   if (!FLAG[7]) \n
begin

RISC_OPCODE[3] = LD_PC1SP; \
RISC_OPCODE[4] = INC_SP; \
RISC_OPCODE[5] = LD_PChSP; \
RISC_OPCODE[6] = INC_SP; \
end \
NUM_Tcnt = FLAG[7] ? 6'd8 : 6'd20; \
end

`define DECORDER_RET_Z \
begin \
if (FLAG[7]) \
begin \
RISC_OPCODE[3] = LD_PC1SP; \
RISC_OPCODE[4] = INC_SP; \
RISC_OPCODE[5] = LD_PChSP; \
RISC_OPCODE[6] = INC_SP; \
end \
NUM_Tcnt = FLAG[7] ? 6'd20 : 6'd8; \
end

`define DECORDER_RET_C \
begin \
if (FLAG[4]) \
begin \
RISC_OPCODE[3] = LD_PC1SP; \
RISC_OPCODE[4] = INC_SP; \
RISC_OPCODE[5] = LD_PChSP; \
RISC_OPCODE[6] = INC_SP; \
end \
NUM_Tcnt = FLAG[4] ? 6'd20 : 6'd8; \
end


`define DECODER_RET_NC
begin

    if (!FLAG[4])
    begin
        RISC_OPCODE[3] = LD_PC ISP; \
        RISC_OPCODE[4] = INC_SP; \
        RISC_OPCODE[5] = LD_PChSP; \
        RISC_OPCODE[6] = INC_SP; \
        end
    NUM_Tcnt = FLAG[4] ? 6'd8 : 6'd20; \
end

`define DECODER_PUSH_nn(n1, n2)
begin
    RISC_OPCODE[2] = DEC_SP; \
    RISC_OPCODE[3] = LD_SP``n1; \
    RISC_OPCODE[4] = DEC_SP; \
    RISC_OPCODE[5] = LD_SP``n2; \
    NUM_Tcnt = 6'd16; \
end

`define DECODER_POP_nn(n1, n2)
begin
    RISC_OPCODE[2] = LD_``n2``SP; \
    RISC_OPCODE[3] = INC_SP; \
    RISC_OPCODE[4] = LD_``n1``SP; \
    RISC_OPCODE[5] = INC_SP; \
    NUM_Tcnt = 6'd12; \
end

`define DECODER_JP_Z_a16
begin
    RISC_OPCODE[1] = LD_XPC; \
end
RISC_OPCODE[3] = LD_TPC; \
RISC_OPCODE[6] = JP_Z_TX; \
NUM_Tcnt = FLAG[7] ? 6'd16 : 6'd12; \
end 

`define DECODER_JP_NZ_a16 \
begin \
  RISC_OPCODE[1] = LD_XPC; \
  RISC_OPCODE[3] = LD_TPC; \
  RISC_OPCODE[6] = JP_NZ_TX; \
  NUM_Tcnt = FLAG[7] ? 6'd12 : 6'd16; \
end 

`define DECODER_JP_C_a16 \
begin \
  RISC_OPCODE[1] = LD_XPC; \
  RISC_OPCODE[3] = LD_TPC; \
  RISC_OPCODE[6] = JP_C_TX; \
  NUM_Tcnt = FLAG[4] ? 6'd16 : 6'd12; \
end 

`define DECODER_JP_NC_a16 \
begin \
  RISC_OPCODE[1] = LD_XPC; \
  RISC_OPCODE[3] = LD_TPC; \
  RISC_OPCODE[6] = JP_NC_TX; \
  NUM_Tcnt = FLAG[4] ? 6'd12 : 6'd16; \
end 

`define DECODER_JP_a16 \
begin \
  RISC_OPCODE[1] = LD_XPC; \
RISC_OPCODE[3] = LD_TPC; \
NUM_Tcnt = FLAG[7] ? 6'd16 : 6'd12; \
end
RISC_OPCODE[3] = LD_TPC; \
RISC_OPCODE[6] = JP_TX; \
NUM_Tcnt = 6'd16; \
end

`define DECODER_CALL_a16 \
begin \
    RISC_OPCODE[2] = LD_XPC; \
    RISC_OPCODE[3] = LD_TPC; \
    RISC_OPCODE[5] = DEC_SP; \
    RISC_OPCODE[6] = LD_SPPCh; \
    RISC_OPCODE[7] = DEC_SP; \
    RISC_OPCODE[8] = LD_SPPCl; \
    RISC_OPCODE[9] = JP_TX; \
    NUM_Tcnt = 6'd24; \
end

`define DECODER_CALL_Z_a16 \
begin \
    RISC_OPCODE[2] = LD_XPC; \
    RISC_OPCODE[3] = LD_TPC; \
    if (FLAG[7]) \
    begin \
        RISC_OPCODE[5] = DEC_SP; \
        RISC_OPCODE[6] = LD_SPPCh; \
        RISC_OPCODE[7] = DEC_SP; \
        RISC_OPCODE[8] = LD_SPPCl; \
        RISC_OPCODE[9] = JP_Z_TX; \
    end \
    NUM_Tcnt = FLAG[7] ? 6'd24 : 6'd12; \
end

`define DECODER_CALL_NZ_a16 \

begin \
RISC_OPCODE[2] = LD_XPC; \
RISC_OPCODE[3] = LD_TPC; \
if (!FLAG[7]) \
begin \
  RISC_OPCODE[5] = DEC_SP; \
  RISC_OPCODE[6] = LD_SPPCh; \
  RISC_OPCODE[7] = DEC_SP; \
  RISC_OPCODE[8] = LD_SPPC1; \
  RISC_OPCODE[9] = JP_NZ_TX; \
end \
NUM_Tcnt = FLAG[7] ? 6'd12 : 6'd24; \
end 

`define DECODER_CALL_C_a16 \nbegin \
  RISC_OPCODE[2] = LD_XPC; \
  RISC_OPCODE[3] = LD_TPC; \
  if (FLAG[4]) \nbegin \
    RISC_OPCODE[5] = DEC_SP; \
    RISC_OPCODE[6] = LD_SPPCh; \
    RISC_OPCODE[7] = DEC_SP; \
    RISC_OPCODE[8] = LD_SPPC1; \
    RISC_OPCODE[9] = JP_C_TX; \
end \
  NUM_Tcnt = FLAG[4] ? 6'd24 : 6'd12; \
end 

`define DECODER_CALL_NC_a16 \nbegin \
  RISC_OPCODE[2] = LD_XPC; \
  RISC_OPCODE[3] = LD_TPC; \

if (!(FLAG[4])) {
    begin 
    RISC_OPCODE[5] = DEC_SP; \
    RISC_OPCODE[6] = LD_SPPCh; \n
    RISC_OPCODE[7] = DEC_SP; \n
    RISC_OPCODE[8] = LD_SPPCl; \n
    RISC_OPCODE[9] = JP_NC_TX; \n
    end \n
    NUM_Tcnt = FLAG[4] ? 6'd12 : 6'd24; \n
end

`define DECODER_RST(addr) \
begin 
    RISC_OPCODE[2] = DEC_SP; \n
    RISC_OPCODE[3] = LD_SPPCh; \n
    RISC_OPCODE[4] = DEC_SP; \n
    RISC_OPCODE[5] = LD_SPPCl; \n
    RISC_OPCODE[6] = RST_``addr; \n
    NUM_Tcnt = 6'd16; \n
end

// Read/Write timing is important for TIMER
`define DECODER_LDh_a8_A \
begin 
    RISC_OPCODE[1] = LD_TPC; \n
    RISC_OPCODE[3] = LD_HTA; \n
    NUM_Tcnt = 6'd12; \n
end

`define DECODER_LDh_a8_A_a8 \
begin 
    RISC_OPCODE[1] = LD_TPC; \n
    RISC_OPCODE[3] = LD_AHT; \n
end

`define DECODER_LDh_A_a8 \
begin 
    RISC_OPCODE[1] = LD_TPC; \n
    RISC_OPCODE[3] = LD_AHT; \n
end
```vhdl
  `define DECODER_LD_H_C_A \
    begin \ 
      RISC_OPCODE[2] = LD_HCA; \ 
      NUM_Tcnt = 6'd8; \ 
    end \

  `define DECODER_LD_H_A_C \
    begin \ 
      RISC_OPCODE[2] = LD_AHC; \ 
      NUM_Tcnt = 6'd8; \ 
    end \

  `define DECODER_ADD_SP_R8 \
    begin \ 
      RISC_OPCODE[1] = LD_TPC; \ 
      RISC_OPCODE[3] = ADD_SPT; \ 
      NUM_Tcnt = 6'd16; \ 
    end \

  `define DECODER_LD_HL_SPR8 \
    begin \ 
      RISC_OPCODE[1] = LD_TPC; \ 
      RISC_OPCODE[3] = LD_HL_SPR8; \ 
      NUM_Tcnt = 6'd12; \ 
    end \

  `define DECODER_LD_a16_SP \
    begin \ 
      RISC_OPCODE[1] = LD_XPC; \ 
      RISC_OPCODE[3] = LD_TPC; \ 
```
RISC_OPCODE[6] = LD_TXSP1; \
RISC_OPCODE[7] = INC_TX; \
RISC_OPCODE[8] = LD_TXSPh; \
NUM_Tcnt = 6'd20; \\
end

#define DECODER_LD_a16_A \
begin \
    RISC_OPCODE[1] = LD_XPC; \
    RISC_OPCODE[3] = LD_TPC; \
    RISC_OPCODE[5] = LD_TXA; \
    NUM_Tcnt = 6'd16; \\
end

#define DECODER_LD_A_a16 \
begin \
    RISC_OPCODE[1] = LD_XPC; \
    RISC_OPCODE[3] = LD_TPC; \
    RISC_OPCODE[5] = LD_ATX; \
    NUM_Tcnt = 6'd16; \\
end

#define DECODER_CB_ALU_op_MEM_HL(op) \
begin \
    RISC_OPCODE[2] = LD_THL; \
    RISC_OPCODE[3] = `op`_T; \
    RISC_OPCODE[4] = LD_HLT; \
    NUM_Tcnt = 6'd12; \\
end

#define DECODER_CB_BIT_op_b_n(op, b, n) \
begin \
    RISC_OPCODE[0] = `op````b``_``n; \

// Cycle count is wrong on the html

`define DECODER_CB_BIT_op_b_MEM_HL(op, b) \
begin \
  RISC_OPCODE[1] = LD_THL; \
  RISC_OPCODE[2] = `op``b``T; \
  NUM_Tcnt = 6'd8; \
end

`define DECODER_CB_RES_SET_op_b_MEM_HL(op, b) \
begin \
  RISC_OPCODE[1] = LD_THL; \
  RISC_OPCODE[2] = `op``b``T; \
  RISC_OPCODE[3] = LD_HLT; \
  NUM_Tcnt = 6'd12; \
end

`define DECODER_INTR(addr)\ 
begin \
  RISC_OPCODE[0] = DI; \
  RISC_OPCODE[1] = DEC_SP; \
  RISC_OPCODE[2] = LD_SPPCh; \
  RISC_OPCODE[3] = LATCH_INTQ; \
  RISC_OPCODE[4] = RST_IF; \
  RISC_OPCODE[5] = DEC_SP; \
  RISC_OPCODE[6] = LD_SPPCl; \
  RISC_OPCODE[7] = RST_``addr; \
  NUM_Tcnt = 6'd20; \
end
Listing C.3: GB_Z80_DECODER.vh

```vh
/* This are the ALU OPCODEs */
ifndef GB_Z80_ALU_H
#define GB_Z80_ALU_H

typedef enum {
    ALU_NOP,
    ALU_ADD,
    ALU_ADC,
    ALU_SUB,
    ALU_SBC,
    ALU_CP,
    ALU_AND,
    ALU_OR,
    ALU_XOR,
    ALU_INC,
    ALU_DEC,
    ALU_CPL,
    ALU_BIT,
    ALU_SET,
    ALU_RES,
    ALU_INC16, // 16 bit alu operation
    ALU_DEC16, // 16 bit alu operation
    ALU_DAA,

    /* Shifter Operations */
    SHIFTER_SWAP,
    SHIFTER_RLC,
    SHIFTER_RL,
    SHIFTER_RRC
}
```

Listing C.4: GB_Z80_ALU.vh

`timescale 1ns / 1ns

// `include "PPU.vh"
`define NO_BOOT 0

module PPU3
(
    input logic clk,
    input logic rst,

    input logic [15:0] ADDR,
    input logic WR,
    input logic RD,
    input logic [7:0] MMIO_DATA_out,
    output logic [7:0] MMIO_DATA_in,

    output logic IRQ_V_BLANK,
    output logic IRQ_LCDC,

    output logic [1:0] PPU_MODE,
    output logic PPU_RD,
    output logic [15:0] PPU_ADDR,
)
input logic [7:0] PPU_DATA_in,

output logic [1:0] PX_OUT,
output logic PX_valid
);

logic [7:0] LCDC, STAT, SCX, SCY, LYC, DMA, BGP, OBPO, OBPI, WX, WY; // Register alias

logic [7:0] FF40, FF40_NEXT;
asgn LCDC = FF40;

logic [7:0] FF41, FF41_NEXT;
asgn STAT = FF41;

logic [7:0] FF42, FF42_NEXT;
asgn SCY = FF42;

logic [7:0] FF43, FF43_NEXT;
asgn SCX = FF43;

logic [7:0] FF44;

logic [7:0] FF45, FF45_NEXT;
asgn LYC = FF45;

logic [7:0] FF46, FF46_NEXT;
asgn DMA = FF46;

logic [7:0] FF47, FF47_NEXT;
asgn BGP = FF47;

logic [7:0] FF48, FF48_NEXT;
assign OBP0 = {FF48[7:2], 2'b00}; // Last 2 bits are not used

logic [7:0] FF49, FF49_NEXT;
assign OBP1 = {FF49[7:2], 2'b00};

logic [7:0] FF4A, FF4A_NEXT;
assign WY = FF4A;

logic [7:0] FF4B, FF4B_NEXT;
assign WX = FF4B;

typedef enum {OAM_SEARCH, RENDER, H_BLANK, V_BLANK} PPU_STATE_t;

PPU_STATE_t PPU_STATE, PPU_STATE_NEXT;

// Current Coordinates
logic [7:0] LX, LX_NEXT; // LX starts from 0, LCD starts from LX + SCX & 7
logic [7:0] LY, LY_NEXT;
assign FF44 = LY;

// OAM Machine
logic OAM_SEARCH_GO;
logic [15:0] OAM_SEARCH_PPU_ADDR;

// BGWD Machine
logic BGWD_RENDER_GO;
logic SHIFT_REG_GO;

// Current Rendering Tile Map Pattern Number
logic [7:0] BG_MAP;
logic [7:0] WD_MAP;
logic [7:0] SP_MAP;
// PPU Running Counter for every 60Hz refresh
short int unsigned PPU_CNT, PPU_CNT_NEXT;
logic [2:0] SCX_CNT, SCX_CNT_NEXT;

//assign IRQ_V_BLANK = (LY == 144 && PPU_CNT == 0);

// Sprite Logic
logic isSpriteOnLine;
assign isSpriteOnLine = ((( PPU_DATA_in + (LCDC[2] << 3)) > (LY + 8)) && (PPU_DATA_in <= (LY + 16)));
logic [3:0] sp_table_cnt; //sp_table_cnt_next;
logic [5:0] sp_name_table [0:9]; //logic [5:0] sp_name_table_next [0:9];
logic [7:0] sp_name_table_x [0:9];

genvar sp_n_gi;
generate
for (sp_n_gi = 0; sp_n_gi < 10; sp_n_gi++)
begin : sp_n_gen
    assign sp_name_table_x[sp_n_gi] = {sp_name_table[sp_n_gi], 2'b00};
end
endgenerate

logic [7:0] sp_y_table [0:9]; //logic [7:0] sp_y_table_next [0:9];
logic [7:0] sp_x_table [0:9]; //logic [7:0] sp_x_table_next [0:9];
logic sp_found; //sp_found_next; // Search Result
logic isHitSP; // is there a sprite to fetch on current X?
logic [3:0] sp_to_fetch;
logic [9:0] sp_not_used, sp_not_used_next; // which sprite has been used
logic SP_RENDER_GO;
//logic [15:0] SPRITE_PPU_ADDR;
logic [9:0] SP_SHIFT_REG_LOAD;
logic [8:0] SP_TILE_DATA0, SP_TILE_DATA1;
logic [1:0] SP_PX_MAP [9:0];
logic [3:0] SP_NEXT SLOT, SP_NEXT SLOT NEXT;
logic [2:0] SP_CNT;
logic [7:0] SP_FLAG;
logic [1:0] SP_PRIPN [0:9];
logic [1:0] SP_PRIPN_NEXT [0:9];

PPU_SHIFT_REG SP_SHIFT_REG9 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [9]), . q (SP_PX_MAP [9]));
PPU_SHIFT_REG SP_SHIFT_REG8 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [8]), . q (SP_PX_MAP [8]));
PPU_SHIFT_REG SP_SHIFT_REG7 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [7]), . q (SP_PX_MAP [7]));
PPU_SHIFT_REG SP_SHIFT_REG6 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [6]), . q (SP_PX_MAP [6]));
PPU_SHIFT_REG SP_SHIFT_REG5 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [5]), . q (SP_PX_MAP [5]));
PPU_SHIFT_REG SP_SHIFT_REG4 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [4]), . q (SP_PX_MAP [4]));
PPU_SHIFT_REG SP_SHIFT_REG3 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [3]), . q (SP_PX_MAP [3]));
PPU_SHIFT_REG SP_SHIFT_REG2 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [2]), . q (SP_PX_MAP [2]));
PPU_SHIFT_REG SP_SHIFT_REG1 (. clk (clk), . rst (rst), . data ('{SP_TILE_DATA1, SP_TILE_DATA0}), . go (SHIFT_REG_GO), . load (SP_SHIFT_REG_LOAD [1]), . q (SP_PX_MAP [1]));
PPU_SHIFT_REG SP_SHIFT_REG0 (.clk(clk), .rst(rst), .data('SP_TILE_DATA1, SP_TILE_DATA0), .go(SHIFT_REG_GO), .load(SP_SHIFT_REG_LOAD[0]), .q(SP_PW_MAP[0]));

// Fetch Logic

localparam OAM_BASE = 16'hFE00;
logic [15:0] VRAM_DATA_BASE;
assign VRAM_DATA_BASE = LCDC[4] ? 16'h8000 : 16'h9000;

// LY + SCY is the effective Y for Background, LX - 8 is the effective X for Background
// LY - WY is the effective Y for Window, LX - WX - 1 is the effective X for Window

`define GET_BG_TILE_ON_LINE_AT_x(x) (16'h9800 | {LCDC[3], 10'b0}) | {((LY + SCY) & 8'hF8), 2'b00} | (((``x + SCX) & 8'hF8) >> 3)
`define GET_xth_BG_TILE_DATA0(x) LCDC[4] ? VRAM_DATA_BASE + {``x, 4'b0} |
{(LY + SCY) & 7, 1'b0} : VRAM_DATA_BASE -{``x[7], 11'b0} + {``x[6:0], 4'b0} | {(LY + SCY) & 8'h07, 1'b0}
`define GET_xth_BG_TILE_DATA1(x) LCDC[4] ? VRAM_DATA_BASE + {``x, 4'b0} |
{(LY + SCY) & 7, 1'b1} : VRAM_DATA_BASE -{``x[7], 11'b0} + {``x[6:0], 4'b0} | {(LY + SCY) & 8'h07, 1'b1}
`define GET_WD_TILE_ON_LINE_AT_x(x) (16'h9800 | {LCDC[6], 10'b0}) | {((LY - WY) & 8'hF8), 2'b00} | (((``x - WX - 1) & 8'hF8) >> 3)
`define GET_xth_WD_TILE_DATA0(x) LCDC[4] ? VRAM_DATA_BASE + {``x, 4'b0} |
{(LY - WY) & 7, 1'b0} : VRAM_DATA_BASE -{``x[7], 11'b0} + {``x[6:0], 4'b0} | {(LY - WY) & 8'h07, 1'b0}
`define GET_xth_WD_TILE_DATA1(x) LCDC[4] ? VRAM_DATA_BASE + {``x, 4'b0} |
{(LY - WY) & 7, 1'b1} : VRAM_DATA_BASE -{``x[7], 11'b0} + {``x[6:0], 4'b0} | {(LY - WY) & 8'h07, 1'b1}
`define GET_xth_SP_TILE_DATA0(x) SP_FLAG[6] ? 16'h8000 + {``x, 4'b0} |
(((8 + (LCDC[2] << 3) + sp_y_table[sp_to_fetch] - LY - 16 - 1) & 15) << 1)) : 16'h8000 + {``x, 4'b0} | (((LY + 16 - sp_y_table[sp_to_fetch])
& 15) << 1))
```
define GET_xth_SP_TILE_DATA1(x) SP_FLAG[6] ? 16'h8000 + (({'x, 4'b0} |
1)) + 1 : 16'h8000 + (({'x, 4'b0} | ((LY + 16 - sp_y_table[
sp_to_fetch]) & 15) << 1)) + 1
```
logic isHitWD;

// Fetched Data
logic [15:0] BGWD_PPU_ADDR;
//logic bgwd_to_fetch;
logic [2:0] BGWD_CNT;
logic [7:0] BGWD_MAP;
logic [7:0] BGWD_TILE_DATA0, BGWD_TILE_DATA1;
logic isFetchWD, isFetchWD_NEXT;
logic FIRST_FETCH_WD_DONE, FIRST_FETCH_WD_DONE_NEXT;
logic [1:0] BGWD_PX_MAP_A, BGWD_PX_MAP_B;
logic BGWD_SHIFT_REG_SEL, BGWD_SHIFT_REG_SEL_NEXT; // 0 selects A, 1
selects B, selected shift register will run, unselected one will load
logic [1:0] BGWD_SHIFT_REG_LOAD;

PPU_SHIFT_REG BGWD_SHIFT_REG_A (.clk(clk), .rst(rst), .data('{
    BGWD_TILE_DATA1, BGWD_TILE_DATA0}), .go(SHIFT_REG_GO &!
    BGWD_SHIFT_REG_SEL), .load(BGWD_SHIFT_REG_LOAD[0]), .q(BGWD_PX_MAP_A));
PPU_SHIFT_REG BGWD_SHIFT_REG_B (.clk(clk), .rst(rst), .data('{
    BGWD_TILE_DATA1, BGWD_TILE_DATA0}), .go(SHIFT_REG_GO &
    BGWD_SHIFT_REG_SEL), .load(BGWD_SHIFT_REG_LOAD[1]), .q(BGWD_PX_MAP_B));

// Display Logic
logic [1:0] BGWD_PX_MAP;
assign BGWD_PX_MAP = BGWD_SHIFT_REG_SEL ? BGWD_PX_MAP_B : BGWD_PX_MAP_A;
logic [1:0] BGWD_PX_DATA;
assign  BGWD_PX_DATA = {BGP[{BGWD_PX_MAP, 1'b1}],BGP[{BGWD_PX_MAP, 1'b0}]};

always_comb
begin
  PX_OUT = BGWD_PX_DATA;
  if (LCDC[1]) // Sprite Display?
  begin
    for (int i = 9 ; i > -1 ; i --)
      begin
        if (SP_PRIPN[i][1] && (SP_PX_MAP[i] != 2'b00)) // SP below
          begin
            PX_OUT = SP_PRIPN[i][0] ? {OBP1[{SP_PX_MAP[i], 1'b1}],
                                      OBP1[{SP_PX_MAP[i], 1'b0}] : {OBP0[{SP_PX_MAP[i], 1'b1}],
                                      OBP0[{SP_PX_MAP[i], 1'b0}]};
          end
      end
  end
  if (LCDC[0]) // BG Display?
  begin
    PX_OUT = (BGWD_PX_MAP == 2'b00) ? PX_OUT : BGWD_PX_DATA;
  end
  if (LCDC[1]) // Sprite Display?
  begin
    for (int i = 9 ; i > -1 ; i --)
      begin
        if (!SP_PRIPN[i][1] && (SP_PX_MAP[i] != 2'b00)) // SP above
          begin
            PX_OUT = SP_PRIPN[i][0] ? {OBP1[{SP_PX_MAP[i], 1'b1}],
                                      OBP1[{SP_PX_MAP[i], 1'b0}] : {OBP0[{SP_PX_MAP[i], 1'b1}],
                                      OBP0[{SP_PX_MAP[i], 1'b0}]};
          end
logic BGWD_SHIFT_REG_A_VALID, BGWD_SHIFT_REG_A_VALID_NEXT;
logic BGWD_SHIFT_REG_B_VALID, BGWD_SHIFT_REG_B_VALID_NEXT;
logic [2:0] RENDER_CNT, RENDER_CNT_NEXT;

/*/ STAT Interrupts */
logic IRQ_STAT, IRQ_STAT_NEXT; // The Internal IRQ signal, IRQ LCDC

always_ff @(posedge clk)
begin
  if (rst) IRQ_STAT <= 0;
  else IRQ_STAT <= IRQ_STAT_NEXT;
end

always_comb
begin
  IRQ_STAT_NEXT = (FF41_NEXT[6] && LY == LYC) ||
                  (FF41_NEXT[3] && PPU_STATE == H_BLANK) ||
                  (FF41_NEXT[5] && PPU_STATE == OAM_SEARCH) ||
                  ((FF41_NEXT[4] || FF41_NEXT[5]) && PPU_STATE == V_BLANK);
end

assign IRQ_LCDC = IRQ_STAT_NEXT && !IRQ_STAT;

/*/ Register State Machine */
always_ff @ (posedge clk)

begin

if (rst)
begin

FF40 <= `NO_BOOT ? 8'h91 : 0;
FF41 <= 0;
FF42 <= 0;
FF43 <= 0;
FF45 <= 0;
FF46 <= 0;
FF47 <= `NO_BOOT ? 8'hFC : 0;
FF48 <= `NO_BOOT ? 8'hFF : 0;
FF49 <= `NO_BOOT ? 8'hFF : 0;
FF4A <= 0;
FF4B <= 0;
end
else
begin

FF40 <= FF40_NEXT;
FF41 <= FF41_NEXT;
FF42 <= FF42_NEXT;
FF43 <= FF43_NEXT;
FF45 <= FF45_NEXT;
FF46 <= FF46_NEXT;
FF47 <= FF47_NEXT;
FF48 <= FF48_NEXT;
FF49 <= FF49_NEXT;
FF4A <= FF4A_NEXT;
FF4B <= FF4B_NEXT;
end

end

always_comb
begin

    FF40_NEXT = (WR && (ADDR == 16'hFF40)) ? MMIO_DATA_out : FF40;
    FF41_NEXT = (WR && (ADDR == 16'hFF41)) ? {MMIO_DATA_out[7:3], FF41[2:0]} : {FF41[7:3], LYC == LY, PPU_MODE};
    FF42_NEXT = (WR && (ADDR == 16'hFF42)) ? MMIO_DATA_out : FF42;
    FF43_NEXT = (WR && (ADDR == 16'hFF43)) ? MMIO_DATA_out : FF43;
    FF45_NEXT = (WR && (ADDR == 16'hFF45)) ? MMIO_DATA_out : FF45;
    FF46_NEXT = (WR && (ADDR == 16'hFF46)) ? MMIO_DATA_out : FF46;
    FF47_NEXT = (WR && (ADDR == 16'hFF47)) ? MMIO_DATA_out : FF47;
    FF48_NEXT = (WR && (ADDR == 16'hFF48)) ? MMIO_DATA_out : FF48;
    FF49_NEXT = (WR && (ADDR == 16'hFF49)) ? MMIO_DATA_out : FF49;
    FF4A_NEXT = (WR && (ADDR == 16'hFF4A)) ? MMIO_DATA_out : FF4A;
    FF4B_NEXT = (WR && (ADDR == 16'hFF4B)) ? MMIO_DATA_out : FF4B;

    case (ADDR)
        16'hFF40: MMIO_DATA_in = FF40;
        16'hFF41: MMIO_DATA_in = {1'b1, FF41[6:0]};
        16'hFF42: MMIO_DATA_in = FF42;
        16'hFF43: MMIO_DATA_in = FF43;
        16'hFF44: MMIO_DATA_in = FF44;
        16'hFF45: MMIO_DATA_in = FF45;
        16'hFF46: MMIO_DATA_in = FF46;
        16'hFF47: MMIO_DATA_in = FF47;
        16'hFF48: MMIO_DATA_in = FF48;
        16'hFF49: MMIO_DATA_in = FF49;
        16'hFF4A: MMIO_DATA_in = FF4A;
        16'hFF4B: MMIO_DATA_in = FF4B;
        default : MMIO_DATA_in = 8'hFF;
    endcase

end

/* PPU State Machine */
always_ff @(posedge clk)
begin
if (rst)
  begin
    PPU_STATE <= V_BLANK;
    LX <= 0;
    LY <= 8'h91;
    PPU_CNT <= 0;

    sp_not_used <= 10'b11_1111_1111;
    SCX_CNT <= 0;
    isFetchWD <= 0;
    FIRST_FETCH_WD_DONE <= 0;

    BGWD_SHIFT_REG_SEL <= 0;
    BGWD_SHIFT_REG_A_VALID <= 0;
    BGWD_SHIFT_REG_B_VALID <= 0;

    RENDER_CNT <= 0;

    SP_NEXT_SLOT <= 0;

    for (int i = 0; i < 10; i++) SP_PRIPN[i] <= 0;
  end

else
  begin
    PPU_STATE <= PPU_STATE_NEXT;
    LX <= LX_NEXT;
    LY <= LY_NEXT;
    PPU_CNT <= PPU_CNT_NEXT;

    sp_not_used <= sp_not_used_next;
    SCX_CNT <= SCX_CNT_NEXT;
  end
isFetchWD <= isFetchWD_NEXT;
FIRST_FETCH_WD_DONE <= FIRST_FETCH_WD_DONE_NEXT;

BGWD_SHIFT_REG_SEL <= BGWD_SHIFT_REG_SEL_NEXT;
BGWD_SHIFT_REG_A_VALID <= BGWD_SHIFT_REG_A_VALID_NEXT;
BGWD_SHIFT_REG_B_VALID <= BGWD_SHIFT_REG_B_VALID_NEXT;

RENDER_CNT <= RENDER_CNT_NEXT;

SP_NEXT_SLOT <= SP_NEXT_SLOT_NEXT;

for (int i = 0; i < 10; i++) SP_PRIPN[i] <= SP_PRIPN_NEXT[i];

always_comb
begin
   // Registers Defaults
   PPU_STATE_NEXT = PPU_STATE;
   LX_NEXT = LX;
   LY_NEXT = LY;
   PPU_CNT_NEXT = PPU_CNT;

   SCX_CNT_NEXT = SCX_CNT;

   sp_not_used_next = sp_not_used;

   isFetchWD_NEXT = isFetchWD;
   FIRST_FETCH_WD_DONE_NEXT = FIRST_FETCH_WD_DONE;

   BGWD_SHIFT_REG_SEL_NEXT = BGWD_SHIFT_REG_SEL;
end
BGWD_SHIFT_REG_A_VALID_NEXT = BGWD_SHIFT_REG_A_VALID;
BGWD_SHIFT_REG_B_VALID_NEXT = BGWD_SHIFT_REG_B_VALID;

RENDER_CNT_NEXT = RENDER_CNT;

SP_NEXT_SLOT_NEXT = SP_NEXT_SLOT;

for (int i = 0; i < 10; i++) SP_PRIPN_NEXT[i] = SP_PRIPN[i];

// Combinational Defaults
PPU_ADDR = 0;
PPU_RD = 0;
PPU_MODE = 2'b01; // VBLANK

OAM_SEARCH_GO = 0;
BGWD_RENDER_GO = 0;

isHitWD = (WY <= LY) && (LX == WX + 1) && LCDC[5];

SP_RENDER_GO = 0;
SP_SHIFT_REG_LOAD = 0;

SHIFT_REG_GO = 0;
BGWD_SHIFT_REG_LOAD = 2'b00;

PX_valid = 0;

IRQ_V_BLANK = 0;

if (LCDC[7]) // LCD Enable
begin
    PPU_CNT_NEXT = PPU_CNT + 1;
    unique case (PPU_STATE)

OAM_SEARCH:

begin

  PPU_MODE = 2'b10;
  PPU_RD = 1;
  OAM_SEARCH_GO = 1;
  PPU_ADDR = PPU_CNT[0] ? OAM_BASE + (PPU_CNT << 1) - 1 :
             OAM_BASE + (PPU_CNT << 1);
  sp_not_used_next = 10'b11_1111_1111;
  if (PPU_CNT == 79) PPU_STATE_NEXT = RENDER;

end

RENDER:

begin

  PPU_MODE = 2'b11;
  PPU_RD = 1;
  if (isHitWD && !isFetchWD)

    begin
    
      RENDER_CNT_NEXT = 0;
      BGWD_SHIFT_REG_A_VALID_NEXT = 0;
      BGWD_SHIFT_REG_B_VALID_NEXT = 0;
      isFetchWD_NEXT = 1;

    end

  else if(!(BGWD_SHIFT_REG_A_VALID || !BGWD_SHIFT_REG_B_VALID) && RENDER_CNT <= 6)

    begin

      BGWD_RENDER_GO = 1;
      if(!isFetchWD)

        begin

          unique case (BGWD_CNT)

          0: PPU_ADDR = `GET_BG_TILE_ON_LINE_AT_x(LX);
          1: PPU_ADDR = `GET_xth_BG_TILE_DATA0(BGWD_MAP)

          2: PPU_ADDR = `GET_xth_BG_TILE_DATA1(BGWD_MAP)

        end

    end

end
; 3,4,5:
endcase
end
else
begin
unique case (BGWD_CNT)
0: PPU_ADDR = `GET_WD_TILE_ON_LINE_AT_x(LX + {FIRST_FETCH_WD_DONE, 3'b00});
   1: PPU_ADDR = `GET_xth_WD_TILE_DATA0(BGWD_MAP)
   ;
   2: PPU_ADDR = `GET_xth_WD_TILE_DATA1(BGWD_MAP)
   ;
3,4,5:;
endcase
end
if (BGWD_CNT == (5 & {2'b11, !isHitSP})) // Why sprite will only stall 5 - LX & 7 ?
begin
if (BGWD_SHIFT_REG_SEL)
begin
   BGWD_SHIFT_REG_A_VALID_NEXT = 1;
   BGWD_SHIFT_REG_LOAD[0] = 1;
end
else
begin
   BGWD_SHIFT_REG_B_VALID_NEXT = 1;
   BGWD_SHIFT_REG_LOAD[1] = 1;
end
if (!BGWD_SHIFT_REG_A_VALID && !
   BGWD_SHIFT_REG_B_VALID) BGWD_SHIFT_REG_SEL_NEXT = !BGWD_SHIFT_REG_SEL;
if (isFetchWD) FIRST_FETCH_WD_DONE_NEXT = 1;
end
end

else if (isHitSP)

begin

SP_RENDER_GO = 1;

unique case (SP_CNT)

0: PPU_ADDR = OAM_BASE + sp_name_table_x[sp_to_fetch] + 2; // Get Pattern Number

1: PPU_ADDR = OAM_BASE + sp_name_table_x[sp_to_fetch] + 3; // Get Attributes

2,3: PPU_ADDR = `GET_xth_SP_TILE_DATA0(LCDC[2] ? {SP_MAP[7:1], 1'b0} : SP_MAP);

4,5: PPU_ADDR = `GET_xth_SP_TILE_DATA1(LCDC[2] ? {SP_MAP[7:1], 1'b0} : SP_MAP);

endcase

if (SP_CNT == 5)

begin

sp_not_used_next[sp_to_fetch] = 0;

SP_SHIFT_REG_LOAD[SP_NEXT SLOT] = 1;

SP_PRIPN_NEXT[SP_NEXT SLOT] = {SP_FLAG[7], SP_FLAG[4]};

SP_NEXT SLOT NEXT = SP_NEXT SLOT + 1;

end

end

if ((BGWD_SHIFT_REG_A_VALID || BGWD_SHIFT_REG_B_VALID) && !isHitSP && !(isHitWD && !isFetchWD))

begin

RENDER_CNT_NEXT = RENDER_CNT + 1;

SHIFT_REG_GO = 1;

if (SCX_CNT != (SCX & 7)) SCX_CNT_NEXT = SCX_CNT + 1;
else

begin
LX_NEXT = LX + 1;

if (LX >= 8)
    PX_valid = 1; // On screen

end

if (RENDER_CNT == 7)
    begin
        BGWD_SHIFT_REG_SEL_NEXT = !BGWD_SHIFT_REG_SEL;
        if (BGWD_SHIFT_REG_SEL == 0)
            BGWD_SHIFT_REG_A_VALID_NEXT = 0;
        else BGWD_SHIFT_REG_B_VALID_NEXT = 0;
    end
end

if (LX_NEXT == 160 + 8) // Start of Horizontal Blank
    begin
        PPU_STATE_NEXT = H_BLANK;
        isFetchWD_NEXT = 0;
        FIRST_FETCH_WD_DONE_NEXT = 0;
        BGWD_SHIFT_REG_A_VALID_NEXT = 0;
        BGWD_SHIFT_REG_B_VALID_NEXT = 0;
        RENDER_CNT_NEXT = 0;
        sp_not_used_next = 10'b11_1111_1111;
        SP_NEXT_SLOT_NEXT = 0;
        SCX_CNT_NEXT = 0;
    end
end

H_BLANK:
    begin
        PPU_MODE = 2'b00;
        if (PPU_CNT == 455) // end of line
            begin
LY_NEXT = LY + 1;
LX_NEXT = 0;
PPU_CNT_NEXT = 0;
PPU_STATE_NEXT = OAM_SEARCH;
if (LY_NEXT == 144)
begin
    PPU_STATE_NEXT = V_BLANK;
    IRQ_V_BLANK = 1;
end
end

V_BLANK:
begin
    PPU_MODE = 2'b01;
    /*
     " Line 153 takes only a few clocks to complete (the exact
     timings are below). The rest of
     the clocks of line 153 are spent in line 0 in mode 1! "
     */
    if (LY == 153)
begin
        LY_NEXT = 0;
        LX_NEXT = 0;
end
if (PPU_CNT == 455 && LY != 0) // end of line
begin
    LY_NEXT = LY + 1;
    PPU_CNT_NEXT = 0;
end
if (PPU_CNT == 455 && LY == 0)
begin
    PPU_STATE_NEXT = OAM_SEARCH; // end of Vertical Blank
```verilog
PPU_CNT_NEXT = 0;

end
end
case
end
else // LCD is off
begin
    PPU_MODE = 2'b00;
    LY_NEXT = 0;
    LX_NEXT = 0;
    PPU_CNT_NEXT = 0;
    PPU_STATE_NEXT = OAM_SEARCH;
    PPU_CNT_NEXT = 0;
end
end

/* OAM Search Machine */
always_ff @(posedge clk)
begin
    if (rst || PPU_STATE == H_BLANK) // reset at the end of the scanline
    begin
        sp_table_cnt <= 0;
        sp_found <= 0;
        for (int i = 0; i < 10; i++)
        begin
            sp_y_table[i] <= 8'hFF;
            sp_x_table[i] <= 8'hFF;
        end
    end
    else if (OAM_SEARCH_GO)
    begin
        if (!PPU_CNT[0]) // even cycles
        begin

    end
```
if (isSpriteOnLine && (sp_table_cnt < 10))
begin
    sp_table_cnt <= (sp_table_cnt + 1);
    sp_name_table[sp_table_cnt] <= (PPU_CNT >> 1);
    sp_y_table[sp_table_cnt] <= PPU_DATA_in;
    sp_found <= 1;
end
else // odd cycles
begin
    if (sp_found)
    begin
        sp_x_table[sp_table_cnt - 1] <= PPU_DATA_in;
    end
    sp_found <= 0;
end
end

/* BGWD Machine */
always_ff @(posedge clk)
begin
    if (rst || !BGWD_RENDER_GO)
    begin
        BGWD_CNT <= 0;
        BGWD TILE_DATA0 <= 0;
        BGWD TILE_DATA1 <= 0;
        BGWD_MAP <= 0;
    end
else
begin
    BGWD_CNT <= BGWD_CNT == 5 ? 0 : BGWD_CNT + 1;
unique case (BGWD_CNT)
0: BGWD_MAP <= PPU_DATA_in;
1: BGWD_TILE_DATA0 <= PPU_DATA_in;
2: BGWD_TILE_DATA1 <= PPU_DATA_in;
3,4,5:;

endcase
end
end

/* Sprite Machine */
always_ff @(posedge clk)
begin
  if (rst || PPU_STATE == H_BLANK) // reset at the end of the scanline
  begin
    SP_CNT <= 0;
    SP_TILE_DATA0 <= 0;
    SP_TILE_DATA1 <= 0;
    SP_MAP <= 0;
    SP_FLAG <= 0;
  end
  else if (SP_RENDER_GO)
  begin
    SP_CNT <= (SP_CNT == 5) ? 0 : SP_CNT + 1;
    unique case (SP_CNT)
    0: SP_MAP <= PPU_DATA_in;
    1: SP_FLAG <= PPU_DATA_in;
    //2,3: if (!SP_FLAG[5]) SP_TILE_DATA0 <= PPU_DATA_in; else
    SP_TILE_DATA0 <= {<<<{PPU_DATA_in}};
    //4,5: if (!SP_FLAG[5]) SP_TILE_DATA1 <= PPU_DATA_in; else
    SP_TILE_DATA1 <= {<<<{PPU_DATA_in}};
    2: if (!SP_FLAG[5]) SP_TILE_DATA0 <= PPU_DATA_in; else
    SP_TILE_DATA0 <= {PPU_DATA_in[0], PPU_DATA_in[1], PPU_DATA_in[2],
                      PPU_DATA_in[3], PPU_DATA_in[4], PPU_DATA_in[5],
                      PPU_DATA_in[6], PPU_DATA_in[7]};
4: if (!SP_FLAG[5]) SP_TILE_DATA1 <= PPU_DATA_in; else
   SP_TILE_DATA1 <= {PPU_DATA_in[0], PPU_DATA_in[1], PPU_DATA_in[2],
                     PPU_DATA_in[3], PPU_DATA_in[4], PPU_DATA_in[5], PPU_DATA_in[6],
                     PPU_DATA_in[7]};
   3,5;
   endcase
end
input logic go,
input logic load,
output logic [1:0] q
);

logic [7:0] shift_reg [0:1];

always_ff @(posedge clk)
begin
    if (rst)
        begin
            shift_reg[0] <= 0;
            shift_reg[1] <= 0;
        end
    else if (load)
        begin
            shift_reg[0] <= data[0];
            shift_reg[1] <= data[1];
        end
    else
        begin
            if (go)
                begin
                    shift_reg[0][7:1] <= shift_reg[0][6:0];
                    shift_reg[0][0] <= 0;
                    shift_reg[1][7:1] <= shift_reg[1][6:0];
                    shift_reg[1][0] <= 0;
                end
        end
end

assign q = {shift_reg[1][7], shift_reg[0][7]};
Listing C.5: PPU3.sv

```verilog
//

`timescale 1ns / 1ns

`-----
/*
  This is the functional block of Sharp LR35902 AKA DMG-CPU
  Clock Frequency: 4194304(2^22) Hz
  Machine Cycle: 1048576(2^20) Hz
  Port naming based on Gameboy1-cpuboard.gif
*/

`-----

`define NO_BOOT 0

// All tristate signals are redesigned to be separate in/out
module LR35902
(
  input logic clk, // XTAL
  input logic rst, // Power On Reset
  /* Video SRAM */
  input logic [7:0] MD_in, // video sram data
  output logic [7:0] MD_out, // video sram data
  output logic [12:0] MA,
  output logic MWR, // high active
  output logic MCS, // high active
  output logic MDE, // high active
  /* LCD */
  output logic [1:0] LD, // PPU DATA 1-0
  output logic PX_VALID,
)
```
output logic CPG, // CONTROL
output logic CP, // CLOCK
output logic ST, // HORSYNC
output logic CPL, // DATALCH
output logic FR, // ALTSIGL
output logic S, // VERTSYN

/*@ Joy Pads */
input logic P10,
input logic P11,
input logic P12,
input logic P13,
output logic P14,
output logic P15,

/*@ Serial Link */
output logic S_OUT,
input logic S_IN,
input logic SCK_in, // serial link clk in
output logic SCK_out, // serial link clk out

/*@ Work RAM/Cartridge */
output logic CLK_GC, // Game Cartridge Clock
output logic WR, // high active
output logic RD, // high active
output logic CS, // high active
output logic [15:0] A,
input logic [7:0] D_in, // work ram/cartridge data bus
output logic [7:0] D_out, // work ram/cartridge data bus

/*@ Audio */
output logic [15:0] LOUT,
output logic [15:0] ROUT

);
logic [7:0] GB_Z80_D_in;
logic [7:0] GB_Z80_D_out;
logic [15:0] GB_Z80_ADDR;
logic GB_Z80_RD, GB_Z80_WR;
logic GB_Z80_HALT;
logic [4:0] GB_Z80_INTQ;

GB_Z80_SINGLE GB_Z80_CPU(. clk(clk), . rst(rst), . ADDR(GB_Z80_ADDR), .
 DATA_in(GB_Z80_D_in), .DATA_out(GB_Z80_D_out),
 . RD(GB_Z80_RD), . WR(GB_Z80_WR), . CPU_HALT( GB_Z80_HALT), . INTQ(GB_Z80_INTQ));

/* Begin Peripherals for GB-Z80 */

/* ROM Region $0x0000 to 0x7FFF*/

// The Boot Rom is mapped from $0x0000 to $0x00FF if $0xFF50 is not
 written before
logic brom_en, brom_en_next;
logic [7:0] DATA_BROM;
brom boot_rom(. addr(GB_Z80_ADDR[7:0]), . data(DATA_BROM), . clk(~clk));

/* Video RAM Region $0x8000 to $0x9FFF */

/* Cartridge RAM Region $0xA000 to $0xBFFF */

/* Work RAM Region $0xC000 to $0xDFFF */ /* Echo RAM Region $0xE000 to
 $0xFDFF */

/* OAM Region $0xFE00 to $0xFE9F */ /* Reserved Unusable Region $0xFEA0 to
logic OAM_WR;
logic [7:0] DATA_OAM_in;
logic [7:0] DATA_OAM_out;
logic [7:0] OAM_ADDR;
Quartus_single_port_ram_160 OAM(.q(DATA_OAM_in), .addr(OAM_ADDR), .clk(~
    clk), .we(OAM_WR), .data(DATA_OAM_out));

/* Hardware I0 Register Region $0xFF00 to $0xFF4B */
logic [7:0] FF00, FF00_NEXT;
assign P15 = FF00[5];
assign P14 = FF00[4];
logic [7:0] FF0F, FF0F_NEXT; // Interrupt Flag

// Sound
logic MMIO_SOUND_WR, MMIO_SOUND_RD;
logic [7:0] MMIO_SOUND_DATA_in, MMIO_SOUND_DATA_out;
SOUND2 GB_SOUND(.clk(!clk), .rst(rst), .ADDR(GB_Z80_ADDR), .WR(    
    MMIO_SOUND_WR), .RD(MMIO_SOUND_RD), .MMIO_DATA_out(MMIO_SOUND_DATA_out)
    ,
    .MMIO_DATA_in(MMIO_SOUND_DATA_in), .SOUND_LEFT(OUT), .
    SOUND_RIGHT(ROUT));

// Timer
logic MMIO_TIMER_WR, MMIO_TIMER_RD;
logic [7:0] MMIO_TIMER_DATA_in, MMIO_TIMER_DATA_out;
logic IRQ_TIMER;
TIMER GB_TIMER (.clk(clk), .rst(rst), .ADDR(GB_Z80_ADDR), .WR(    
    MMIO_TIMER_WR), .RD(MMIO_TIMER_RD), .MMIO_DATA_out(MMIO_TIMER_DATA_out)
    ,
    .MMIO_DATA_in(MMIO_TIMER_DATA_in), .IRQ_TIMER(IRQ_TIMER));
// DMA Controller
logic [7:0] FF46;
logic [7:0] DMA_ADDR, DMA_ADDR_NEXT;
logic [7:0] DMA_SETUP_ADDR, DMA_SETUP_ADDR_NEXT;
logic [7:0] DMA_SETUP_CNT, DMA_SETUP_CNT_NEXT;
logic DMA_SETUP, DMA_SETUP_NEXT;
typedef enum {DMA_IDLE, DMA_GO} DMA_STATE_t;
DMA_STATE_t DMA_STATE, DMA_STATE_NEXT;
logic [9:0] DMA_CNT, DMA_CNT_NEXT;

/* Reserved Unusable Region $0xFF4C to $0xFF7F */

/* High RAM Region $0xFF80 to $0xFFFE */

/* Interrupt Enable Register $0xFFFF */
logic [7:0] FFFF, FFFF_NEXT;
assign GB_Z80_INTQ = (DMA_STATE == DMA_GO) ? 0 : FFFF_NEXT[4:0] & FFOF_NEXT[4:0];
logic HRAM_WR;
logic [7:0] DATA_HRAM_in;
logic [7:0] DATA_HRAM_out;
Quartus_single_port_ram_128 HRAM (.q(DATA_HRAM_in), .addr(GB_Z80_ADDR[6:0]), .clk(~clk), .we(HRAM_WR), .data(DATA_HRAM_out));

/* PPU */
logic MMIO_PPU_WR, MMIO_PPU_RD;
logic [7:0] MMIO_PPU_DATA_in, MMIO_PPU_DATA_out;
logic IRQ_V_BLANK, IRQ_LCDC;
logic [1:0] PPU_MODE;
```verilog
logic PPU_RD;
logic [7:0] PPU_DATA_in;
logic [15:0] PPU_ADDR;
PPU3 GB_PPU(.clk(clk), .rst(rst), .ADDR(GB_Z80_ADDR), .WR(MMI0_PPU_WR), .RD(MMI0_PPU_RD), .MMIO_DATA_out(MMI0_PPU_DATA_out),
/MMIO_DATA_in(MMI0_PPU_DATA_in), .IRQ_V_BLANK(IRQ_V_BLANK),
/IRQ_LCDC(IRQ_LCDC), .PPU_MODE(PPU_MODE),
/PPU_ADDR(PPU_ADDR), .PPU_RD(PPU_RD), .PPU_DATA_in(PPU_DATA_in),
/PX_OUT(LD), .PX_valid(PX_VALID));

MEM_UNIT
/* Memory Management Unit */
// Map the CPU Memory Address to correct Peripheral Address
always_ff @(posedge clk)
begin
    if (rst)
        begin
            brom_en <= `NO_BOOT ? 0 : 1;
            FF00 <= 8'hCF;
            FF0F <= 8'hE0;
            FFFF <= 8'h00;
            DMA_ADDR <= 0;
            DMA_STATE <= DMA_IDLE;
            DMA_CNT <= 0;
            DMA_SETUP_CNT <= 0;
            DMA_SETUP_ADDR <= 0;
            DMA_SETUP <= 0;
        end
    else
        begin
            brom_en <= brom_en_next;
            FF00 <= FF00_NEXT;
```

163
always_comb
begin
  GB_Z80_D_in = 8'hFF;
  MWR = 0; MDE = 0; MCS = 0;
  MD_out = 0;
  MA = 0;
  A = 0;
  D_out = 0;
  WR = 0; RD = 0; CS = 0;
  HRAM_WR = 0; OAM_WR = 0;
  DATA_HRAM_out = 8'hFF;
  DATA_OAM_out = 8'hFF;
  brom_en_next = brom_en;
  OAM_ADDR = GB_Z80_ADDR[7:0];
  MMIO_PPU_WR = 0; MMIO_PPU_RD = 0; MMIO_PPU_DATA_out = 8'hFF;
  PPU_DATA_in = 8'hFF;
  MMIO_TIMER_WR = 0; MMIO_TIMER_RD = 0; MMIO_TIMER_DATA_out = 8'hFF;
  MMIO_SOUND_WR = 0; MMIO_SOUND_RD = 0; MMIO_SOUND_DATA_out = 8'hFF;
/* Interrupt Register */
FF00\_NEXT = FF00;

FFOF\_NEXT = FFOF;

if (IRQ\_V\_BLANK) FFOF\_NEXT[0] = 1;

if (IRQ\_LCDC) FFOF\_NEXT[1] = 1;

if (IRQ\_TIMER) FFOF\_NEXT[2] = 1;

FFFF\_NEXT = FFFF;

/* Memory Access Handlers */

if (GB\_Z80\_ADDR == 16'hFF50 && GB\_Z80\_WR) brom\_en\_next = 0; // Capture Write to FF50 which disables Boot Rom

/* DMA */

DMA\_STATE\_NEXT = DMA\_STATE;

DMA\_CNT\_NEXT = DMA\_CNT;

DMA\_ADDR\_NEXT = DMA\_ADDR;

DMA\_SETUP\_CNT\_NEXT = DMA\_SETUP\_CNT;

DMA\_SETUP\_ADDR\_NEXT = DMA\_SETUP\_ADDR;

DMA\_SETUP\_NEXT = DMA\_SETUP;

if (GB\_Z80\_ADDR == 16'hFF46 && GB\_Z80\_WR) // Capture DMA write
begin

DMA\_SETUP\_NEXT = 1;

DMA\_SETUP\_CNT\_NEXT = 1;

DMA\_SETUP\_ADDR\_NEXT = GB\_Z80\_D\_out;
end

unique case (DMA\_STATE)

    DMA\_IDLE: DMA\_CNT\_NEXT = 0;

    DMA\_GO:

    begin

        DMA\_CNT\_NEXT = DMA\_CNT + 1;

        OAM\_WR = 1;

        OAM\_ADDR = DMA\_CNT >> 2;

end
if ((DMA_ADDR, 8'h00) + (DMA_CNT >> 2)) >= 16'h8000 && ((DMA_ADDR, 8'h00) + (DMA_CNT >> 2)) <= 16'h9FFF) // Copy from VRAM
  begin
    MA = {DMA_ADDR, 8'h00} + (DMA_CNT >> 2);
    MCS = 1; MOE = 1;
    DATA_OAM_out = MD_in;

    if (GB_Z80_ADDR <= 16'h7FFF || (GB_Z80_ADDR >= 16'hA000 && GB_Z80_ADDR < 16'hFE00)) // Allow CPU to access WRAM/CART Bus at this time
      begin
        A = GB_Z80_ADDR;
        GB_Z80_D_in = D_in;
        D_out = GB_Z80_D_out;
        CS = 1; RD = GB_Z80_RD; WR = GB_Z80_WR;
      end
  end
else // Copy from ROM or Work RAM
  begin
    A = {DMA_ADDR, 8'h00} + (DMA_CNT >> 2);
    CS = 1; RD = 1;
    DATA_OAM_out = D_in;

    if (PPU_MODE == 2'b11 && PPU_ADDR >= 16'h8000 && PPU_ADDR <= 16'h9FFF) // Allow GPU to Access VRAM at this time
      begin
        MA = PPU_ADDR;
        PPU_DATA_in = MD_in;
        MCS = 1; MOE = PPU_RD; MWR = 0;
      end
  end
if (DMA_CNT == ((160 << 2) - 1))
  begin
DMA_CNT_NEXT = 0;
DMA_STATE_NEXT = DMA_IDLE;

end
end
case

if (DMA_SETUP)
begin
DMA.Setup_CNT_NEXT = DMA.Setup_CNT + 1;
if (DMA.Setup_CNT == 3'b100)
begin
DMA.Setup_NEXT = 0;
DMA.Addr_NEXT = DMA.Setup_ADDR;
DMA_CNT_NEXT = 0;
DMA.State_NEXT = DMA.GO;
DMA.Setup_CNT_NEXT = 0;
end
end

/* ADDR MUX */

if (DMA.State == DMA.GO)
begin
if (GB.Z80_ADDR >= 16'hFF80 && GB.Z80_ADDR < 16'hFFFF) // only high ram access is allowed
begin
GB.Z80_D_in = DATA_HRAM_in;
DATA_HRAM_out = GB.Z80_D_out;
HRAM_WR = GB.Z80_WR;
end
end

if (DMA.State != DMA.GO) // DMA has higher priority than any of other
memory access

begin

if (GB_Z80_ADDR >= 16'h0000 && GB_Z80_ADDR <= 16'h0FFF)
begin

A = brom_en ? 0 : GB_Z80_ADDR;
GB_Z80_D_in = brom_en ? DATA_BROM : D_in;
D_out = brom_en ? 0 : GB_Z80_D_out;
CS = brom_en ? 0 : 1;
RD = brom_en ? 0 : GB_Z80_RD;
WR = brom_en ? 0 : GB_Z80_WR;
end

else if (GB_Z80_ADDR >= 16'h0100 && GB_Z80_ADDR <= 16'h7FFF)
begin

A = GB_Z80_ADDR;
GB_Z80_D_in = D_in;
D_out = GB_Z80_D_out;
CS = 1; RD = GB_Z80_RD; WR = GB_Z80_WR;
end

else if (GB_Z80_ADDR >= 16'h8000 && GB_Z80_ADDR <= 16'h9FFF) // VRAM
begin

if (PPU_MODE != 2'b11)
begin

MA = GB_Z80_ADDR;
GB_Z80_D_in = MD_in;
MD_out = GB_Z80_D_out;
MCS = 1; MOE = GB_Z80_RD; MWR = GB_Z80_WR;
end
else GB_Z80_D_in = 16'hFF;
end
else if (GB_Z80_ADDR >= 16'hA000 && GB_Z80_ADDR <= 16'hBFFF) // RAM for MBC
begin
```plaintext
A = GB_Z80_ADDR;
GB_Z80_D_in = D_in;
D_out = GB_Z80_D_out;
CS = 1; RD = GB_Z80_RD; WR = GB_Z80_WR;
end

else if (GB_Z80_ADDR >= 16'hC000 && GB_Z80_ADDR <= 16'hFDFF) // WRAM with its echo
begin
A = GB_Z80_ADDR;
GB_Z80_D_in = D_in;
D_out = GB_Z80_D_out;
CS = 1; RD = GB_Z80_RD; WR = GB_Z80_WR;
end

else if (GB_Z80_ADDR >= 16'hFE00 && GB_Z80_ADDR<= 16'hFEFF)// OAM
begin
if (!PPU_MODE[1])
begin
    GB_Z80_D_in = GB_Z80_ADDR < 16'hFEA0 ? DATA_OAM_in : 8'hFF;
    DATA_OAM_out = GB_Z80_ADDR < 16'hFEA0 ? GB_Z80_D_out : 8'hFF;
    OAM_WR = GB_Z80_ADDR < 16'hFEA0 ? GB_Z80_WR : 0;
end
else GB_Z80_D_in = 16'hFF;
end
else if (GB_Z80_ADDR == 16'hFF00) // JoyPad
begin
    GB_Z80_D_in = {2'b11, FF00[5:4], P13, P12, P11, P10};
    if (GB_Z80_WR) FF00_NEXT = GB_Z80_D_out & 8'h30;
end
else if (GB_Z80_ADDR == 16'hFF01 || GB_Z80_ADDR == 16'hFF02) // Serial
```
begin
    if (GB_Z80_ADDR == 16'hFF01) GB_Z80_D_in = 8'h00;
    if (GB_Z80_ADDR == 16'hFF02) GB_Z80_D_in = 8'h7E;
end

else if (GB_Z80_ADDR == 16'hFF03) GB_Z80_D_in = 8'hFF; // Undocumented

else if (GB_Z80_ADDR >= 16'hFF04 && GB_Z80_ADDR <= 16'hFF07) // Timer
    begin
        MMIO_TIMER_WR = GB_Z80_WR;
        MMIO_TIMER_RD = GB_Z80_RD;
        GB_Z80_D_in = MMIO_TIMER_DATA_in;
        MMIO_TIMER_DATA_out = GB_Z80_D_out;
    end
else if (GB_Z80_ADDR >= 16'hFF08 && GB_Z80_ADDR <= 16'hFF0E)
    GB_Z80_D_in = 8'hFF; // Undocumented
else if (GB_Z80_ADDR == 16'hFF0F) // Interrupt Flag
    begin
        if (GB_Z80_RD) GB_Z80_D_in = {3'b111, FF0F[4:0]};
        if (GB_Z80_WR) FF0F_NEXT = GB_Z80_D_out;
    end
else if (GB_Z80_ADDR >= 16'hFF10 && GB_Z80_ADDR <= 16'hFF3F) // Sound
    begin
        MMIO_SOUND_WR = GB_Z80_WR;
        MMIO_SOUND_RD = GB_Z80_RD;
        GB_Z80_D_in = MMIO_SOUND_DATA_in;
        MMIO_SOUND_DATA_out = GB_Z80_D_out;
    end
else if (GB_Z80_ADDR >= 16'hFF40 && GB_Z80_ADDR <= 16'hFF4B) // PPU
    begin
        MMIO_PPU_WR = GB_Z80_WR;
        MMIO_PPU_RD = GB_Z80_RD;
    end
GB_Z80_D_in = MMIO_PPU_DATA_in;

MMIO_PPU_DATA_out = GB_Z80_D_out;

end

else if (GB_Z80_ADDR >= 16'hFF4C && GB_Z80_ADDR <= 16'hFF7F)

GB_Z80_D_in = 8'hFF; // Unusable

else if (GB_Z80_ADDR >= 16'hFF80 && GB_Z80_ADDR < 16'hFFFF) // High Ram

begin

GB_Z80_D_in = DATA_HRAM_in;

DATA_HRAM_out = GB_Z80_D_out;

HRAM_WR = GB_Z80_WR;

end

else if (GB_Z80_ADDR == 16'hFFFF)

begin

if (GB_Z80_RD) GB_Z80_D_in = FFFF;

if (GB_Z80_WR) FFFF_NEXT = GB_Z80_D_out;

end

else GB_Z80_D_in = 8'hFF;

if (PPU_MODE == 2'b11 && PPU_ADDR >= 16'h8000 && PPU_ADDR <= 16'h9FFF)

begin

MA = PPU_ADDR;

PPU_DATA_in = MD_in;

MCS = 1; MOE = PPU_RD; MWR = 0;

end

if (PPU_MODE[1] && PPU_ADDR >= 16'hFE00 && PPU_ADDR < 16'hFEA0)

begin

OAM_WR = 0; OAM_ADDR = PPU_ADDR;

PPU_DATA_in = DATA_OAM_in;

end

end
if (GB_Z80_ADDR == 16'hFF46 && GB_Z80_RD) // DMA Register can be read anytime
begin
    GB_Z80_D_in = DMA_SETUP_ADDR;
end
end
endmodule

Listing C.6: LR35902.sv

`timescale 1ns / 1ps

//
// GameBoy Sound Peripheral
//

module SOUND2
(
    input logic clk,
    input logic rst,
    input logic [7:0] ADDR,
    input logic WR,
    input logic RD,
    input logic [7:0] MMIO_DATA_out,
    output logic [7:0] MMIO_DATA_in,
output logic [15:0] SOUND_LEFT,
output logic [15:0] SOUND_RIGHT
);

logic [7:0] SOUND_REG [0:22];
logic [7:0] SOUND_REG_NEXT [0:22];
logic [7:0] WAVE_RAM [0:15];
logic [7:0] WAVE_RAM_NEXT [0:15];

logic PWR_RST;
assign PWR_RST = rst || !SOUND_REG_NEXT[22][7];

logic clk_len_ctr, clk_vol_env, clk_sweep;
FRAME_SEQUENCER FS(.clk(clk), .rst(PWR_RST), .*);

logic [3:0] TRIGGER;
assign TRIGGER[0] = SOUND_REG_NEXT[4][7];
assign TRIGGER[1] = SOUND_REG_NEXT[9][7];
assign TRIGGER[2] = SOUND_REG_NEXT[14][7];
assign TRIGGER[3] = SOUND_REG_NEXT[19][7];

logic [3:0] LC_LOAD;
assign LC_LOAD[0] = WR && ADDR == 8'h11;
assign LC_LOAD[1] = WR && ADDR == 8'h16;
assign LC_LOAD[2] = WR && ADDR == 8'h1B;
assign LC_LOAD[3] = WR && ADDR == 8'h20;

logic [3:0] ON;
logic [3:0] SOUND [0:3];

/* Channel 1 */
logic [10:0] CH1_PERIOD;
assign CH1_PERIOD = {SOUND_REG_NEXT[4][2:0], SOUND_REG[3]};
logic [2:0] CH1_SWEEP_PERIOD;
assign CH1_SWEEP_PERIOD = SOUND_REG[0][6:4];
logic CH1_NEGATE;
assign CH1_NEGATE = SOUND_REG[0][3];
logic [2:0] CH1_SWEEP_SHIFT;
assign CH1_SWEEP_SHIFT = SOUND_REG[0][2:0];
logic [10:0] CH1_SQWAVE_PERIOD;
logic CH1_SWEEPER_EN;
logic [1:0] CH1_DUTY;
assign CH1_DUTY = SOUND_REG[1][7:6];
logic [5:0] CH1_LENGTH;
assign CH1_LENGTH = SOUND_REG[1][5:0];
logic [3:0] CH1_VOL_INIT;
assign CH1_VOL_INIT = SOUND_REG[2][7:4];
logic CH1_VOL_MODE;
assign CH1_VOL_MODE = SOUND_REG[2][3];
logic [2:0] CH1_VOL_PERIOD;
assign CH1_VOL_PERIOD = SOUND_REG[2][2:0];
logic CH1_LEN_EN;
assign CH1_LEN_EN = SOUND_REG_NEXT[4][6];
logic CH1_SWEEPER_OVERFLOW;

SWEEPER CH1_SWEPPER (.clk(clk), .clk_sweep(clk_sweep), .rst(rst), .
  ch1_period(CH1_PERIOD), .sweep_period(CH1_SWEEP_PERIOD), .negate( 
    CH1_NEGATE),
  .shift(CH1_SWEEP_SHIFT), .load(TRIGGER[0]), .
  sqwave_period(CH1_SQWAVE_PERIOD), .en(CH1_SWEEPER_EN), .overflow( 
    CH1_SWEEPER_OVERFLOW));

SQ_WAVE CH1_SQ_WAVE(*, .duty(CH1_DUTY), .length(MMI0_DATA_out), .vol_init 
  (CH1_VOL_INIT), .vol_mode(CH1_VOL_MODE), .vol_period(CH1_VOL_PERIOD), .
  period(CH1_SQWAVE_PERIOD),
  .trigger(TRIGGER[0]), .LC_LOAD(LC_LOAD[0]), .len_en(
CH1_LEN_EN), .shut_down(PWR_RST), .ON(ON[0]), .SOUND(SOUND[0]), .overflow(CH1_SWEEPER_OVERFLOW));

/* Channel 2 */
logic [10:0] CH2_PERIOD;
assign CH2_PERIOD = {SOUND_REG_NEXT[9][2:0], SOUND_REG[8]};
logic [1:0] CH2_DUTY;
assign CH2_DUTY = SOUND_REG[6][7:6];
logic [5:0] CH2_LENGTH;
assign CH2_LENGTH = SOUND_REG[6][5:0];
logic [3:0] CH2_VOL_INIT;
assign CH2_VOL_INIT = SOUND_REG[7][7:4];
logic CH2_VOL_MODE;
assign CH2_VOL_MODE = SOUND_REG[7][3];
logic [2:0] CH2_VOL_PERIOD;
assign CH2_VOL_PERIOD = SOUND_REG[7][2:0];
logic CH2_LEN_EN;
assign CH2_LEN_EN = SOUND_REG_NEXT[9][6];
SQ_WAVE CH2_SQ_WAVE(.* .duty(CH2_DUTY), .length(MMI0_DATA_out), .vol_init
 (CH2_VOL_INIT), .vol_mode(CH2_VOL_MODE), .vol_period(CH2_VOL_PERIOD), .
 period(CH2_PERIOD),
 .trigger(TRIGGER[1]), .LC_LOAD(LC_LOAD[1]), .len_en(CH2_LEN_EN), .shut_down(PWR_RST), .ON(ON[1]), .SOUND(SOUND[1]), .
overflow(1'b0));

/* Channel 3 */
logic CH3_POWER;
assign CH3_POWER = SOUND_REG[10][7];
logic [7:0] CH3_LENGTH;
assign CH3_LENGTH = SOUND_REG[11];
logic [1:0] CH3_VOL;
assign CH3_VOL = SOUND_REG[12][6:5];
logic [10:0] CH3_PERIOD;
assign CH3_PERIOD = {SOUND_REG_NEXT[14][2:0], SOUND_REG[13]};
logic CH3_LEN_EN;
assign CH3_LEN_EN = SOUND_REG_NEXT[14][6];

WAVE CH3_WAVE(.*, .power(CH3_POWER), .length(MMIO_DATA_out), .vol(CH3_VOL),
  .period(CH3_PERIOD), .trigger(TRIGGER[2]), .LC_LOAD(LC_LOAD[2]),
  .len_en(CH3_LEN_EN), .shut_down(PWR_RST), .ON(ON[2]), .SOUND
  (SOUND[2]));

/* Channel 4 */
logic [5:0] CH4_LENGTH;
assign CH4_LENGTH = SOUND_REG[16][5:0];
logic [3:0] CH4_VOL_INIT;
assign CH4_VOL_INIT = SOUND_REG[17][7:4];
logic CH4_VOL_MODE;
assign CH4_VOL_MODE = SOUND_REG[17][3];
logic [2:0] CH4_VOL_PERIOD;
assign CH4_VOL_PERIOD = SOUND_REG[17][2:0];
logic [3:0] CH4_SHIFT;
assign CH4_SHIFT = SOUND_REG[18][7:4];
logic CH4_LSFR_MODE;
assign CH4_LSFR_MODE = SOUND_REG[18][3];
logic [2:0] CH4_DIV;
assign CH4_DIV = SOUND_REG[18][2:0];
logic CH4_LEN_EN;
assign CH4_LEN_EN = SOUND_REG_NEXT[19][6];

NOISE CH4_NOISE(.*, .length(CH4_LENGTH), .vol_init(CH4_VOL_INIT),
  .vol_mode(CH4_VOL_MODE), .vol_period(CH4_VOL_PERIOD), .shift(CH4_SHIFT),
logic [3:0] LEFT_EN, RIGHT_EN;
assign LEFT_EN = SOUND_REG[21][7:4];
assign RIGHT_EN = SOUND_REG[21][3:0];
logic [2:0] LEFT_VOL, RIGHT_VOL;
assign LEFT_VOL = SOUND_REG[20][6:4];
assign RIGHT_VOL = SOUND_REG[20][2:0];

always_ff @(posedge clk)
begin
  if (PWR_RST) for (int i = 0; i < 23; i ++) SOUND_REG[i] <= 0;
  else for (int i = 0; i < 23; i ++) SOUND_REG[i] <= SOUND_REG_NEXT[i];

  if (rst) for (int i = 0; i < 16; i ++) WAVE_RAM[i] <= 0;
  else for (int i = 0; i < 16; i ++) WAVE_RAM[i] <= WAVE_RAM_NEXT[i];
end

always_comb
begin
  for (int i = 0; i < 23; i++) SOUND_REG_NEXT[i] = SOUND_REG[i];
  for (int i = 0; i < 16; i++) WAVE_RAM_NEXT[i] = WAVE_RAM[i];
  MMIO_DATA_in = 8'hFF;
  /* Trigger Auto Reset */
  SOUND_REG_NEXT[4][7] = 0;
  SOUND_REG_NEXT[9][7] = 0;
  SOUND_REG_NEXT[14][7] = 0;
  SOUND_REG_NEXT[19][7] = 0;
  if (ADDR <= 8'h26 && ADDR >= 8'h10)
    begin

if (WR) SOUND_REG_NEXT[ADDR - 8'10] = MMIO_DATA_out;
MMIO_DATA_in = SOUND_REG[ADDR - 8'10];

/* REG MASKS */
case (ADDR)
    8'10 : MMIO_DATA_in = MMIO_DATA_in | 8'h80;
    8'h11, 8'h16: MMIO_DATA_in = MMIO_DATA_in | 8'h3F;
    8'h13, 8'h18, 8'h1B, 8'h1D, 8'h20, 8'h15, 8'h1F: MMIO_DATA_in = 8'hFF;
    8'h14, 8'h19, 8'h1E, 8'h23: MMIO_DATA_in = MMIO_DATA_in | 8'hBF;
    8'h1A: MMIO_DATA_in = MMIO_DATA_in | 8'h7F;
    8'h1C: MMIO_DATA_in = MMIO_DATA_in | 8'h9F;
    8'h26: MMIO_DATA_in = {MMIO_DATA_in[7], 3'b111, ON};
endcase
end
else if (ADDR >= 8'h30 && ADDR <= 8'h3F)
begin
    if (WR) WAVE_RAM_NEXT[ADDR - 8'h30] = MMIO_DATA_out;
    MMIO_DATA_in = WAVE_RAM[ADDR - 8'h30];
end

/* Frequency Sweeper */
if (CH1_SLEEPER_EN && clk_sweep)
begin
    SOUND_REG_NEXT[4][2:0] = CH1_SQWAVE_PERIOD[10:8];
    SOUND_REG_NEXT[3] = CH1_SQWAVE_PERIOD[7:0];
end

SOUND_LEFT = 0; SOUND_RIGHT = 0;
for (int i = 0; i < 4; i++)
begin
    if (LEFT_EN[i]) SOUND_LEFT = SOUND_LEFT + SOUND[i];
for (int i = 0; i < 4; i++)
begin
    if (RIGHT_EN[i]) SOUND_RIGHT = SOUND_RIGHT + SOUND[i];
end

SOUND_LEFT = SOUND_LEFT * (LEFT_VOL + 1);
SOUND_RIGHT = SOUND_RIGHT * (RIGHT_VOL + 1);
end

module SWEEPER
(
    input logic clk,
    input logic clk_sweep,
    input logic rst,
    input logic [10:0] ch1_period,
    input logic [2:0] sweep_period,
    input logic negate,
    input logic [2:0] shift,
    input logic load,
    output logic overflow,
    output logic [10:0] sqwave_period,
    output logic en
);

logic [2:0] counter;
logic [2:0] shift_int;
logic [10:0] period;
logic [11:0] period_new;
assign en = (sweep_period != 0 && shift_int != 0);

always_comb
begin
  overflow = 0;
  period_new = {1'b0, period};
  if (en)
  begin
    if (negate) period_new = period - (period >> shift_int);
    else period_new = period + (period >> shift_int);
    if (period_new > 2047) overflow = 1;
  end
end

always_ff @(posedge clk)
begin
  if (rst) begin counter <= 0; period <= 0; shift_int <= 0; end
  else if (load) begin period <= ch1_period; counter <= sweep_period;
  shift_int <= shift; end
  else
  begin
    if (counter != 0 && clk_sweep)
    begin
      counter <= counter - 1;
    end
    if (counter == 0 && clk_sweep)
    begin
      counter <= sweep_period;
    end
  end
end
if (clk_sweep && en && counter == 0 && !overflow) period <= period_new;
end

assign sqwave_period = (en) ? period_new : ch1_period;
endmodule

module FRAME_SEQUENCER
(
    input logic clk,
    input logic rst,
    output logic clk_len_ctr,
    output logic clk_vol_env,
    output logic clk_sweep
);

logic [15:0] counter;

always_ff @(posedge clk)
begin
    if (rst) counter <= 0;
    else counter <= counter + 1;
end

assign clk_vol_env = counter[15] && counter[14:0] == 15'd0;
assign clk_sweep = counter[14] && counter[13:0] == 14'd0;
assign clk_len_ctr = counter[13] && counter [12:0] == 13'd0;
endmodule

module SOUND_TIMER
(  
  input logic clk,  
  input logic rst,  
  input logic load,  
  input logic [13:0] period,  
  output logic tick  
);  

logic [13:0] counter;  
always_ff @(posedge clk)  
begin  
  if (rst) counter <= 0;  
  else if (counter == 0 || load) counter <= period;  
  else counter <= counter - 1;  
end  

assign tick = (counter == 0);  
endmodule  

module LENGTH_COUNTER #( parameter len_max = 64 ) (  
  input logic clk,  
  input logic clk_len_ctr,  
  input logic rst,  
  input logic load,  
  input logic trigger,  
  input logic [7:0] length,  
  output logic en  
);  

logic [8:0] counter;  
always_ff @(posedge clk)
begin
  if (rst) counter <= 0;
  else if (load) counter <= (len_max - length);
  else if (trigger) counter <= len_max;
  else if (counter != 0 && clk_len_ctr) counter <= counter - 1;
end

assign en = (counter != 0);

endmodule

module VOLUME_ENVELOPE
(
  input logic clk,
  input logic clk_vol_env,
  input logic rst,
  input logic load,
  input logic mode,
  input logic [3:0] vol_init,
  input logic [2:0] period,
  output logic [3:0] vol
);

logic [3:0] volume;
logic [2:0] counter;

always_ff @(posedge clk)
begin
  if (rst) begin counter <= 0; volume <= vol_init; end
  else if (load) begin
    counter <= period;
    volume <= vol_init;
  end

end
end
else
begin
    if (clk_vol_env && counter != 0) counter <= counter - 1;
    if (clk_vol_env && counter == 0 && period != 0 && ((mode && volume != 4'hF) || (!mode && volume != 4'h0)))
        begin
            counter <= period;
            volume <= mode ? volume + 1 : volume - 1;
            end
        end
assign vol = period != 0 ? volume : vol_init;
endmodule
module DUTY_CYCLE
(
    input logic clk,
    input logic rst,
    input logic tick,
    input logic [1:0] duty,
    output logic sq_wave
);
logic [7:0] DUTY_TEMPLATE [0:3];
logic [2:0] counter;
assign DUTY_TEMPLATE[0] = 8'b0000_0001;
assign DUTY_TEMPLATE[1] = 8'b1000_0001;
assign DUTY_TEMPLATE[2] = 8'b1000_0111;
assign DUTY_TEMPLATE[3] = 8'b0111_1110;
always_ff @(posedge clk)
begin
    if (rst) counter <= 0;
    else if (tick) counter <= counter + 1;
end

assign sq_wave = DUTY_TEMPLATE[duty][counter];
endmodule

module SQ_WAVE
(
    input logic clk,
    input logic clk_len_ctr,
    input logic clk_vol_env,
    input logic rst,
    input logic [1:0] duty,
    input logic [5:0] length,
    input logic [3:0] vol_init,
    input logic vol_mode,
    input logic [2:0] vol_period,
    input logic [10:0] period,
    input logic trigger,
    input logic len_en,
    input logic shut_down,
    input logic overflow,
    input logic LC_LOAD,
    output logic ON,
    output logic [3:0] SOUND
);

logic tick;
logic sq_wave;
logic en;
logic [3:0] vol;
SOUND_TIMER TIMER(.clk(clk), .rst(rst), .load(trigger), .period({((12'd2048 - period), 2'd0}), .tick(tick));
DUTY_CYCLE DUTY (.*);
LENGTH_COUNTER LC(.clk(clk), .clk_len_ctr(clk_len_ctr), .rst(rst || shut_down), .load(LC_LOAD), .trigger(trigger), .length({2'd0, length}), .en(en));
VOLUME_ENVELOPE ENV(.clk(clk), .clk_vol_env(clk_vol_env), .rst(rst), .load (trigger), .mode(vol_mode), .vol_init(vol_init), .period(vol_period), . vol(vol));
assign ON = en && !shut_down && !overflow;
assign SOUND = (en || !len_en) && !shut_down && !overflow && sq_wave ? vol : 0;
endmodule

module WAVE
(
    input logic clk,
    input logic clk_len_ctr,
    input logic rst,
    input logic power,
    input logic [7:0] length,
    input logic [1:0] vol,
    input logic [10:0] period,
    input logic trigger,
    input logic len_en,
    input logic shut_down,
    input logic [7:0] WAVE_RAM [0:15],
    input logic LC_LOAD,
output logic ON,
output logic [3:0] SOUND
);

logic [4:0] ptr;
logic [4:0] ptr_2;
assign ptr_2 = ptr + 1;
logic [4:0] sample_h, sample_l;
assign sample_h = WAVE_RAM[ptr_2 >> 1][7:4];
assign sample_l = WAVE_RAM[ptr_2 >> 1][3:0];

logic [4:0] sample;

logic tick;
logic en;

always_ff @(posedge clk)
begin
  if (rst)
  begin
    ptr <= 0;
    sample <= 0;
  end
  else if (shut_down) ptr <= 0;
  else if (tick)
  begin
    ptr <= ptr + 1;
    sample <= ptr[0] ? sample_l : sample_h;
  end
end
SOUND_TIMER TIMER (.clk(clk), .rst(rst), .load(trigger), .period({1'b0, 12'b2048 - period, 1'b0}), .tick(tick));
LENGTH_COUNTER #(256) LC(.clk(clk), .clk_len_ctr(clk_len_ctr), .rst(rst ||
assign ON = en && !shut_down && power;
assign SOUND = (en || !len_en) && !shut_down && power && (vol != 0) ?
    sample >> (vol - 1) : 0;

endmodule

module NOISE
(
    input logic clk,
    input logic clk_len_ctr,
    input logic clk_vol_env,
    input logic rst,
    input logic [5:0] length,
    input logic [3:0] vol_init,
    input logic vol_mode,
    input logic [2:0] vol_period,
    input logic [3:0] shift,
    input logic lsfr_mode,
    input logic [2:0] div,
    input logic trigger,
    input logic len_en,
    input logic shut_down,
    input logic LC_LOAD,

    output logic ON,
    output logic [3:0] SOUND
);

logic [14:0] LSFR;
logic tick;
logic en;
logic [3:0] vol;
logic [13:0] period;

assign period = (div == 0) ? 2 << 2 : (2 << 3) * div;

SOUND_TIMER TIMER (. clk (clk), . rst (rst), . load (trigger), . period (period << (shift + 1)), . tick (tick));

LENGTH_COUNTER LC (. clk (clk), . clk_len_ctr (clk_len_ctr), . rst (rst ||
    shut_down), . load (LC_LOAD), . trigger (trigger), . length ({2'd0, length}),
    . en (en));

VOLUME_ENVELOPE ENV (. clk (clk), . clk_vol_env (clk_vol_env), . rst (rst), . load
    (trigger), . mode (vol_mode), . vol_init (vol_init), . period (vol_period), .
    vol (vol));

always_ff @(posedge clk)
begin
    if (rst || trigger)
        begin
            LSFR <= {15{1'b1}};
        end
    else if (tick) LSFR <= lsfr_mode ? {LSFR[1]^LSFR[0], LSFR[14:8], LSFR
        [1]^LSFR[0], LSFR[6:1]}: {LSFR[1]^LSFR[0], LSFR[14:1]};
end

assign ON = en && !shut_down;
assign SOUND = (en || !len_en) && !shut_down && !LSFR[0] ? vol : 0;
Listing C.7: SOUND2.sv

```verilog
`timescale 1ns / 1ps

// Timier for the Gameboy

// Based On http://gbdev.gg8.se/wiki/articles/Timer_Obscure_Behaviour

module TIMER
(
    //
    input logic clk,
    input logic rst,

    input logic [15:0] ADDR,
    input logic WR,
    input logic RD,
    input logic [7:0] MMIO_DATA_out,
    output logic [7:0] MMIO_DATA_in,

    output logic IRQ_TIMER
);

logic [7:0] DIV, TIMA, TMA, TAC;
logic [15:0] BIG_COUNTER, BIG_COUNTER_NEXT;
logic FALL_EDGE_TIMER_CLK;
```
logic TIMER_CLK_PREV, TIMER_CLK_PREV_NEXT, TIMER_CLK_NOW;
logic TIMER_OVERFLOW, TIMER_OVERFLOW_NEXT;
logic [1:0] TIMER_OVERFLOW_CNT, TIMER_OVERFLOW_CNT_NEXT;

logic [7:0] FF04;
assign FF04 = DIV;
assign DIV = BIG_COUNTER[15:8];

logic [7:0] FF05, FF05_NEXT;
assign TIMA = FF05;

logic [7:0] FF06, FF06_NEXT;
assign TMA = FF06;

logic [7:0] FF07, FF07_NEXT;
logic [7:0] TAC_PREV, TAC_PREV_NEXT, TAC_NEXT;
assign TAC = FF07;
assign TAC_NEXT = FF07_NEXT;

/* Main State Machine */
always_ff @(posedge clk)
begin
    if (rst)
        begin
            BIG_COUNTER <= 0;
            FF05 <= 0;
            FF06 <= 0;
            FF07 <= 8'hF8;
            TIMER_CLK_PREV <= 0;
            TIMER_OVERFLOW <= 0;
            TIMER_OVERFLOW_CNT <= 0;
            TAC_PREV <= 0;
        end
else
begin
  BIG_COUNTER <= BIG_COUNTER_NEXT;
  FF05 <= FF05_NEXT;
  FF06 <= FF06_NEXT;
  FF07 <= FF07_NEXT;
  TIMER_CLK_PREV <= TIMER_CLK_PREV_NEXT;
  TIMER_OVERFLOW <= TIMER_OVERFLOW_NEXT;
  TIMER_OVERFLOW_CNT <= TIMER_OVERFLOW_CNT_NEXT;
  TAC_PREV <= TAC_PREV_NEXT;
end
end

always_comb
begin
  FF05_NEXT = FF05;
  FF06_NEXT = FF06;
  FF07_NEXT = FF07;
  if (WR && (ADDR == 16'hFF07)) FF07_NEXT = MMIO_DATA_out;
  TIMER_CLK_NOW = 0;
  FALL_EDGE_TIMER_CLK = 0;
  unique case (TAC[1:0])
    2'd0:
      begin
        TIMER_CLK_PREV_NEXT = BIG_COUNTER[9];
        TIMER_CLK_NOW = BIG_COUNTER[9];
      end
    2'd3:
      begin
        TIMER_CLK_PREV_NEXT = BIG_COUNTER[7];
        TIMER_CLK_NOW = BIG_COUNTER[7];
      end
    2'd2:
begin
    TIMER_CLK_PREV_NEXT = BIG_COUNTER[5];
    TIMER_CLK_NOW = BIG_COUNTER[5];
end
2'd1:
begin
    TIMER_CLK_PREV_NEXT = BIG_COUNTER[3];
    TIMER_CLK_NOW = BIG_COUNTER[3];
end
endcase

TAC_PREV_NEXT = TAC;

FALL_EDGE_TIMER_CLK = (TIMER_CLK_PREV && !TIMER_CLK_NOW && TAC[2]) ||

// FALL_EDGE_TIMER_CLK = (TIMER_CLK_PREV && !TIMER_CLK_NOW && TAC[2])

// FALL_EDGE_TIMER_CLK = (!TIMER_CLK_PREV && TIMER_CLK_NOW && TAC[2])

TIMER_OVERFLOW_NEXT = TIMER_OVERFLOW;
TIMER_OVERFLOW_CNT_NEXT = TIMER_OVERFLOW_CNT;
IRQ_TIMER = 0;
if (FALL_EDGE_TIMER_CLK)
begin
    FF05_NEXT = FF05 + 1; // increase TIMA when there is a falling
    edge of Timer clock
    if (FF05 == 8'hFF)
begin
        TIMER_OVERFLOW_NEXT = 1;
end
end
if (TIMER_OVERFLOW) TIMER_OVERFLOW_CNT_NEXT = TIMER_OVERFLOW_CNT + 1;
if (TIMER_OVERFLOW_CNT == 2'b11)
TIMER_OVERFLOW_NEXT = 0;

BIG_COUNTER_NEXT = (WR && (ADDR == 16'hFF04)) ? 1 : BIG_COUNTER + 1;
// Reset big counter if write into FF04

if (WR && (ADDR == 16'hFF05)) FF05_NEXT = (TIMER_OVERFLOW_CNT == 2'b11) ? FF05 : MMIO_DATA_out; // Latch behavior
if (WR && (ADDR == 16'hFF06))
begin
  FF06_NEXT = MMIO_DATA_out;
  if (TIMER_OVERFLOW_CNT == 2'b11) // Latch behavior
  begin
    FF05_NEXT = MMIO_DATA_out;
  end
end

case (ADDR)
16'hFF04: MMIO_DATA_in = FF04;
16'hFF05: MMIO_DATA_in = FALL_EDGE_TIMER_CLK ? FF05_NEXT : FF05;
// Since the original Timer is Latch based, increase happens at the same clock cycle
16'hFF06: MMIO_DATA_in = FF06;
16'hFF07: MMIO_DATA_in = {5'b11111, FF07[2:0]};
default : MMIO_DATA_in = 8'hFF;
endcase

if (FALL_EDGE_TIMER_CLK) // When TIMA is about to overflow but writing something to it
begin
  if (FF05 == 8'hFF && FF05_NEXT != 8'h00) TIMER_OVERFLOW_NEXT = 0;
end
if (TIMER_OVERFLOW_CNT == 2'b10) FF05_NEXT = FF06_NEXT; // count 3T after overflow
if (TIMER_OVERFLOW && TIMER_OVERFLOW_CNT == 2'b00) IRQ_TIMER = 1; //
INTQ to CPU is delayed by 2T from overflow (Anywhere from 1T-4T is acceptable?)

endmodule

Listing C.8: TIMER.sv

`timescale 1ns / 1ns

//

 //////////////////////////////////////////////////////////////////////////\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n
module LH5264

(  
    input logic [7:0] D_in,  
    input logic [12:0] A,  
    input logic CE1,  
    input logic CE2,  
    input logic clk,  
    input logic OE,  
    output logic [7:0] D_out
);

logic we;
assign we = CE1 && CE2;
logic [7:0] q;
assign D_out = OE ? q : 8'hFF;
Quartus_single_port_ram_8k RAM_8K(.data(D_in), .addr(A), .clk(clk), .we(we ), .q(q));
endmodule

Listing C.9: LH5264.sv

`timescale 1ns / 1ps

//

// This is the MBC 1 Memory Bank Controller for The GameBoy

//

`define SDRAM_RAM_BASE 26'h20000000

module MBC1
(
    input logic clk,
    input logic reset,
    input logic [7:0] NUM_ROM_BANK, // How many ROM banks in this cartridge?
    input logic [7:0] NUM_RAM_BANK, // How many RAM banks in this cartridge?
    input logic [15:0] CART_ADDR,
    output logic [7:0] CART_DATA_in,
    input logic [7:0] CART_DATA_out,
    input logic CART_RD,
    input logic CART_WR,
    output logic [25:0] MBC1_ADDR,
)
output logic MBC1_RD,
output logic MBC1_WR,
input logic [7:0] MBC1_DATA_in,
output logic [7:0] MBC1_DATA_out
);

// 4 writable registers
logic [6:0] BANK_NUM, BANK_NUM_NEXT; // {BANK2(2-bit), BANK1(5-bit)}
logic [6:0] BANK_NUM_ACTUAL; // 0x4000-0x5FFF 0x2000-0x3FFF
logic RAM_ROM_MODE, RAM_ROM_MODE_NEXT; // 0x6000-0x7FFF
logic RAM_EN, RAM_EN_NEXT; // 0x0000-0x1FFF

always_ff @(posedge clk)
begin
    if (reset)
        begin
            BANK_NUM <= 0;
            RAM_ROM_MODE <= 0;
            RAM_EN <= 0;
        end
    else
        begin
            BANK_NUM <= BANK_NUM_NEXT;
            RAM_ROM_MODE <= RAM_ROM_MODE_NEXT;
            RAM_EN <= RAM_EN_NEXT;
        end
end

always_comb
begin
    BANK_NUM_NEXT = BANK_NUM;
    RAM_ROM_MODE_NEXT = RAM_ROM_MODE;
    RAM_EN_NEXT = RAM_EN;
MBC1_ADDR = 0;
MBC1_RD = 0;
MBC1_WR = 0;
BANK_NUM_ACTUAL = BANK_NUM;
if (BANK_NUM_ACTUAL == 8'h00 || BANK_NUM_ACTUAL == 8'h20 ||
    BANK_NUM_ACTUAL == 8'h40 || BANK_NUM_ACTUAL == 8'h60)
begin
    BANK_NUM_ACTUAL = BANK_NUM_ACTUAL + 1;
end
BANK_NUM_ACTUAL = BANK_NUM_ACTUAL % NUM_ROM_BANK;

CART_DATA_in = MBC1_DATA_in;
MBC1_DATA_out = CART_DATA_out;

if (CART_ADDR < 16'h4000 && CART_RD) // ROM Bank 0 (READ ONLY)
begin
    MBC1_ADDR = {10'b0, CART_ADDR};
    // RAM Banking
    if (RAM_ROM_MODE)
begin
        MBC1_ADDR = {10'b0, CART_ADDR} + ((BANK_NUM_ACTUAL[6:5]) << 19);
    end
    MBC1_RD = CART_RD;
end
else if (CART_ADDR < 16'h8000 && CART_RD) // ROM Bank N (READ ONLY)
begin
    MBC1_ADDR = {10'b0, CART_ADDR} + (BANK_NUM_ACTUAL << 14) - 26'h4000;
    MBC1_RD = CART_RD;
end
else if (CART_ADDR >= 16'hA000 && CART_ADDR < 16'hC000) // RAM Bank N
(READ/WRITE)
begin
    if (RAM_EN)
        begin
            MBC1_ADDR = `SDRAM_RAM_BASE + {10'b0, CART_ADDR} - 26'hA000 +
                (RAM_ROM_MODE ? (BANK_NUM[6:5] % NUM_RAM_BANK) << 13 : 0);
            MBC1_RD = CART_RD;
            MBC1_WR = CART_WR;
        end
    else CART_DATA_in = 8'hFF;
end
else if (CART_ADDR < 16'h2000 && CART_WR) // RAM enable (WRITE ONLY)
    begin
        if (CART_DATA_out[3:0] == 4'hA) RAM_EN_NEXT = 1;
        else RAM_EN_NEXT = 0;
    end
else if (CART_ADDR >= 16'h2000 && CART_ADDR < 16'h4000 && CART_WR) // Bank1 (WRITE ONLY)
    begin
        BANK_NUM_NEXT = {BANK_NUM[6:5], CART_DATA_out[4:0]};
    end
else if (CART_ADDR >= 16'h4000 && CART_ADDR < 16'h6000 && CART_WR) // Bank2 (WRITE ONLY)
    begin
        BANK_NUM_NEXT = {CART_DATA_out[1:0], BANK_NUM[4:0]};
    end
else if (CART_ADDR >= 16'h6000 && CART_ADDR < 16'h8000 && CART_WR) // Mode (WRITE ONLY)
    begin
        RAM_ROM_MODE_NEXT = CART_DATA_out[0];
    end
end
//timescale 1ns / 1ps

//

// This is the MBC 1 Memory Bank Controller for The GameBoy

//

`define SDRAM_RAM_BASE 26'h2000000

module MBC5

(
  input logic clk,
  input logic reset,
  input logic [9:0] NUM_ROM_BANK, // How many ROM banks in this cartridge?
  input logic [4:0] NUM_RAM_BANK, // How many RAM banks in this cartridge?
  input logic [15:0] CART_ADDR,
  output logic [7:0] CART_DATA_in,
  input logic [7:0] CART_DATA_out,
  input logic CART_RD,
  input logic CART_WR,
  output logic [25:0] MBC5_ADDR,
  output logic MBC5_RD,
  output logic MBC5_WR,
  input logic [7:0] MBC5_DATA_in,
  output logic [7:0] MBC5_DATA_out
);

endmodule
// 4 writable registers
logic [8:0] ROM_BANK_NUM, ROM_BANK_NUM_NEXT; // {ROMB1(1-bit), ROMBO(8-bit)}
logic [8:0] ROM_BANK_NUM_ACTUAL; // 0x3000-0x3FFF 0x2000-0x2FFF
logic [3:0] RAM_BANK_NUM, RAM_BANK_NUM_NEXT; // 0x4000-0x5FFF
logic RAM_EN, RAM_EN_NEXT; // 0x0000-0x1FFF

always_ff @(posedge clk)
begin
  if (reset)
    begin
      ROM_BANK_NUM <= 1;
      RAM_BANK_NUM <= 0;
      RAM_EN <= 0;
    end
  else
    begin
      ROM_BANK_NUM <= ROM_BANK_NUM_NEXT;
      RAM_BANK_NUM <= RAM_BANK_NUM_NEXT;
      RAM_EN <= RAM_EN_NEXT;
    end
end

always_comb
begin
  ROM_BANK_NUM_NEXT = ROM_BANK_NUM;
  RAM_EN_NEXT = RAM_EN;
  RAM_BANK_NUM_NEXT = RAM_BANK_NUM;
  MBC5_ADDR = 0;
  MBC5_RD = 0;
  MBC5_WR = 0;
ROM_BANK_NUM_ACTUAL = ROM_BANK_NUM % NUM_ROM_BANK;

CART_DATA_in = MBC5_DATA_in;
MBC5_DATA_out = CART_DATA_out;

if (CART_ADDR < 16'h4000 && CART_RD) // ROM Bank 0 (READ ONLY)
begin
    MBC5_ADDR = {10'b0, CART_ADDR};
    MBC5_RD = CART_RD;
end
else if (CART_ADDR < 16'h8000 && CART_RD) // ROM Bank N (READ ONLY)
begin
    MBC5_ADDR = {10'b0, CART_ADDR} + (ROM_BANK_NUM_ACTUAL << 14) - 26'h4000;
    MBC5_RD = CART_RD;
end
else if (CART_ADDR >= 16'hA000 && CART_ADDR < 16'hC000) // RAM Bank N (READ/WRITE)
begin
    if (RAM_EN)
begin
        MBC5_ADDR = `SDRAM_RAM_BASE + {10'b0, CART_ADDR} - 26'hA000 +
((RAM_BANK_NUM % NUM_RAM_BANK) << 13);
        MBC5_RD = CART_RD;
        MBC5_WR = CART_WR;
    end
    else CART_DATA_in = 8'hFF;
end
else if (CART_ADDR < 16'h2000 && CART_WR) // RAM enable (WRITE ONLY)
begin
    if (CART_DATA_out[3:0] == 4'hA) RAM_EN_NEXT = 1;
else RAM_EN_NEXT = 0;
end

else if (CART_ADDR >= 16'h2000 && CART_ADDR < 16'h3000 && CART_WR) //
ROMBO (WRITE ONLY)
begin
    ROM_BANK_NUM_NEXT = {ROM_BANK_NUM[8], CART_DATA_out[7:0]};
end

else if (CART_ADDR >= 16'h3000 && CART_ADDR < 16'h4000 && CART_WR) //
ROMB1 (WRITE ONLY)
begin
    ROM_BANK_NUM_NEXT = {CART_DATA_out[0], ROM_BANK_NUM[7:0]};
end

else if (CART_ADDR >= 16'h4000 && CART_ADDR < 16'h6000 && CART_WR) //
RAM Bank (WRITE ONLY)
begin
    RAM_BANK_NUM_NEXT = CART_DATA_out[3:0];
end

end

endmodule

Listing C.11: MBC3.sv

`timescale 1ns / 1ps

//

/////////\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\"
(  
  input logic clk,  
  input logic reset,  
  input logic [9:0] NUM_ROM_BANK, // How many ROM banks in this  
  cartridge?  
  input logic [4:0] NUM_RAM_BANK, // How many RAM banks in this  
  cartridge?  
  input logic [15:0] CART_ADDR,  
  output logic [7:0] CART_DATA_in,  
  input logic [7:0] CART_DATA_out,  
  input logic CART_RD,  
  input logic CART_WR,  
  output logic [25:0] MBC5_ADDR,  
  output logic MBC5_RD,  
  output logic MBC5_WR,  
  input logic [7:0] MBC5_DATA_in,  
  output logic [7:0] MBC5_DATA_out  
);

// 4 writable registers
logic [8:0] ROM_BANK_NUM, ROM_BANK_NUM_NEXT; // {ROMB1(1-bit), ROMB0(8-bit)}
logic [8:0] ROM_BANK_NUM_ACTUAL; // 0x3000-0x3FFF 0x2000-0x2FFF
logic [3:0] RAM_BANK_NUM, RAM_BANK_NUM_NEXT; // 0x4000-0x5FFF
logic RAM_EN, RAM_EN_NEXT; // 0x0000-0x1FFF

always_ff @(posedge clk)
begin
  if (reset)
    begin
      ROM_BANK_NUM <= 1;
      RAM_BANK_NUM <= 0;
always_comb begin
    ROM_BANK_NUM_NEXT = ROM_BANK_NUM;
    RAM_EN_NEXT = RAM_EN;
    RAM_BANK_NUM_NEXT = RAM_BANK_NUM;
    MBC5_ADDR = 0;
    MBC5_RD = 0;
    MBC5_WR = 0;

    ROM_BANK_NUM_ACTUAL = ROM_BANK_NUM \% NUM_ROM_BANK;

    CART_DATA_in = MBC5_DATA_in;
    MBC5_DATA_out = CART_DATA_out;

    if (CART_ADDR < 16'h4000 && CART_RD) // ROM Bank 0 (READ ONLY)
    begin
        MBC5_ADDR = {10'b0, CART_ADDR};
        MBC5_RD = CART_RD;
    end
    else if (CART_ADDR < 16'h8000 && CART_RD) // ROM Bank N (READ ONLY)
    begin
        MBC5_ADDR = {10'b0, CART_ADDR} + (ROM_BANK_NUM_ACTUAL \<< 14) - 26'h4000;
    end
MBC5_RD = CART_RD;

else if (CART_ADDR >= 16’hA000 && CART_ADDR < 16’hC000) // RAM Bank N (READ/WRITE)
begin
    if (RAM_EN)
begin
        MBC5_ADDR = 'SDRAM_RAM_BASE + {10'b0, CART_ADDR} - 26’hA000 +
((RAM_BANK_NUM % NUM_RAM_BANK) << 13);
        MBC5_RD = CART_RD;
        MBC5_WR = CART_WR;
    end
    else CART_DATA_in = 8’hFF;
end
else if (CART_ADDR < 16’h2000 && CART_WR) // RAM enable (WRITE ONLY)
begin
    if (CART_DATA_out[3:0] == 4’hA) RAM_EN_NEXT = 1;
    else RAM_EN_NEXT = 0;
end
else if (CART_ADDR >= 16’h2000 && CART_ADDR < 16’h3000 && CART_WR) // ROMB0 (WRITE ONLY)
begin
    ROM_BANK_NUM_NEXT = {ROM_BANK_NUM[8], CART_DATA_out[7:0]};
end
else if (CART_ADDR >= 16’h3000 && CART_ADDR < 16’h4000 && CART_WR) // ROMB1 (WRITE ONLY)
begin
    ROM_BANK_NUM_NEXT = {CART_DATA_out[0], ROM_BANK_NUM[7:0]};
end
else if (CART_ADDR >= 16’h4000 && CART_ADDR < 16’h6000 && CART_WR) // RAM Bank (WRITE ONLY)
begin
    RAM_BANK_NUM_NEXT = CART_DATA_out[3:0];
module GameBoy_Top(
    input logic clk,
    input logic rst,
    /* GameBoy Pixel Conduit */
    output logic PX_VALID,
    output logic [1:0] LD,
    /* GameBoy Joypad Conduit */
    input logic P10,
    input logic P11,
    input logic P12,
    input logic P13,
    output logic P14,
    output logic P15,
    /* GameBoy Cartridge Conduit */
    output logic [15:0] CART_ADDR,
    input logic [7:0] CART_DATA_in,

Listing C.12: MBC3.sv
```
output logic [7:0] CART_DATA_out,
output logic CART_RD,
output logic CART_WR,
/* GameBoy Audio Conduit */
output logic [15:0] LOUT,
output logic [15:0] ROUT
);

/* Video SRAM */
logic [7:0] MD_in; // video sram data
logic [7:0] MD_out; // video sram data
logic [12:0] MA;
logic MWR; // high active
logic MCS; // high active
logic MOE; // high active
/* LCD */
logic CPG; // CONTROL
logic CP; // CLOCK
logic ST; // HORSYNC
logic CPL; // DATALCH
logic FR; // ALTSIGL
logic S; // VERTSYN

/* Serial Link */
logic S_OUT;
logic S_IN;
logic SCK_in; // serial link clk
logic SCK_out; // serial link clk
/* Work RAM/Cartridge */
logic CLK_GC; // Game Cartridge Clock
logic WR; // high active
logic RD; // high active
```
logic CS; // high active
logic [15:0] A;
logic [7:0] D_in; // data bus
logic [7:0] D_out; // data bus

/* The DMG-CPU */
LR35902 DMG_CPU (.clk(clk), .rst(rst), .MD_in(MD_in), .MD_out(MD_out), .MA(MA), .MWR(MWR), .MCS(MCS), .MOE(MOE), .LD(LD), .PX_VALID(PX_VALID),
    .S_OUT(S_OUT), .S_IN(S_IN), .SCK_in(SCK_in), .SCK_out(SCK_out), .CLK_GC(CLK_GC), .WR(WR), .RD(RD), .CS(CS), .A(A), .D_in(D_in),
    .D_out(D_out), .LOUT(LOUT), .ROUT(ROUT));

/* VRAM Connection */
LH5264 VRAM (.D_out(MD_in), .D_in(MD_out), .CE1(MCS), .CE2(MWR), .A(MA), .OE(MOE), .clk(~clk));

/* WRAM Connection */
logic [7:0] WRAM_Din, WRAM_Dout, WRAM_WR;
LH5264 WRAM (.D_out(WRAM_Din), .D_in(WRAM_Dout), .CE1(WRAM_WR), .CE2(A[14]),
    .A(A), .OE(A[14]), .clk(~clk));

/* Cartridge */
assign D_in = (A < 16'h8000 || (A >= 16'hA000 && A < 16'hC000)) ?
    CART_DATA_in : WRAM_Din;
assign CART_DATA_out = (A < 16'h8000 || (A >= 16'hA000 && A < 16'hC000)) ?
    D_out : 0;
assign CART_RD = RD && (A < 16'h8000 || (A >= 16'hA000 && A < 16'hC000));
assign CART_WR = WR && (A < 16'h8000 || (A >= 16'hA000 && A < 16'hC000));
assign CART_ADDR = (A < 16'h8000 || (A >= 16'hA000 && A < 16'hC000)) ? A :
    0;
assign WRAM_Dout = !(A < 16'h8000 || (A >= 16'hA000 && A < 16'hC000)) ? D_out : 8'hFF;
assign WRAM_WR = WR && (A >= 16'hC000 && A < 16'hFE00);
endmodule
output logic [7:0] VGA_R, VGA_G, VGA_B,
output logic   VGA_CLK, VGA_HS, VGA_VS,

output logic   VGA_BLANK_n,

output logic   VGA_SYNC_n,

/* GameBoy Pixel Conduit */
input logic [1:0] LD,
input logic PX_VALID

);

// VGA signals
logic [15:0] LX;
logic [15:0] LY;

logic [7:0]   bg_r, bg_g, bg_b;

logic [1:0]   GB_PIXEL;

// Instantiations
vga_counters counters(.*);

/* The Framebuffer for gameboy */
logic [14:0] frame_buffer_cnt;
logic frame_buffer_switch;

always_ff @(posedge GameBoy_clk or posedge GameBoy_reset)
begin
    if (GameBoy_reset)
        begin
            frame_buffer_cnt <= 0;
            frame_buffer_switch <= 0;
        end
else if (PX_VALID)
begin
  if (frame_buffer_cnt == 23039)
  begin
    frame_buffer_cnt <= 0;
  end
  else frame_buffer_cnt <= frame_buffer_cnt + 1;
end

logic [15:0] READ_LX, READ_LY;

assign READ_LX = LX > 160 ? LX - 160 : 0;
assign READ_LY = LY > 80 ? LY - 80 : 0;

logic [7:0] GB_LX, GB_LY;
logic [2:0] GB_COL_CNT, GB_ROW_CNT;
always_ff @(posedge clk_vga)
begin
  if (LX < 160 || LX >= 1120)
  begin
    GB_LX <= 0;
    GB_COL_CNT <= 0;
  end
  else
  begin
    GB_COL_CNT <= GB_COL_CNT + 1;
  end
  if (GB_COL_CNT == 5)
  begin
    GB_COL_CNT <= 0;
    GB_LX <= GB_LX + 1;
  end
if (LY <= 80 || LY >= 944)
begin
    GB_LY <= 0;
    GB_ROW_CNT <= 0;
end
else if (LX == 1)
begin
    GB_ROW_CNT <= GB_ROW_CNT + 1;
end
if (GB_ROW_CNT == 6)
begin
    GB_ROW_CNT <= 0;
    GB_LY <= GB_LY + 1;
end

Quartus_dual_port_dual_clk_ram_23040 LCD_FRAME_BUFFER0 (.write_clk (~GameBoy_clk), .read_clk (~clk_vga), .data(LD), .we(PX_VALID), .write_addr(frame_buffer_cnt), .read_addr({7'b0, GB_LX} + {2'b0, GB_LY, 5'b0} + {GB_LY, 7'b0}), .q(GB_PIXEL));

always_ff @(posedge clk)
begin
    if (reset)
    begin
        bg_r <= 8'd192;
        bg_g <= 8'd156;
        bg_b <= 8'd14;
    end
end
else if (chipselect && write)
begin
    bg_r <= 8'h80;
end

always_comb
begin
{VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
if (VGA_BLANK_n)
begin
    if (LX >= 160 && LX <= 1120 && LY >= 80 && LY <= 944)
        begin
            unique case (GB_PIXEL)
                2'b11:
                    begin
                        VGA_R = 51;
                        VGA_G = 44;
                        VGA_B = 80;
                    end
                2'b10:
                    begin
                        VGA_R = 70;
                        VGA_G = 135;
                        VGA_B = 143;
                    end
        end
end

//; Palette Name: Kirokaze Gameboy
//; Colors: 4
//FF332c50
//FF46878f
//FF94e344
//FFe2f3e4
2'b01:
    begin
        VGA_R = 148;
        VGA_G = 227;
        VGA_B = 68;
    end
2'b00:
    begin
        VGA_R = 226;
        VGA_G = 243;
        VGA_B = 228;
    end
endcase

// Retro
if (GB_ROW_CNT == 0 || GB_COL_CNT == 0)
    begin
        VGA_R = 51;
        VGA_G = 44;
        VGA_B = 80;
    end
end
else {VGA_R, VGA_G, VGA_B} = {bg_r, bg_g, bg_b};
end
endmodule

module vga_counters
(
    input logic clk_vga, reset,
    output logic [15:0] LX,
    output logic [15:0] LY,
    output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
logic [15:0] hcount, vcount, hcount_next, vcount_next;

  /*
   * 1280 X 1024 VGA timing for a 108 MHz clock: one pixel every cycle
   *
   * HCOUNT 1687 0       1279       1687 0
   * _________________ | __________ |
   * ________|  Video    |___________|  Video
   *
   *
   * |SYNC| BP |--->  HACTIVE -->|FP|HACTIVESYNC| BP |<-- HACTIVE
   * _________________ | __________
   * |___|      VGA_HS     |___|
   */

  // Parameters for hcount
  parameter HACTIVE    = 1280,
              HFRONT_PORCH = 48,
              HSYNC       = 112,
              HBACK_PORCH = 248,
              HTOTAL      = HACTIVE + HFRONT_PORCH + HSYNC +
                              HBACK_PORCH;  // 1688

  // Parameters for vcount
  parameter VACTIVE    = 1024,
              VFRONT_PORCH = 1,
              VSYNC       = 3,
              VBACK_PORCH = 38,
              VTOTAL      = VACTIVE + VFRONT_PORCH + VSYNC +
                              VBACK_PORCH;  // 1066

logic endOfLine;
assign endOfLine = hcount == HTOTAL - 1;

logic endOfField;
assign endOfField = vcount == VTOTAL - 1;

always_ff @(posedge clk_vga or posedge reset)
begin
  if (reset)
    begin
      hcount <= 0;
      vcount <= 0;
    end
  else
    begin
      hcount <= hcount_next;
      vcount <= vcount_next;
    end
  end

always_comb
begin
  hcount_next = hcount + 1;
  vcount_next = vcount;
  if (endOfLine)
    begin
      hcount_next = 0;
      vcount_next = vcount + 1;
    end
  if (endOfField) vcount_next = 0;
end
assign VGA_HS = !(hcount >= HACTIVE + HFRONT_PORCH) && (hcount <
HACTIVE + HFRONT_PORCH + HSYNC));
assign VGA_VS = !(vcount >= VACTIVE + VFRONT_PORCH) && (vcount <
VACTIVE + VFRONT_PORCH + VSYNC));
assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal;
unused
assign VGA_BLANK_n = (hcount < HACTIVE) && (vcount < VACTIVE);
assign VGA_CLK = clk_vga; // 108 MHz clock: rising edge sensitive
assign LX = hcount_next;
assign LY = vcount_next;
endmodule

Listing C.14: GameBoy_VGA.sv

`timescale 1ns / 1ps

module GameBoy_Audio
(
  input logic clk,
  input logic rst,

  input logic [15:0] GB_LOUT,
  input logic [15:0] GB_ROUT,

  output logic [15:0] right_data,
  output logic right_valid,
  input logic right_ready,
  output logic [15:0] left_data,
  output logic left_valid,
input logic left_ready;

logic [15:0] counter;
logic [6:0] init_counter;

always_ff @(posedge clk)
begin
    if (rst)
        begin
            counter <= 0;
            init_counter <= 0;
        end
    else
        begin
            if (init_counter != 7'b111_1111)
                init_counter <= init_counter + 1;
            else
                begin
                    counter <= counter + 1;
                end
        end
end

always_comb
begin
    right_valid = 0;
    left_valid = 0;
    right_data = 0;
    left_data = 0;
    if (init_counter != 7'b111_1111)
        begin
            right_data = 16'h0000;
        end
end
left_data = 16'h0000;
right_valid = 1;
left_valid = 1;

end
else
begin
if (counter[7:0] == 8'hFF)
begin
    right_valid = 1;
    left_valid = 1;
    right_data = GB_ROUT << 6;
    left_data = GB_LOUT << 6;
end
end
end

endmodule

Listing C.15: GameBoy_Audio.sv

module GameBoy_Joypad
(
    input logic clk,
    input logic reset,
    /* Avalon Slave */
    input logic [7:0] writedata_slv,
    input logic write_slv,
    input chipselect_slv,
    /* Gameboy JoyPad Conduit */
    input logic P15,
    input logic P14,
    output logic P13,
    output logic P12,
    output logic P11,
output logic P10
);

logic [7:0] joypad;

always_ff @(posedge clk)
begin
    if (reset)
        begin
            joypad <= 8'h00;
        end
    else if (chipselect_slv && write_slv)
        begin
            joypad <= writedata_slv;
        end
end

always_comb
begin
    P10 = 1;
P11 = 1;
P12 = 1;
P13 = 1;
    if (!P14)
        begin
            if (joypad[0]) // RIGHT
                P10 = 0;
            if (joypad[1]) // LEFT
                P11 = 0;
            if (joypad[2]) // UP
                P12 = 0;
            if (joypad[3]) // DOWN
                P13 = 0;
        end
end
```vhdl
end
if (!P15)
begin
  if (joypad[4]) // A
    P10 = 0;
  if (joypad[5]) // B
    P11 = 0;
  if (joypad[6]) // SELECT
    P12 = 0;
  if (joypad[7]) // START
    P13 = 0;
end
endmodule
```

Listing C.16: GameBoy_Joypad.sv

```
// ===================================================================
// Copyright (c) 2013 by Terasic Technologies Inc.
// ===================================================================

// Modified 2019 by Stephen A. Edwards

// Permission:

// Terasic grants permission to use and modify this code for use in
// synthesis for all Terasic Development Boards and Altera
// Development Kits made by Terasic. Other use of this code,
// including the selling, duplication, or modification of any
// portion is strictly prohibited.

// Disclaimer:
```
This VHDL/Verilog or C/C++ source code is intended as a design reference which illustrates how these types of functions can be implemented. It is the user's responsibility to verify their design for consistency and functionality through the use of formal verification methods. Terasic provides no warranty regarding the use or functionality of this code.

Terasic Technologies Inc

9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070, Taiwan

web: http://www.terasic.com/
email: support@terasic.com

module soc_system_top(

    inout ADC_CS_N,
    output ADC_DIN,
    input ADC_DOUT,
    output ADC_SCLK,

    input AUD_ADCDAT,
    inout AUD_ADCLRCK,
    inout AUD_BCLK,
    output AUD_DACDAT,
    inout AUD_DACLRCK,
    output AUD_XCK,

);
CLOCK2
input CLOCK2_50,

CLOCK3
input CLOCK3_50,

CLOCK4
input CLOCK4_50,

CLOCK
input CLOCK_50,

DRAM
output [12:0] DRAM_ADDR,
output [1:0] DRAM_BA,
output DRAM_CAS_N,
output DRAM_CKE,
output DRAM_CLK,
output DRAM_CS_N,
inout [15:0] DRAM_DQ,
output DRAM_LDQM,
output DRAM_RAS_N,
output DRAM_UDQM,
output DRAM_WE_N,

FAN
output FAN_CTRL,

FPGA
output FPGA_I2C_SCLK,
inout FPGA_I2C_SDAT,

GPIO
inout [35:0] GPIO_0,
inout [35:0] GPIO_1,

/////////// HEX0 ///////////
output [6:0] HEX0,

/////////// HEX1 ///////////
output [6:0] HEX1,

/////////// HEX2 ///////////
output [6:0] HEX2,

/////////// HEX3 ///////////
output [6:0] HEX3,

/////////// HEX4 ///////////
output [6:0] HEX4,

/////////// HEX5 ///////////
output [6:0] HEX5,

/////////// HPS ///////////
inout HPS_CONV_USB_N,
output [14:0] HPS_DDR3_ADDR,
output [2:0] HPS_DDR3_BA,
output HPS_DDR3_CAS_N,
output HPS_DDR3_CKE,
output HPS_DDR3_CK_N,
output HPS_DDR3_CK_P,
output HPS_DDR3_CS_N,
output [3:0] HPS_DDR3_DM,
inout [31:0] HPS_DDR3_DQ,
inout [3:0] HPS_DDR3_DQS_N,
inout [3:0] HPS_DDR3_DQS_P,
output HPS_DDR3_ODT,
output HPS_DDR3_RAS_N,
output HPS_DDR3_RESET_N,
input HPS_DDR3_RZQ,
output HPS_DDR3_WE_N,
output HPS_ENET_GTX_CLK,
inout HPS_ENET_INT_N,
output HPS_ENET_MDC,
inout HPS_ENET_MDIO,
input HPS_ENET_RX_CLK,
inout [3:0] HPS_ENET_RX_DATA,
inout HPS_ENET_RX_DV,
output [3:0] HPS_ENET_TX_DATA,
output HPS_ENET_TX_EN,
inout HPS_GSENSOR_INT,
inout HPS_I2C1_SCLK,
inout HPS_I2C1_SDAT,
inout HPS_I2C2_SCLK,
inout HPS_I2C2_SDAT,
inout HPS_I2C_CONTROL,
inout HPS_KEY,
inout HPS_LED,
inout HPS_LTC_GPIO,
output HPS_SD_CLK,
inout HPS_SD_CMD,
inout [3:0] HPS_SD_DATA,
output HPS_SPIM_CLK,
input HPS_SPIM_MISO,
output HPS_SPIM_MOSI,
inout HPS_SPIM_SS,
inout HPS_UART_RX,
output HPS_UART_TX,
input HPS_USB_CLKOUT,
inout [7:0] HPS_USB_DATA,
input HPS_USB_DIR,
input HPS_USB_NXT,
output HPS_USB_STP,

// IRDA //////////
input IRDA_RXD,
output IRDA_TXD,

// KEY ///////////
input [3:0] KEY,

// LEDR ///////////
output [9:0] LEDR,

// PS2 ///////////
inout PS2_CLK,
inout PS2_CLK2,
inout PS2_DAT,
inout PS2_DAT2,

// SW ///////////
input [9:0] SW,

// TD ///////////
input TD_CLK27,
inout [7:0] TD_DATA,
input TD_HS,
output TD_RESET_N,
input TD_VS,
///// VGA /////

output [7:0] VGA_B,
output VGA_BLANK_N,
output VGA_CLK,
output [7:0] VGA_G,
output VGA_HS,
output [7:0] VGA_R,
output VGA_SYNC_N,
output VGA_VS
);

soc_system soc_system0(
    .clk_clk ( CLOCK_50 ),
    .reset_reset ( (~KEY[0] && ~KEY[1]) ),
    .hps_ddr3_mem_a ( HPS_DDR3_ADDR ),
    .hps_ddr3_mem_ba ( HPS_DDR3_BA ),
    .hps_ddr3_mem_ck ( HPS_DDR3_CK_P ),
    .hps_ddr3_mem_ck_n ( HPS_DDR3_CK_N ),
    .hps_ddr3_mem_cke ( HPS_DDR3_CKE ),
    .hps_ddr3_mem_cs_n ( HPS_DDR3_CS_N ),
    .hps_ddr3_mem_ras_n ( HPS_DDR3_RAS_N ),
    .hps_ddr3_mem_cas_n ( HPS_DDR3_CAS_N ),
    .hps_ddr3_mem_we_n ( HPS_DDR3_WE_N ),
    .hps_ddr3_mem_reset_n ( HPS_DDR3_RESET_N ),
    .hps_ddr3_mem_dq ( HPS_DDR3_DQ ),
    .hps_ddr3_mem_dqs ( HPS_DDR3_DQS_P ),
    .hps_ddr3_mem_dqs_n ( HPS_DDR3_DQS_N ),
    .hps_ddr3_mem_odt ( HPS_DDR3_ODT ),
    .hps_ddr3_mem_dm ( HPS_DDR3_DM ),
    .hps_ddr3_oct_rzqin ( HPS_DDR3_RZQ ),
    .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
);
.hps_hps_io_emac1_inst_TXD0 ( HPS_ENET_TX_DATA[0] ),
.hps_hps_io_emac1_inst_TXD1 ( HPS_ENET_TX_DATA[1] ),
.hps_hps_io_emac1_inst_TXD2 ( HPS_ENET_TX_DATA[2] ),
.hps_hps_io_emac1_inst_TXD3 ( HPS_ENET_TX_DATA[3] ),
.hps_hps_io_emac1_inst_RXD0 ( HPS_ENET_RX_DATA[0] ),
.hps_hps_io_emac1_inst_RXD1 ( HPS_ENET_RX_DATA[1] ),
.hps_hps_io_emac1_inst_RXD2 ( HPS_ENET_RX_DATA[2] ),
.hps_hps_io_emac1_inst_RXD3 ( HPS_ENET_RX_DATA[3] ),
.hps_hps_io_emac1_inst_MDIO ( HPS_ENET_MDIO ),
.hps_hps_io_emac1_inst_MDC ( HPS_ENET_MDC ),
.hps_hps_io_emac1_inst_RX_CTL ( HPS_ENET_RX_DV ),
.hps_hps_io_emac1_inst_TX_CTL ( HPS_ENET_TX_EN ),
.hps_hps_io_emac1_inst_RX_CLK ( HPS_ENET_RX_CLK ),
.hps_hps_io_emac1_inst_RXD1 ( HPS_ENET_RX_DATA[1] ),
.hps_hps_io_emac1_inst_RXD2 ( HPS_ENET_RX_DATA[2] ),
.hps_hps_io_emac1_inst_RXD3 ( HPS_ENET_RX_DATA[3] ),
.hps_hps_io_sdio_inst_CMD ( HPS_SD_CMD ),
.hps_hps_io_sdio_inst_D0 ( HPS_SD_DATA[0] ),
.hps_hps_io_sdio_inst_D1 ( HPS_SD_DATA[1] ),
.hps_hps_io_sdio_inst_CLK ( HPS_SD_CLK ),
.hps_hps_io_sdio_inst_D2 ( HPS_SD_DATA[2] ),
.hps_hps_io_sdio_inst_D3 ( HPS_SD_DATA[3] ),
.hps_hps_io_usb1_inst_D0 ( HPS_USB_DATA[0] ),
.hps_hps_io_usb1_inst_D1 ( HPS_USB_DATA[1] ),
.hps_hps_io_usb1_inst_D2 ( HPS_USB_DATA[2] ),
.hps_hps_io_usb1_inst_D3 ( HPS_USB_DATA[3] ),
.hps_hps_io_usb1_inst_D4 ( HPS_USB_DATA[4] ),
.hps_hps_io_usb1_inst_D5 ( HPS_USB_DATA[5] ),
.hps_hps_io_usb1_inst_D6 ( HPS_USB_DATA[6] ),
.hps_hps_io_usb1_inst_D7 ( HPS_USB_DATA[7] ),
.hps_hps_io_usb1_inst_CLK ( HPS_USB_CLKOUT ),
.hps_hps_io_usb1_inst_STP ( HPS_USB_STP ),
.hps_hps_io_usb1_inst_DIR ( HPS_USB_DIR ),
.hps_hps_io_usb1_inst_NXT ( HPS_USB_NXT ),
.hps_hps_io_spim1_inst_CLK (HPS_SPIM_CLK),
.hps_hps_io_spim1_inst_MOSI (HPS_SPIM_MOSI),
.hps_hps_io_spim1_inst_MISO (HPS_SPIM_MISO),
.hps_hps_io_spim1_inst_SS0 (HPS_SPIM_SS),

.hps_hps_io_uart0_inst_RX (HPS_UART_RX),
.hps_hps_io_uart0_inst_TX (HPS_UART_TX),

.hps_hps_io_i2c0_inst_SDA (HPS_I2C1_SDAT),
.hps_hps_io_i2c0_inst_SCL (HPS_I2C1_SCLK),

.hps_hps_io_i2c1_inst_SDA (HPS_I2C2_SDAT),
.hps_hps_io_i2c1_inst_SCL (HPS_I2C2_SCLK),

.hps_hps_io_gpio_inst_GPIO09 (HPS_CONV_USB_N),
.hps_hps_io_gpio_inst_GPIO35 (HPS_ENET_INT_N),
.hps_hps_io_gpio_inst_GPIO40 (HPS_LTC_GPIO),

.hps_hps_io_gpio_inst_GPIO48 (HPS_I2C_CONTROL),
.hps_hps_io_gpio_inst_GPIO53 (HPS_LED),
.hps_hps_io_gpio_inst_GPIO54 (HPS_KEY),
.hps_hps_io_gpio_inst_GPIO61 (HPS_GSENSOR_INT),

/* VGA Conduit */
.vga_vga_r (VGA_R),
.vga_vga_g (VGA_G),
.vga_vga_b (VGA_B),
.vga_vga_clk (VGA_CLK),
.vga_vga_hs (VGA_HS),
.vga_vga_vs (VGA_VS),
.vga_vga_blank_n (VGA_BLANK_N),
.vga_vga_sync_n (VGA_SYNC_N),
/* SDRAM Conduit */
.sdram_addr(DRAM_ADDR),
.sdram_ba(DRAM_BA),
.sdram_cas_n(DRAM_CAS_N),
.sdram_cke(DRAM_CKE),
.sdram_cs_n(DRAM_CS_N),
.sdram_dq(DRAM_DQ),
.sdram_dqm({DRAM_UDQM, DRAM_LDQM}),
.sdram_ras_n(DRAM_RAS_N),
.sdram_we_n(DRAM_WE_N),
.sdram_clk_clk(DRAM_CLK),
.gameboy_reset_reset(~KEY[2] & ~KEY[3]),

/* red led */
.ledr_ledr(LEDR),

/* HEX display */
//.hex_display_hex0(HEX0),
//.hex_display_hex1(HEX1),
//.hex_display_hex2(HEX2),
//.hex_display_hex3(HEX3),
//.hex_display_hex4(HEX4),
//.hex_display_hex5(HEX5),

/* Button */
//.button_key(KEY)

/* audio */
.audio_clk_clk(AUD_XCK),
.audio_BCLK(AUD_BCLK),
.audio_DACDAT(AUD_DACDAT),
.audio_DACLRCK(AUD_DACLRCK),
.av_config_SDAT(FPGA_I2C_SDAT),
.av_config_SCLK(FPGA_I2C_SCLK),
.reset_audio_reset(~KEY[0] && ~KEY[1])

);

// The following quiet the "no driver" warnings for output
// pins and should be removed if you use any of these peripherals

assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
assign ADC_DIN = SW[0];
assign ADC_SCLK = SW[0];

// assign AUD_ADCLRCK = SW[1] ? SW[0] : 1'bZ;
// assign AUD_BCLK = SW[1] ? SW[0] : 1'bZ;
// assign AUD_DACDAT = SW[0];
// assign AUD_DACLRCK = SW[1] ? SW[0] : 1'bZ;
// assign AUD_XCK = SW[0];

assign FAN_CTRL = SW[0];

//assign FPGA_I2C_SCLK = SW[0];
//assign FPGA_I2C_SDAT = SW[1] ? SW[0] : 1'bZ;

assign GPIO_0 = SW[1] ? { 36{ SW[0] } } : 36'bZ;
assign GPIO_1 = SW[1] ? { 36{ SW[0] } } : 36'bZ;

//assign HEX0 = { 7{ SW[1] } };
//assign HEX1 = { 7{ SW[2] } };
//assign HEX2 = { 7{ SW[3] } };
//assign HEX3 = { 7{ SW[4] } };
//assign HEX4 = { 7{ SW[5] } };
//assign HEX5 = { 7{ SW[6] } };
assign IRDA_TXD = SW[0];

//assign LEDR = { 10{SW[7]} };

assign PS2_CLK = SW[1] ? SW[0] : 1'bZ;
assign PS2_CLK2 = SW[1] ? SW[0] : 1'bZ;
assign PS2_DAT = SW[1] ? SW[0] : 1'bZ;
assign PS2_DAT2 = SW[1] ? SW[0] : 1'bZ;

assign TD_RESET_N = SW[0];

// assign {VGA_R, VGA_G, VGA_B} = { 24{ SW[0] } };
// assign {VGA_BLANK_N, VGA_CLK,
// VGA_HS, VGA_SYNC_N, VGA_VS} = { 5{ SW[0] } };

endmodule

Listing C.17: soc_system_top.sv

C.2 Software

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/time.h>
#include <math.h>

#include "game_boy.h"
#include "usbkeyboard.h"
#include "usb_HID_keys.h"

#define CART_HEADER_ADDR 0x0100

// Joypad keys (configure here)
#define JOYPAD_RIGHT KEY_D
#define JOYPAD_LEFT KEY_A
#define JOYPAD_UP KEY_W
#define JOYPAD_DOWN KEY_S
#define JOYPAD_A KEY_J
#define JOYPAD_B KEY_K
#define JOYPAD_SELECT KEY_O
#define JOYPAD_START KEY_I

// DE1-Soc H2F AXI bus address
#define H2F_AXI_BASE 0x00000000
#define H2F_AXI_SPAN 0x04000000
#define SDRAM_OFFSET 0x02000000

// FUNCTION DECLARATIONS

uint8_t update_joypad_status(uint8_t key);
void send_joypad_status(uint8_t reg);
char parse_printable_key(int key, bool mod, bool caps);
void read_cart();
void read_cart_header(FILE* ptr);
void save_RAM_to_SAV_file();
void read_SAV_file();

// TYPEDEFS

// Cartridge Header Information

typedef struct {
  uint8_t begin[4];   // 0x0100-0x0103: cart begin code
  uint8_t N_logo[48]; // 0x0104-0x0133: scrolling Nintendo graphic (MUST NOT MODIFY)
  uint8_t game_title[15]; // 0x0134-0x0142: title of the game in upper case ASCII
  uint8_t color_gb; // 0x0143: 0x80 if color GB; else 0x00
  uint8_t licensee_new[2]; // 0x0144-0x0145: (new) licensee code (normally both 0x00 if address 0x014B != 0x33)
  uint8_t SGB_flag; // 0x0146: GB/SGB indicator (0x00 = GB; 0x03 = SGB)
  uint8_t type;   // 0x0147: cartridge type
  uint8_t ROM_size; // 0x0148: ROM size
  uint8_t RAM_size; // 0x0149: RAM size
  uint8_t dest_code; // 0x014A: destination code (0x00 = Japanese; 0x01 = Non-Japanese)
  uint8_t licensee_old; // 0x014B: (old) licensee code (0x33 = check addresses 0x0144-0x0145; 0x79 = Accolade; 0xA4 = Konami)
  uint8_t mask_ROM; // 0x014C: mask ROM version number (usually 0x00)
  uint8_t comp_check; // 0x014D: complement check (PROGRAM WON’T RUN IF INCORRECT)
  uint8_t checksum[2]; // 0x014E-0x014F: checksum (GB ignores this value)
} cart_header;
// GLOBAL VARIABLES

int GB_fd; // ioctl file descriptor

int mmap_fd; // /dev/mem file id

void *h2f_virtual_base; // H2F AXI bus virtual address

volatile uint8_t * sdram_ptr = NULL;

struct libusb_device_handle* keyboard;

uint8_t endpoint_address;

uint8_t joystick_reg; // bit 7-4: START, SELECT, B, A
                          // bit 3-0: DOWN, UP, LEFT, RIGHT

// Cartridge information

cart_header cart_info;

uint8_t *cart_data; // pointer to cart data start address

uint8_t *save_data; // pointer to save data start address

int ROM_size; // in bytes

int RAM_size; // in bytes

uint16_t ROM_bank; // number of ROM banks

uint8_t MBC_num; // MBC number

uint8_t RAM_bank; // number of RAM banks

char ROM_FILE[200];

char *ROM_name;

char SAV_FILE[200];

// MAIN PROGRAM

int main(int argc, char *argv[])
{

    if (argc != 2)
    {

    

printf("ERROR: no ROM file was specified. \n");
exit(1);
}
strcpy_ROM_FILE, argv[1]);
char tmp[200];
strcpy(tmp, ROM_FILE);
ROM_name = strtok(tmp, ".");
strcpy(SAV_FILE, ROM_name);
strcat(SAV_FILE, ".sav");

// check if joypad keys are valid (cannot be ESC or modifiers or SPACE)
uint8_t joypad_keys[8] = {
    JOYPAD_UP, JOYPAD_DOWN, JOYPAD_LEFT, JOYPAD_RIGHT,
    JOYPAD_A, JOYPAD_B, JOYPAD_START, JOYPAD_SELECT};

for (uint8_t i = 0; i < 8; i++)
{
    if (joypad_keys[i] == KEY_ESC || joypad_keys[i] == KEY_SPACE ||
        joypad_keys[i] == KEY_LEFTCTRL || joypad_keys[i] ==
        KEY_RIGHTCTRL ||
        joypad_keys[i] == KEY_LEFTSHIFT || joypad_keys[i] ==
        KEY_RIGHTSHIFT ||
        joypad_keys[i] == KEY_LEFTALT || joypad_keys[i] ==
        KEY_RIGHTALT ||
        joypad_keys[i] == KEY_LEFTMETA || joypad_keys[i] ==
        KEY_RIGHTMETA)
    {
        printf("Not a valid joypad key! Please reconfigure! \n");
        exit(1);
    }
}
static const char filename[] = "/dev/game_boy";
if ((GB_fd = open(filename, O_RDWR)) == -1)
{
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
}

// LOAD CARTRIDGE ROM AND SAV RAM TO DE1-SoC SDRAM

// Declare volatile pointers to I/O registers (volatile
// means that IO load and store instructions will be used
// to access these pointer locations,

// === get FPGA addresses ============
// Open /dev/mem
if ((mmap_fd = open("/dev/mem", (O_RDWR | O_SYNC))) == -1)
{
    printf("ERROR: could not open "/dev/mem"\n");
    return(1);
}

// ------------------------------------------
// get virtual address for
// AXI bus address
h2f_virtual_base = mmap(NULL, H2F_AXI_SPAN, (PROT_READ | PROT_WRITE),
MAP_SHARED, mmap_fd, H2F_AXI_BASE);
if (h2f_virtual_base == MAP_FAILED)
{
    printf("ERROR: mmap3() failed...
");
    close(mmap_fd);
    return(1);
}
printf("**************************************************\n");

sdram_ptr = (uint8_t *) (h2f_virtual_base);

read_cart();

printf("**************************************************\n");

sdram_ptr = (uint8_t *) (h2f_virtual_base + SDRAM_OFFSET);

if (RAM_size != 0)
{
    for (int i = 0; i < RAM_size; i++)
    {
        *(sdram_ptr + i) = 0;  // clear RAM
    }

    read_SAV_file();

    printf("**************************************************\n");
}

// MBC info

sdram_ptr = (uint8_t *) (h2f_virtual_base + SDRAM_OFFSET);

*(sdram_ptr - 5) = MBC_num;         // MBC number
*(sdram_ptr - 4) = RAM_bank;        // Number of banks
*(sdram_ptr - 3) = ROM_bank & 0xFF;  // Lower byte
*(sdram_ptr - 2) = ROM_bank >> 8;    // MSB
*(sdram_ptr - 1) = 1;               // load complete

// JOYPAD INPUT CONTROL

struct usb_keyboard_packet packet;

int transferred;

t char keystate[20];
bool shift;
bool cap_state = 0;
bool double_speed = 0;
/* Open the keyboard */

if ((keyboard = openkeyboard(&endpoint_address)) == NULL) {
    fprintf(stderr, "Did not find a keyboard\n");
    exit(1);
}

/* Look for and handle keypresses */
for (;;) {
    libusb_interrupt_transfer(keyboard, endpoint_address,
        (uint8_t*) & packet, sizeof(packet),
        &transferred, 0);
    if (transferred == sizeof(packet)) {
        sprintf(keystate, "%02x %02x %02x %02x %02x %02x %02x\n", packet.
            modifiers, packet.keycode[0],
            packet.keycode[1], packet.keycode[2], packet.keycode[3],
            packet.keycode[4], packet.keycode[5]);
        //printf("%s\n", keystate);

        shift = packet.modifiers == USB_LSHIFT || packet.modifiers ==
            USB_RSHIFT;

        // will execute the last pressed key
        if (packet.keycode[0] == KEY_ESC) /* ESC pressed? */
            {
            if (RAM_size != 0)
                {
                *(sdram_ptr - 1) = 0;
                save_RAM_to_SAV_file();
                printf("************************************************\n");
                }
else if (packet.keycode[0] == KEY_SPACE) {
    double_speed = !double_speed;
    *(sdram_ptr - 6) = double_speed;
    if (double_speed)
        printf("Double speed: ON \n");
    else
        printf("Double speed: ON \n");
}
else if (packet.keycode[0] == KEY_CAPSLOCK || packet.keycode[1] == KEY_CAPSLOCK ||
    {  
cap_state = !cap_state;
    if (cap_state)
        printf("CAPS on \n");
    else
        printf("CAPS off \n");
}
else if (packet.keycode[5]) {
    // convert usb keycodes to ASCII
    char input_key = parse_printable_key(packet.keycode[5], shift, cap_state);
    printf("Key5 pressed: %c \n", input_key);

    uint8_t reg = 0;
    for (uint8_t i = 0; i <= 5; i++)
```c
{  
    reg += update_joypad_status(packet.keycode[i]);
}

if (reg)  
{  
    joypad_reg = reg;
    send_joypad_status(joypad_reg);
}
else if (packet.keyCode[4])
{
    // convert usb keycodes to ASCII  
    char input_key = parse_printable_key(packet.keyCode[4],  
        shift, cap_state);
    printf("Key4 pressed: %c \n", input_key);

    uint8_t reg = 0;  
    for (uint8_t i = 0; i <= 4; i++)  
    {  
        reg += update_joypad_status(packet.keycode[i]);
    }

    if (reg)  
    {  
        joypad_reg = reg;
        send_joypad_status(joypad_reg);
    }
}
else if (packet.keyCode[3])
{
    // convert usb keycodes to ASCII  
    char input_key = parse_printable_key(packet.keyCode[3],  
```
shift, cap_state);

    printf("Key3 pressed: %c \n", input_key);

    uint8_t reg = 0;
    for (uint8_t i = 0; i <= 3; i++)
    {
        reg += update_joypad_status(packet.keycode[i]);
    }

    if (reg)
    {
        joystick_reg = reg;
        send_joypad_status(joystick_reg);
    }

    else if (packet.keycode[2])
    {
        // convert usb keycodes to ASCII
        char input_key = parse_printable_key(packet.keycode[2],
            shift, cap_state);
        printf("Key2 pressed: %c \n", input_key);

        uint8_t reg = 0;
        for (uint8_t i = 0; i <= 2; i++)
        {
            reg += update_joypad_status(packet.keycode[i]);
        }

        if (reg)
        {
            joystick_reg = reg;
            send_joypad_status(joystick_reg);
        }
else if (packet.keycode[1]) {
    // convert usb keycodes to ASCII
    char input_key = parse_printable_key(packet.keycode[1], shift, cap_state);
    printf("Key1 pressed: %c \n", input_key);

    uint8_t reg = 0;
    for (uint8_t i = 0; i <= 1; i++)
    {
        reg += update_joypad_status(packet.keycode[i]);
    }

    if (reg)
    {
        joystick_reg = reg;
        send_joypad_status(joystick_reg);
    }
}
else if (packet.keycode[0]) {
    // convert usb keycodes to ASCII
    char input_key = parse_printable_key(packet.keycode[0], shift, cap_state);
    printf("Key0 pressed: %c \n", input_key);

    uint8_t reg = update_joypad_status(packet.keycode[0]);
    if (reg)
    {
        joystick_reg = reg;
        send_joypad_status(joystick_reg);
    }
else if (packet.keycode[0] == KEY_NONE)
{
    joypad_reg = 0;
    send_joypad_status(joypad_reg);
}

return 0;

// FUNCTION DEFINITIONS

uint8_t update_joypad_status(uint8_t key)
{
    uint8_t reg;
    switch (key)
    {
    case JOYPAD_RIGHT:
        reg = (1 << 0); break;
    case JOYPAD_LEFT:
        reg = (1 << 1); break;
    case JOYPAD_UP:
        reg = (1 << 2); break;
    case JOYPAD_DOWN:
        reg = (1 << 3); break;
    case JOYPAD_A:
        reg = (1 << 4); break;
    case JOYPAD_B:
        reg = (1 << 5); break;
    case JOYPAD_SELECT:
        reg = (1 << 6); break;
case JOYPAD_START:
    reg = (1 << 7); break;
default:
    reg = 0;
}
return reg;

void send_joypad_status(uint8_t reg)
{
    uint8_t byte = reg;
    printf("Joypad register is: %.2X \n", byte);
    if (ioctl(GB_fd, GAME_BOY_SEND_JOYPAD_STATUS, &byte))
    {
        perror("ioctl(GAME_BOY_SEND_JOYPAD_STATUS) failed");
        return;
    }
}

char parse_printable_key(int key, bool mod, bool caps)
{
    if (key == KEY_BACKSPACE)
    {
        return 8;
    }
    if (key == KEY_ENTER || key == KEY_KPENTER)
    {
        return '\n';
    }
    switch (key)
    {
    case KEY_A:
        return (mod ^ caps) ? 'A' : 'a';
case KEY_B:
    return (mod ^ caps) ? 'B' : 'b';
case KEY_C:
    return (mod ^ caps) ? 'C' : 'c';
case KEY_D:
    return (mod ^ caps) ? 'D' : 'd';
case KEY_E:
    return (mod ^ caps) ? 'E' : 'e';
case KEY_F:
    return (mod ^ caps) ? 'F' : 'f';
case KEY_G:
    return (mod ^ caps) ? 'G' : 'g';
case KEY_H:
    return (mod ^ caps) ? 'H' : 'h';
case KEY_I:
    return (mod ^ caps) ? 'I' : 'i';
case KEY_J:
    return (mod ^ caps) ? 'J' : 'j';
case KEY_K:
    return (mod ^ caps) ? 'K' : 'k';
case KEY_L:
    return (mod ^ caps) ? 'L' : 'l';
case KEY_M:
    return (mod ^ caps) ? 'M' : 'm';
case KEY_N:
    return (mod ^ caps) ? 'N' : 'n';
case KEY_O:
    return (mod ^ caps) ? 'O' : 'o';
case KEY_P:
    return (mod ^ caps) ? 'P' : 'p';
case KEY_Q:
    return (mod ^ caps) ? 'Q' : 'q';
case KEY_R:
return (mod ^ caps) ? 'R' : 'r';
case KEY_S:
    return (mod ^ caps) ? 'S' : 's';
case KEY_T:
    return (mod ^ caps) ? 'T' : 't';
case KEY_U:
    return (mod ^ caps) ? 'U' : 'u';
case KEY_V:
    return (mod ^ caps) ? 'V' : 'v';
case KEY_W:
    return (mod ^ caps) ? 'W' : 'w';
case KEY_X:
    return (mod ^ caps) ? 'X' : 'x';
case KEY_Y:
    return (mod ^ caps) ? 'Y' : 'y';
case KEY_Z:
    return (mod ^ caps) ? 'Z' : 'z';
case KEY_1:
    return (mod) ? '!' : '1';
case KEY_2:
    return (mod) ? '@' : '2';
case KEY_3:
    return (mod) ? '#' : '3';
case KEY_4:
    return (mod) ? '$' : '4';
case KEY_5:
    return (mod) ? '%' : '5';
case KEY_6:
    return (mod) ? '^' : '6';
case KEY_7:
    return (mod) ? '&' : '7';
case KEY_8:
    return (mod) ? '*' : '8';
479
480
481
482

case KEY_9 :
return ( mod ) ? '( ' : '9 ';
case KEY_0 :
return ( mod ) ? ') ' : '0 ';

483

case KEY_TAB :

484

return 9;

485

case KEY_SPACE :

486

return ' ';

487

case KEY_MINUS :

488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511

return ( mod ) ? '_ ' : ' - ';
case KEY_EQUAL :
return ( mod ) ? '+ ' : '= ';
case KEY_LEFTBRACE :
return ( mod ) ? '{ ' : '[ ';
case KEY_RIGHTBRACE :
return ( mod ) ? '} ' : '] ';
case KEY_BACKSLASH :
return ( mod ) ? '| ' : ' \\ ';
case KEY_SEMICOLON :
return ( mod ) ? ': ' : '; ';
case KEY_APOSTROPHE :
return ( mod ) ? '" ' : '\ ' ';
case KEY_GRAVE :
return ( mod ) ? '~ ' : ' ` ';
case KEY_COMMA :
return ( mod ) ? ' < ' : ' , ';
case KEY_DOT :
return ( mod ) ? ' > ' : '. ';
case KEY_SLASH :
return ( mod ) ? '? ' : '/ ';
case KEY_KPSLASH :
return '/ ';
case KEY_KPASTERISK :

249


return '*';
case KEY_KPMINUS:
    return '-';
case KEY_KPPLUS:
    return '+';
case KEY_KP1:
    return '1';
case KEY_KP2:
    return '2';
case KEY_KP3:
    return '3';
case KEY_KP4:
    return '4';
case KEY_KP5:
    return '5';
case KEY_KP6:
    return '6';
case KEY_KP7:
    return '7';
case KEY_KP8:
    return '8';
case KEY_KP9:
    return '9';
case KEY_KP0:
    return '0';
case KEY_KPDOT:
    return '.';
default:
    return ' ';
void read_cart()
{
    FILE* cart_ptr;
    cart_ptr = fopen(ROM_FILE, "rb");

    if (cart_ptr == NULL)
    {
        printf("Unable to open the ROM file \"%s\"\n", ROM_FILE);
        exit(1);
    }
    else
    {
        printf("ROM file \"%s\" opened successfully! \n\n", ROM_FILE);
        printf("Cartridge information: \n");
        read_cart_header(cart_ptr);

        cart_data = (uint8_t *)malloc(ROM_size);

        fseek(cart_ptr, 0, SEEK_SET);
        for (int i = 0; i < ROM_size; i++)
        {
            fread(cart_data + i, 1, 1, cart_ptr);
            //printf("Address %.4X: %.2X \n", i, *(cart_data + i));
            //printf("Address of cart_data[\%d] is: %p \n", i, (void *)(cart_data+i));
        }
        fclose(cart_ptr);

        printf("Loading %d bytes of ROM into SDRAM...", ROM_size);
        for (int i = 0; i < ROM_size; i++)
        {
            *(sdram_ptr + i) = *(cart_data + i);
        }
    }
}
// printf("SDRAM %.4X: %.2X \n", i, *(sdram_ptr+i));
  }
  printf("complete! \n");
}
}

// get cartridge information (e.g. MBC type, ROM size, RAM size, etc.)
void read_cart_header(FILE * ptr) {
  fseek(ptr, CART_HEADER_ADDR, SEEK_SET);
  fread(&cart_info, 1, 0x14F - CART_HEADER_ADDR + 0x01, ptr);

  printf("- Game title: %s \n", cart_info.game_title);

  if (cart_info.color_gb == 0x80)
    printf("- Console: Game Boy Color \n");
  else
    printf("- Console: Game Boy \n");

  if (cart_info.SGB_flag == 0x03)
    printf("- Super Game Boy functions supported \n");

  char cart_str[26];
  switch (cart_info.type) {
  case 0x00:
    strcpy(cart_str, "ROM ONLY");
    MBC_num = 0;
    break;
  case 0x01:
    strcpy(cart_str, "ROM+MBC1");
    MBC_num = 1;
    break;
case 0x02:
    strcpy(cart_str, "ROM+MBC1+RAM");
    MBC_num = 1;
    break;
case 0x03:
    strcpy(cart_str, "ROM+MBC1+RAM+BATT");
    MBC_num = 1;
    break;
case 0x05:
    strcpy(cart_str, "ROM+MBC2");
    MBC_num = 2;
    break;
case 0x06:
    strcpy(cart_str, "ROM+MBC2+BATTERY");
    MBC_num = 2;
    break;
case 0x08:
    strcpy(cart_str, "ROM+RAM");
    MBC_num = 1;
    break;
case 0x09:
    strcpy(cart_str, "ROM+RAM+BATTERY");
    MBC_num = 1;
    break;
case 0x0B:
    strcpy(cart_str, "ROM+MMM01");
    break;
case 0x0C:
    strcpy(cart_str, "ROM+MMM01+SRAM");
    break;
case 0x0D:
    strcpy(cart_str, "ROM+MMM01+SRAM+BATT");
    break;
case 0xF:
    strcpy(cart_str, "ROM+MBC3+TIMER+BATT");
    MBC_num = 3;
    break;

case 0x10:
    strcpy(cart_str, "ROM+MBC3+TIMER+RAM+BATT");
    MBC_num = 3;
    break;

case 0x11:
    strcpy(cart_str, "ROM+MBC3");
    MBC_num = 3;
    break;

case 0x12:
    strcpy(cart_str, "ROM+MBC3+RAM");
    MBC_num = 3;
    break;

case 0x13:
    strcpy(cart_str, "ROM+MBC3+RAM+BATT");
    MBC_num = 3;
    break;

case 0x19:
    strcpy(cart_str, "ROM+MBC5");
    MBC_num = 5;
    break;

case 0x1A:
    strcpy(cart_str, "ROM+MBC5+RAM");
    MBC_num = 5;
    break;

case 0x1B:
    strcpy(cart_str, "ROM+MBC5+RAM+BATT");
    MBC_num = 5;
    break;

case 0x1C:
strcpy(cart_str, "ROM+MBC5+RUMBLE");
MBC_num = 5;
break;
case 0x1D:
    strcpy(cart_str, "ROM+MBC5+RUMBLE+SRAM");
    MBC_num = 5;
    break;
case 0x1E:
    strcpy(cart_str, "ROM+MBC5+RUMBLE+SRAM+BATT");
    MBC_num = 5;
    break;
case 0x1F:
    strcpy(cart_str, "Pocket Camera");
    break;
case 0xFD:
    strcpy(cart_str, "Bandai TAMA5");
    break;
case 0xFE:
    strcpy(cart_str, "Hudson HuC-3");
    break;
case 0xFF:
    strcpy(cart_str, "Hudson HuC-1");
    break;
default:
    strcpy(cart_str, "Invalid cartridge type");
    exit(1);
}
printf("- Cartridge type: %s \n", cart_str);

switch (cart_info.ROM_size)
{
    case 0x00:
        ROM_bank = 2;  // 32kB
break;

case 0x01:
    ROM_bank = 4; // 64kB
    break;

case 0x02:
    ROM_bank = 8; // 128kB
    break;

case 0x03:
    ROM_bank = 16; // 256kB
    break;

case 0x04:
    ROM_bank = 32; // 512kB
    break;

case 0x05:
    ROM_bank = 64; // 1MB
    break;

case 0x06:
    ROM_bank = 128; // 2MB
    break;

case 0x07:
    ROM_bank = 256; // 4MB
    break;

case 0x08:
    ROM_bank = 512; // 8MB
    break;

case 0x52:
    ROM_bank = 72; // 1.1MB
    break;

case 0x53:
    ROM_bank = 80; // 1.2MB
    break;

case 0x54:
    ROM_bank = 96; // 1.5MB

break;

default:
    printf("Invalid ROM size \n");
    exit(1);
}

ROM_size = ROM_bank * 16 * 1024;
printf("- ROM size: %d bytes (%d banks) \n", ROM_size, ROM_bank);

switch (cart_info.RAM_size)
{
    case 0x00:
        RAM_bank = 0;
        RAM_size = 0;
        break;
    case 0x01:
        RAM_bank = 1;
        RAM_size = 2 * 1024;  // 2kB
        break;
    case 0x02:
        RAM_bank = 1;
        RAM_size = 8 * 1024;  // 8kB
        break;
    case 0x03:
        RAM_bank = 4;
        RAM_size = 4 * 8 * 1024;  // 32kB
        break;
    case 0x04:
        RAM_bank = 16;
        RAM_size = 16 * 8 * 1024;  // 128kB
        break;
    default:
        printf("Invalid RAM size \n");
        exit(1);
```c
}  
printf("- RAM size: %d bytes (%d banks) \n", RAM_size, RAM_bank);
}

// saves RAM contents (in SDRAM) to a SAV file
void save_RAM_to_SAV_file()
{
    FILE* save_ptr;
    save_ptr = fopen(SAV_FILE, "wb");

    if (save_ptr == NULL)
    {
        printf("Unable to open the SAV file \"%s\"! \n", SAV_FILE);
        exit(1);
    }
    else
    {
        printf("SAV file \"%s\" opened successfully! \n\n", SAV_FILE);

        sdram_ptr = (uint8_t *) (h2f_virtual_base + SDRAM_OFFSET);

        save_data = (uint8_t *) malloc(RAM_size);

        printf("Saving game data... \n");
        for (int i = 0; i < RAM_size; i++)
        {
            *(save_data + i) = *(sdram_ptr + i);
        }

        printf("Writing to file: %s \n", SAV_FILE);
        fwrite(save_data, 1, RAM_size, save_ptr);
        fclose(save_ptr);
    }
}  
```
void read_SAV_file()
{
    FILE* save_ptr;
    save_ptr = fopen(SAV_FILE, "rb");

    if (save_ptr == NULL)
    {
        printf("No SAV file was loaded \n");
    }
    else
    {
        printf("SAV file \%s\" opened successfully! \n\n", SAV_FILE);

        save_data = (uint8_t *)malloc(RAM_size);

        fseek(save_ptr, 0, SEEK_SET);
        for (int i = 0; i < RAM_size; i++)
        {
            fread(save_data + i, 1, 1, save_ptr);
            //printf("Address %.4X: %.2X \n", i, *(save_data + i));
            //printf("Address of save_data[\%d] is: %p \n", i, (void *)(save_data+i));
        }
        fclose(save_ptr);

        printf("Loading \%d bytes of RAM into SDRAM...", RAM_size);
        for (int i = 0; i < RAM_size; i++)
        {
            *(sdram_ptr + i) = *(save_data + i);
            //printf("SDRAM %.4X: %.2X \n", i, *(sdram_ptr+i));
        }
}
Listing C.18: main.c

/* * Device driver for the Game Boy joypad * * A Platform device implemented using the misc subsystem * * Justin Hu * Columbia University * * References: * Linux source: Documentation/driver-model/platform.txt * drivers/misc/arm-charlcd.c * http://www.linuxforu.com/tag/linux-device-drivers/ * http://free-electrons.com/docs/ * * "make" to build * insmod game_boy.ko * * Check code style with * checkpatch.pl --file --no-tree game_boy.c */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "game_boy.h"

#define DRIVER_NAME "game_boy"

#define JOYPAD_REG(x)(x)

//

// -----------------------------------------------------

/*
 * Information about our device
 */

struct game_boy_dev {
    struct resource res; /* Resource: our registers */
    void __iomem* virtbase; /* Where registers can be accessed in memory */
    uint8_t joypad_status;  // current joypad status
}

dev;

static void write_joypad_register(uint8_t * reg)
{
    iowrite8(*reg, JOYPAD_REG(dev.virtbase));
    dev.joypad_status = *reg;
}

//
/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */

static long game_boy_ioctl(struct file* f, unsigned int cmd, unsigned long arg)
{
    uint8_t joypad_reg;

    switch (cmd) {
    case GAME_BOY_SEND_JOYPAD_STATUS:
        if (copy_from_user(&joypad_reg, (uint8_t*)arg, sizeof(uint8_t)))
        {
            return -EACCES;
        }
        write_joypad_register(&joypad_reg);
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

//

/***************************************************************************/

/* The operations our device knows how to do */
static const struct file_operations game_boy_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = game_boy_ioctl,
};

/* Information about our device for the "misc" framework -- like a char
dev */
static struct miscdevice game_boy_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &game_boy_fops,
};

//
**********************************************************************

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init game_boy_probe(struct platform_device* pdev)
{
    uint8_t joypad_init = 0x00; // initialize joypad

    int ret;

    /* Register ourselves as a misc device: creates /dev/game_boy */
    ret = misc_register(&game_boy_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &pdev->dev.res);
    if (ret) {

ret = -ENOENT;
goto out_deregister;
}

/* Make sure we can use these registers */
if (request_mem_region(dev.res.start, resource_size(&dev.res),
  DRIVER_NAME) == NULL) {
  ret = -EBUSY;
  goto out_deregister;
}

/* Arrange access to our registers */
dev.virtbase = of_iomap(pdev->dev.of_node, 0);
if (dev.virtbase == NULL) {
  ret = -ENOMEM;
  goto out_release_mem_region;
}

write_joypad_register(&joypad_init);

return 0;

out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
  misc_deregister(&game_boy_misc_device);
  return ret;
}

/* Clean-up code: release resources */
static int game_boy_remove(struct platform_device* pdev)
{
  iounmap(dev.virtbase);
release_mem_region(dev.res.start, resource_size(&dev.res));
.misc_deregister(&game_boy_misc_device);

return 0;
}

//

******************************************************************************

/* Which "compatible" string(s) to search for in the Device Tree */
#ifndef CONFIG_OF
static const struct of_device_id game_boy_of_match[] = {
    { .compatible = "csee4840,joypad-1.0" },
    {} ,
};
MODULE_DEVICE_TABLE(of, game_boy_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver game_boy_driver = {
    .driver = {
        .name = DRIVER_NAME ,
        .owner = THIS_MODULE ,
        .of_match_table = of_match_ptr(game_boy_of_match),
    },
    .remove = __exit_p(game_boy_remove),
};

/* Called when the module is loaded: set things up */
static int __init game_boy_init(void)
{
    pr_info(DRIVER_NAME " : init\n");
    return platform_driver_probe(&game_boy_driver, game_boy_probe);
static void __exit game_boy_exit(void)
{
    platform_driver_unregister(&game_boy_driver);
    pr_info(DRIVER_NAME " : exit\n");
}

module_init(game_boy_init);
module_exit(game_boy_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Justin Hu, Columbia University");
MODULE_DESCRIPTION("Game Boy joystick driver");
```c
#ifndef _GAME_BOY_H
#define _GAME_BOY_H

#include <linux/ioct1.h>

#define GAME_BOY_MAGIC 'q'

/* ioctl's and their arguments */
#define GAME_BOY_SEND_JOYPAD_STATUS   _IOW(GAME_BOY_MAGIC, 1, uint8_t *)

#endif
```

Listing C.20: game_boy.h