FPGABoy

An Implementation of the Nintendo GameBoy on an FPGA

Trevor Rundell, Oleg Kozhushnyan

6.111 Final Project

May 14, 2009

Table of Contents

Table of Figures

Table of Tables

Abstract

FPGABoy is an re-implementation of the classic GameBoy system, built on an FPGA. The system was developed almost entirely from scratch using an open source implementation of a Z80 processor.

System Overview

At the highest level, a game system is just a box that takes some sort of user input, and produces video and audio as an output. We decided to implement such a system, but with a twist. Instead of creating our own, we decided to pay homage to one of the greatest hand held game consoles ever created, the Nintendo GameBoy.

The GameBoy was released in 1989 and since then has been very well documented by the hacker community. We used all the detailed documentation [1, 2] to our advantage when engineering our version of the legendary console. During our design stage we broke the system up into the components shown in Figure 1.

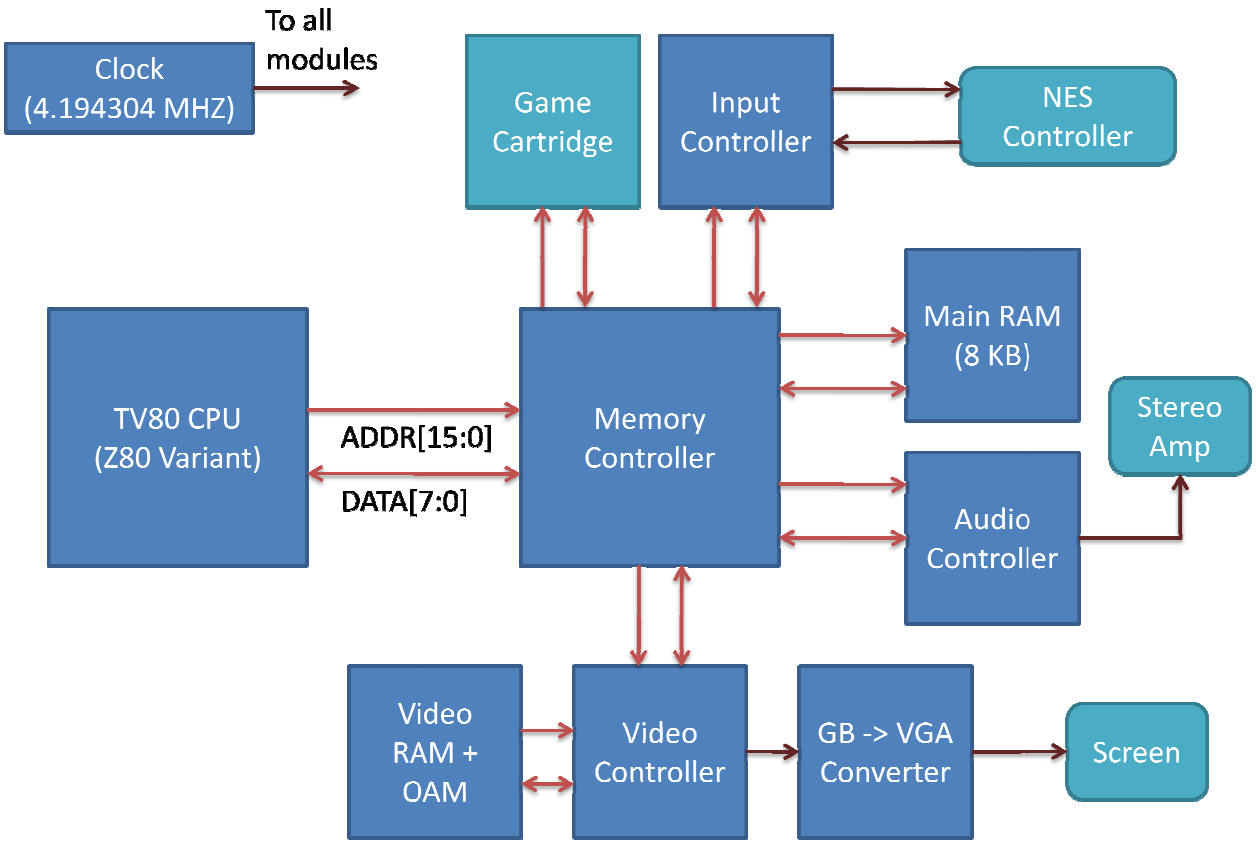


Figure 1. Global system diagram

At the core, we decided to use an open source implementation of the Z80 CPU known as TV80 in order to lessen our workload so we could concentrate on implementing the game system itself. The CPU communicates with the rest of the system through a module known as the Memory Controller. This module acts as a giant switch and handles the communication between the CPU and the rest of the sub modules.

The two primary sub modules that were at the top of our list for implementation were the Input Controller and the Video Controller. These two modules make up the meat of the project and are absolutely necessary for a functioning game system. Next was the Game Cartridge module and the Video Converter that handled interfacing between the FPGABoy and the cartridge and display peripherals. The third sub module known as the Audio Controller was placed low on our list and ultimately never implemented as it was not vital for the full system functionality.

We developed the design in Figure 1 very early on and stuck to it through the whole project. In the end, our final implementation looks nearly identical, at a high level, to our initial design.

Module Descriptions

Central Processing Unit (Trevor)

The original GameBoy uses a custom LR35902 processor, a variant of the Zilog Z80 8-bit processor. This processor has a well slightly different instruction set from the original Z80, but it is still very well documented. Rather than implement our own CPU, we decided to use the open source TV80 core [4], created by Guy Hutchison. The TV80 has built-in support (albeit experimental) for the GameBoy instruction set out of the block which greatly reduced our construction time for this module. However, the core is not without some extremely subtle bugs and a great deal of time was invested to track these down and fix them. As a result, FPGABoy contains a number of debugging features that expose various elements of the CPU's internal state. Additionally, breakpoints and instruction stepping allowed examination of the CPU state at very precise locations within the game's themselves.

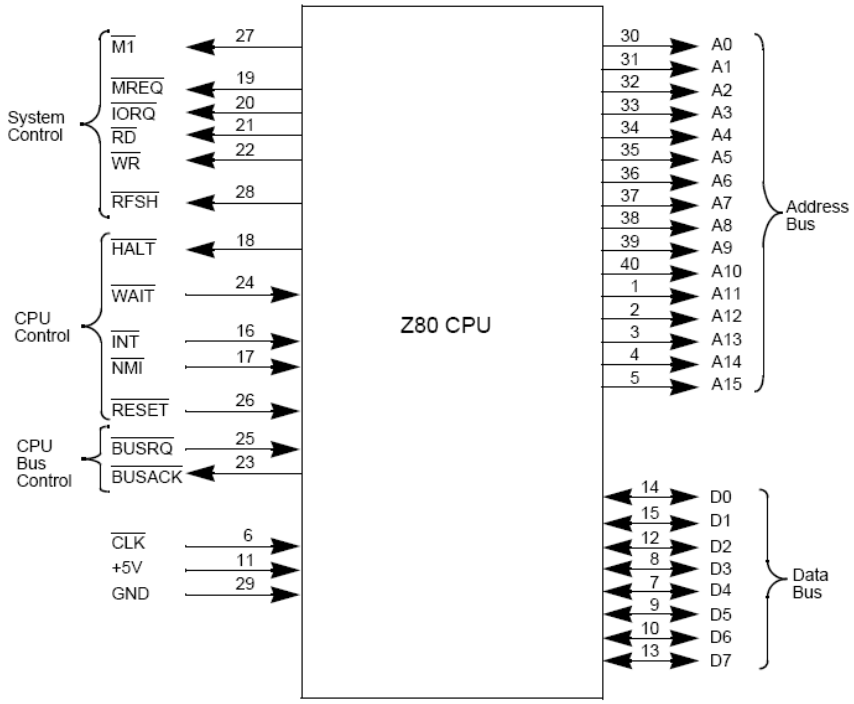


Figure 2: The Z80 pin configuration *(Source: www.zilog.com/docs/z80/um0080.pdf)*

CPU Debugging Features

Hex Display Debug Readout

We tried to make good use of the 64-bit hex display readout on the labkit to display as much information about the CPU's state as possible. To accomplish this the display was divided into five sections. The left most 8 digits are used to display the values on the CPU's address and data busses. Right right most 8 digits are used to display any one of five register combinations. By pressing buttons 0-3 and Enter on the labkit a user can switch between these register outputs to see every major aspect of the CPU state that can be controlled by a GameBoy game.

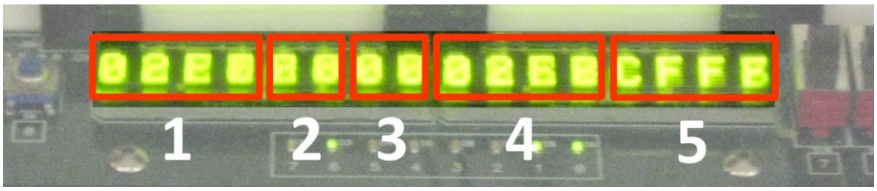


Figure 3: The hex display debug readout

Display	Readout
1	16 bit address bus
2	8 bit data in bus
3	8 bit data out bus

4	16 bit selectable register 0 -> AF 1 -> DE 2 -> PC 3 -> IME flags Enter -> N/A
5	16 bit selectable register 0 -> BC 1 -> HL 2 -> SP 3 -> Int Req / Ack Flags Enter -> Breakpoint Addr

LED Debug Readout

The LED's on the labkit are tied to various pins on the CPU. This allowed us to determine the I/O state of the CPU very quickly.

Table 1: LED debug readout

LED	Pin	LED	Pin
0	$\overline{\text{M1}}$	4	$\overline{\text{HALT}}$
1	$\overline{\text{MREQ}}$	5	$\overline{\text{RESET}}$
2	$\overline{\text{IORQ}}$	6	$\overline{\text{RD}}$
3	$\overline{\text{INT}}$	7	$\overline{\text{WR}}$

Debugging Switches

The labkit's switches can be used to facilitate fast debugging of the CPU.

Table 2: Debugging switches

Switch	Description
0	Power/Reset (switch off to reset)
1	Step enable
2	Breakpoint enable
3	Advance clock (when in step mode)
4-7	Set breakpoint digit

Breakpoint Module

A breakpoint module allows FPGABoy to halt execution when specific instructions are reached. A breakpoint can be set at any time using the hex readout and the labkit switches. The CPU address bus is continuously compared with this value and when the two match the CPU will

pause if breakpoints are enabled. A user can then step through instructions manually, resume normal execution or set a new breakpoint.

CPU Bugs

There were three bugs in the TV80 implementation that had to be fixed before FPGABoy was playable. These ranged from fairly straightforward to extremely subtle. We intend to submit our patches back to the TV80 core repository.

Carry Flag

We noticed that the 16-bit addition instruction LD HL,SP+DD was generating incorrect output. Upon further investigation it was revealed that the ALU's carry flag was not being properly cleared, effectively resulting in the second half of the add instruction always operating as if the carry flag were set. This caused results like the following: $8000h + 0020h = 8120h$.

We were able to fix this by rearranging the locations of the various ALU flags within the flags register itself. The problem ultimately stemmed from a slight difference in the positions of these flags between the GameBoy processor and an actual Z80.

I/O Port Loads

The instruction LD (FF00+C), A was executing as if it were LD (BC),A. This was causing routines that executed in the high ram area of the memory map to execute as NOPs instead. This was a fairly simple fix once we identified the bug.

RETI

The RETI instruction was failing to re-enable the global interrupt flag after returning from an interrupt. This caused many games to lock up or get stuck in menus while they waited for interrupts that never came.

Memory Controller (MMU) (Oleg)

The memory controller is one of the most important components of the FPGABoy system as it ties together all the other modules to allow them to communicate with the CPU. Essentially it acts as a large multiplexer that connects the address and data bus of the CPU to the other modules. In addition to handling the memory map, the module implements the ability for the video module to perform direct memory access(DMA). The DMA component is vital to the video module as it provides the fastest way to copy data into sprite memory. For more details on the video module refer to the corresponding section.

Memory Map

The memory map implemented by the memory controller takes the devices such as the input converter, video module, timer module and the interrupt module and assigns them to regions

in a 16-bit address space. Table 3 is an overview of the memory map as implemented by the FPGABoy.

Table 3. Memory map ***(Source: http://nocash.emubase.de/pandocs.htm#memorymap)***

Start Address	End Address	Mapping
0x0000	**0x3FFF**	16KB ROM Bank 0
0x4000	**0x7FFF**	16KB ROM Bank 1
0x8000	**0x9FFF**	8KB Video RAM
0xA000	**0xBFFF**	8KB External RAM
0xC000	**0xCFFF**	4KB Internal RAM Bank 0
0xD000	**0xDFFF**	4KB Internal RAM Bank 1
0xE000	**0xFDFF**	Echo of Internal RAM (0xC000 - 0xDDFF)
0xFE00	**0xFE9F**	160B Sprite RAM (OAM)
0xFEA0	**0xFEFF**	Reserved
0xFF00	**0xFF7F**	I/O Ports
0xFF80	**0xFFFE**	High RAM
0xFFFF		Interrupt Enable Register

The only time the memory map differs from Table 3 is during the start up sequence. When the FPGABoy first starts executing, the MMU maps the first 0x100 bytes onto a special boot ROM. For more information on the boot ROM, refer to the corresponding section. Afterwards, the first 32KB of the address space are always mapped to the read only block RAM that contains the copy of the GameBoy game. The next 32KB of the address space are taken up by the mappings to the various RAM's in the system. The 8KB external and 8KB internal RAM are implemented by block memories inside the MMU itself while the 8KB of video memory resides in the video module. Following that is 160B of sprite memory that also resides in the video module. The remainder of the mapping is either reserved or is associated with the registers to control various devices.

An important aspect of the memory map module is the ability to access memory banks. For GameBoy games larger than 32KB it is necessary to be able to modify the memory map to be able to access the remainder of the game data. This is done through the memory bank controller that is embedded in the game cartridge. The controller can remap the locations occupied by the 16KB ROM Bank 1 and 4KB RAM Bank 1 to be able to access an address space much larger than available. We do not implement the memory bank controller functionality but it is necessary to explain why the ROM and RAM are partitioned as they are and appear with different Bank numbers.

The most interesting area of the memory map is the region from 0xFF00 to 0xFF7F. That region is populated by the control registers for all the other modules in the system. Instead of implementing the registers in the memory module itself, they are implemented inside the

modules that they control. That is, the memory map module just redirects the address bus, data bus, and read/write flags to the respective modules for any accesses in the I/O region. It is up to the modules themselves to handle latching data and replying to the memory accesses.

Table 4. Memory module control registers

Address	Name	Function
0xFF50	ROM_DISABLE	Disable boot ROM mapping
0xFF46	DMA	Start DMA transfer

The memory map module is controlled through two registers. They are called the ROM_DISABLE and DMA registers located at 0xFF50 and 0xFF46 respectively. The ROM_DISABLE register handles the temporary boot ROM mapping during FPGABoy initialization. When the boot ROM completes, it writes the value 0x01 into the ROM_DISABLE register [2] which causes the memory module to remap the first 0x100 bytes back onto the 16KB ROM Bank 0.

Direct Memory Access (DMA)

The DMA register is used to activate the direct memory transfer between any address in the range 0x0000-0xF19F and the sprite memory(OAM) at 0xFE00-0xFE9F [1]. The DMA is activated when the GameBoy game writes the source address of the transfer into the DMA register. Because the register is only 8-bits wide, all transfers have to start on a 0x100 byte boundary as the lowest 8-bits of the address are ignored [1].

Once the DMA starts, the memory map module stores the starting address in an internal register and disables the ability of the CPU to access all addresses except for the high RAM. Thus it is vital for the DMA to only be initiated by code that resides in the high RAM, otherwise the behavior will be unspecified. While the CPU executes in high RAM, the memory module copies 0xA0 bytes from the start address and into the OAM memory. Once the transfer completes, the mappings are restored and the CPU is free to jump back into normal execution.

The benefit of DMA is that it allows for the whole sprite table to be copied in one memory operation instead of requiring 160 memory accesses and instructions which can be used to perform other game operations.

Essentially, the DMA process is implemented as a minor FSM within the memory map module. It copies one memory location every four cycles during which it loops through its states. In the first state it enables writes to video memory and then transitions to the second state. Then it waits for one clock cycle for the write to stabilize and then transitions into the third state which disables writes to the video module. In the fourth and final state it increments the memory location counter and upon completion disables the DMA operation.

Bootstrap Rom

One important feature of FPGA is the inclusion of the original GameBoy bootstrap ROM. This is a small, built-in, program that is executed when the system is first powered on. One function of this program is to initialize the memory mapped I/O registers to their proper values. The bootstrap ROM also acts as a primitive form of DRM to prevent unlicensed games from being played. It does this by performing a checksum on small range of bytes in the cartridge's memory. If the sum of these bytes matches the expected value, the system disables the bootstrap ROM and allows the lower 255 bytes of the cartridge memory to be accessed. If the checksum fails, the system will lock up and must be reset.

The most recognizable aspect of the bootstrap ROM is the Nintendo logo that appears at the top of the screen when the system is powered on. The logo is then scrolled down to the center of the screen and two tones are played, producing the familiar "da-ding" sound, after which normal execution of the cartridge program begins. Despite the importance of the bootstrap ROM to the GameBoy experience, most GameBoy emulators do not implement this functionality. We obtained a copy of the original GameBoy bootstrap ROM online [2].

Figure 4: The Nintendo logo as generated by the bootstrap ROM

Input Converter Module (Oleg)

The input converter module is responsible for interfacing the CPU with the Nintendo Entertainment System(NES) game pad controller. The primary function of the module is to de-serialize the signal generated by the NES game pad and to convert it into the format which is determined by the GameBoy specification.

NES Game Pad Controller

We selected the NES Game Pad as an input source for the FPGABoy system due to its superiority over the inputs provided on the lab kit and the simplicity of its data protocol. Furthermore it turned out that most of the inputs on the lab kit were used for debugging purposes which made the NES controller a perfect source for providing game input. The game pad has a total of eight buttons which are exactly the same as the buttons found on the

original GameBoy system. The buttons are the directional pad that provides the up, down, left and right inputs along with four others buttons labeled A, B, start and select.

In order to communicate with the game pad we had to understand the way it generates its output. The internal diagram of the NES game pad can be seen in Figure 5.

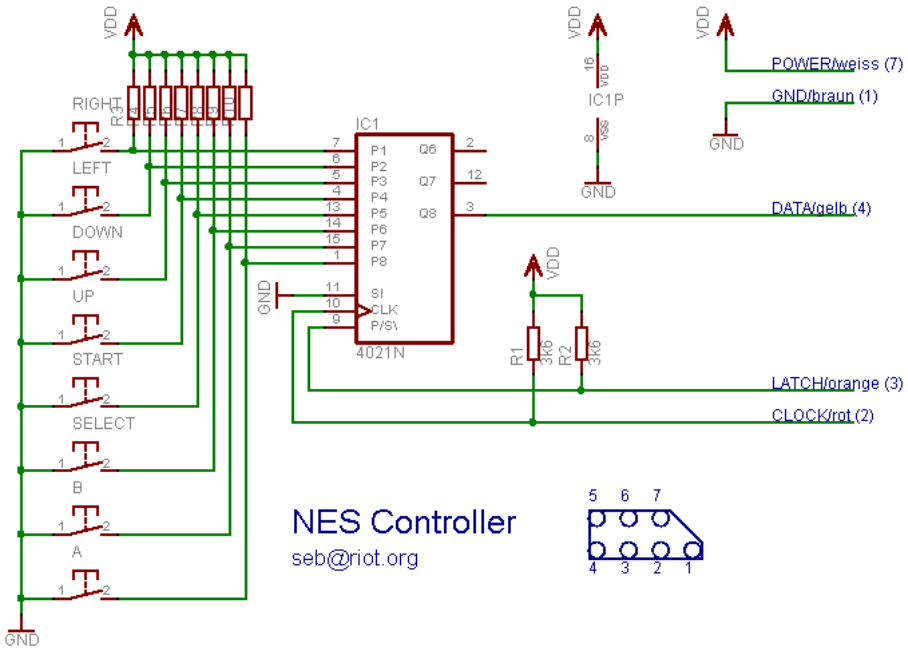


Figure 5. NES Controller schematic *(Source: http://seb.riot.org/nescontr/)*

The design of the game pad is very simple. There are eight button inputs that are pulled up high when they are not pressed. From this we can tell that we will read a value of one when a button is not pressed and a zero when it is held down as it will be shorted to ground. These inverted states are identical to the way the GameBoy expects to read the buttons as an input. The eight buttons are in turn connected to a 4021N chip. The 4021N is an eight bit shift register that converts its inputs into a serial stream.

In order to interact with the game pad, the state of the 4021N needs to be extracted. In order to do this three lines of communication are required. The lines three lines are the Data, Latch and Clock lines which are respectively the output and inputs of the 4021N chip. Whenever the Latch line goes high, the shift register starts to output its state. It outputs each one of its bits(which represent the button states) on the rising edge of the clock that is provided as an input on the Clock line. The following figure describes the communication protocol with the shift register.

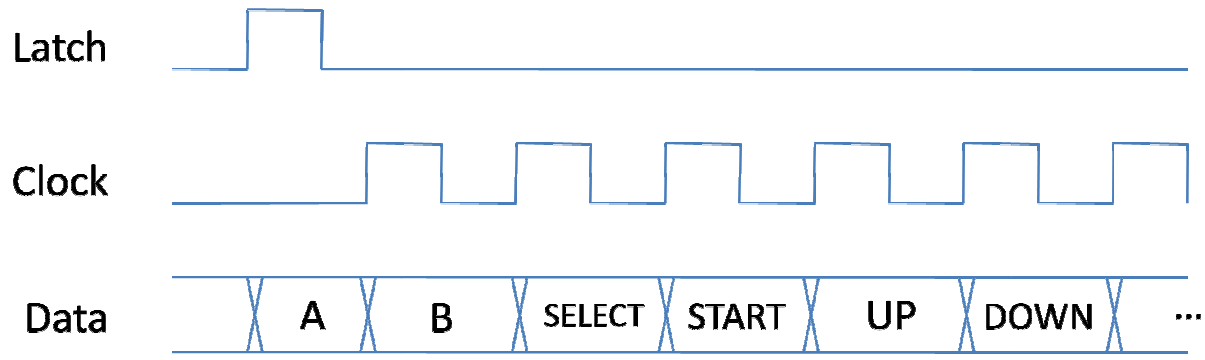


Figure 6. NES game pad communication protocol *(Source: http://seb.riot.org/nescontr/)*

Once all eight bits are read from the register, the latch can be pulled up again and the read sequence repeated. The input converter module repeats this sequence sixty times per second in order to guarantee that the user has the most responsive experience. Once the input converter has the state of all eight buttons in its internal register it is able to convert the input into the GameBoy format.

GameBoy Input Specification

The original GameBoy expects the input from the game pad in a distinct format. This format was the result of the limitation of the number of pins that were on the original CPU [1]. The limitation comes from that only six pins are used to read the state of all eight buttons. Two of the pins are used as selectors for either the four directional buttons or the A, B, select and start buttons. The other four pins read the state of the buttons selected by the previous four. The six pins are memory mapped directly into a register located at the 0xFF00 memory address [1] in the format shown in Figure 7.

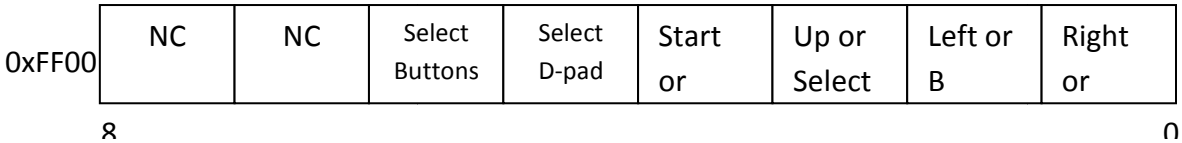


Figure 7. Input register format

When code that desires to read the input executes on the GameBoy, it writes a zero into either bit four or five of the input register to select which set of controls to read. Then it reads from the same register and extracts the state of the selected buttons from the lowest four bits. This is the functionality is supported by the converter.

Input Converter

The input converter is a combination of the previous two components and a conversion step. When the CPU asks to write into the register at 0xFF00, the input converter remembers the value that was written in order to determine the selector bits. Next, when the CPU reads from the register, the module looks at the remembered selector bits and writes the corresponding

button states onto the data bus. Because the button states are updated sixty times per second the FPGABoy is guaranteed to have the latest input at its disposal.

Interrupt Module (Trevor)

The original GameBoy is capable of generating five interrupts for various I/O devices within the system. FPGABoy supports all five interrupts, but the surrounding hardware is only capable of generating four of these. The exception is the serial transfer interrupt which is left unimplemented.

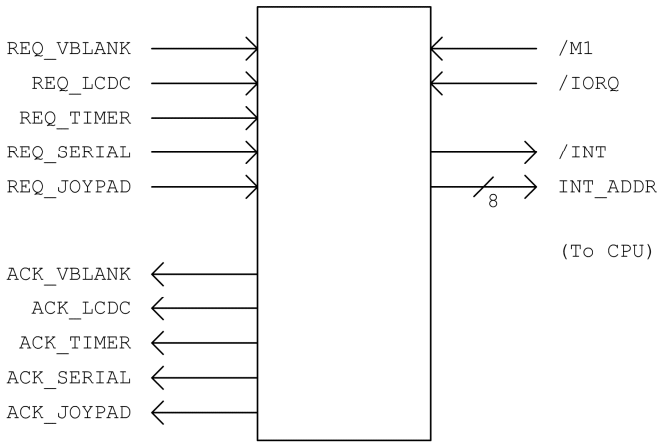


Figure 8: Interrupt module block diagram

For each interrupt there is one request pin (input) and one acknowledge (output) pin. These pins can be connected to whichever module is in charge of generating that particular interrupt.

The input module also maintains several direct connections to the CPU. The $\overline{\text{INT}}$ output can be asserted to request an interrupt to the CPU. The $\overline{\text{M1}}$ and $\overline{\text{IORQ}}$ inputs are used by the CPU to acknowledge that the interrupt request has been received. Lastly, the INT_ADDR output is the lower half of a 16-bit address that the CPU uses to determine which address to jump to when handling an interrupt.

I/O Registers

The interrupt module has two I/O registers which can be accessed through the MMU.

Interrupt Enable (IE) – FFFF (R/W)

```
Bit 0: V-Blank  Interrupt Enable  (INT 40h)  (1=Enable)
Bit 1: LCD STAT Interrupt Enable  (INT 48h)  (1=Enable)
Bit 2: Timer    Interrupt Enable  (INT 50h)  (1=Enable)
Bit 3: Serial   Interrupt Enable  (INT 58h)  (1=Enable)
Bit 4: Joypad   Interrupt Enable  (INT 60h)  (1=Enable)
```

Interrupt Flags (IF) – FF0F (Read Only)*

```
Bit 0: V-Blank  Interrupt Request (INT 40h)  (1=Request)
Bit 1: LCD STAT Interrupt Request (INT 48h)  (1=Request)
Bit 2: Timer    Interrupt Request (INT 50h)  (1=Request)
Bit 3: Serial   Interrupt Request (INT 58h)  (1=Request)
Bit 4: Joypad   Interrupt Request (INT 60h)  (1=Request)
```

The lower five bits of the IF register are tied directly to the corresponding REQ pin for each interrupt. The upper three bits are tied to ground.

* In the original GameBoy this register is R/W so games may force an interrupt request at any time. For simplicity, this register is implemented as a read only in FPGABoy. Any writes made to this location are ignored. This is consistent with other GameBoy emulator implementations.

Enabling Interrupts

On CPU reset, interrupts are initially disabled. They must be enabled in two places by the game before they can be used. The first is the global interrupt enable register which can be enabled and disabled using the EI, DI and RETI instructions. These are handled internally within the CPU. Games can also individually enable each of the five device interrupts by writing to the IE register at FFFFh.

Requesting Interrupts

Interrupts are generated by various modules within the system. For each interrupt, a device connects to the interrupt module's corresponding REQ and ACK pins. To request an interrupt the device asserts the REQ signal. This signal should remain asserted until the interrupt is acknowledged by the interrupt module.

When an interrupt is requested by a device, it is not necessarily passed through to the CPU, or immediately acknowledged. If the interrupt has not been enabled by the corresponding bit of the IE register, the request will be ignored until the bit becomes set at a later time. However, if the interrupt is enabled, the interrupt module asserts the $\overline{\text{INT}}$ signal and enters a wait state until the CPU acknowledges the request by the asserting both the $\overline{\text{M1}}$ and $\overline{\text{IORQ}}$ signals. If multiple requests are asserted on the same clock cycle, the interrupt with the lower priority is handled first. The order of precedence corresponds to the order of bits in the IE register, with lower bits having higher priority.

When the CPU interrupt acknowledge occurs, the module places the proper interrupt address on the INT_ADDR bus and the original request is acknowledged to the requesting device by asserting the corresponding ACK signal. This sequence can be represented by the following state transition diagram.

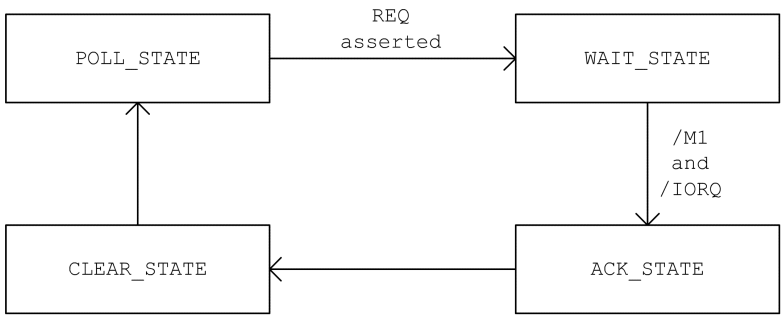


Figure 9: Interrupt module state transition diagram

Assuming the global interrupt enable flag is set, the timing of these states defined by the Z80 specification to be as follows.

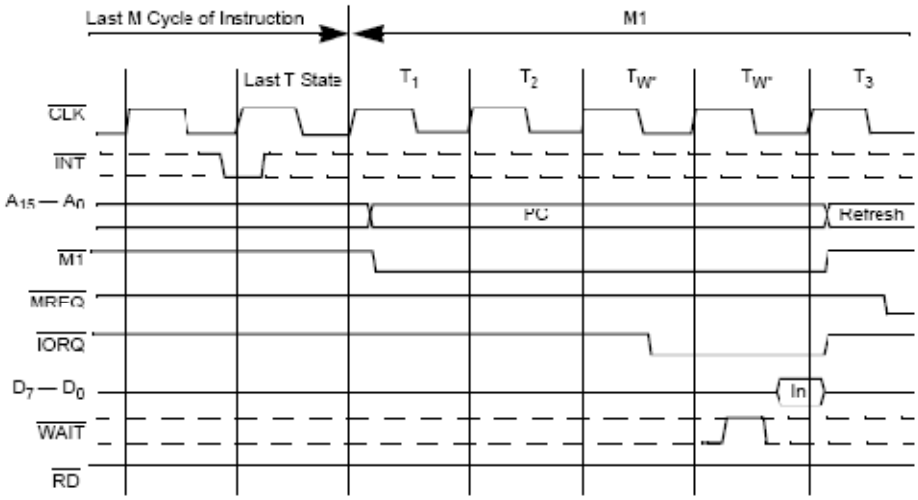


Figure 10: Z80 interrupt request / acknowledge cycle timing (www.zilog.com/docs/z80/um0080.pdf)

Handling Interrupts

Once the interrupt request has been acknowledged, the CPU must handle the interrupt. This is technically done within the CPU and without the help of the interrupt module, but it is included in this section for continuity.

There are three interrupt handling modes that the Z80 CPU can operate in, but the GameBoy does not have an instruction to set this mode programmatically. Instead, the interrupt mode is initialized to the proper value, 2, on a CPU reset. In interrupt mode 2, the CPU can jump to an arbitrary location in memory and begin executing from that address. However the address is not directly specified by the interrupt module. Instead, the 16-bit jump address must be obtained from a table memory. The interrupt module provides the lower 8 bits of the table entry's location and the upper 8 bits are provided by the CPU's 8-bit I register. Like the interrupt mode, the GameBoy's processor does not include instructions for modifying the value

of the I register programmatically so the value must be hard coded on CPU reset. In FPGABoy, we chose to assign this register the value FEh. This allowed us to place the interrupt address table in the only unused area of the GameBoy's memory map, the 96 bytes from 0xFEA0 - 0xFEFF. The table itself is implemented as a small ROM. Each entry corresponds to the 16-bit jump address for the corresponding interrupt. The bytes for each address are stored in little-endian order. The table entry and jump addresses for each of the GameBoy's interrupts are defined by the table below.

Table 5: Interrupt table entry and jump addresses

Interrupt	Table Entry Address	Jump Address
V-Blank	0xFEA0	0x0040
LCDC Stat	0xFEA8	0x0048
Timer Overflow	0xFEB0	0x0050
Serial Transfer	0xFEB8	0x0058
Joypad	0xFEC0	0x0060

Timer Module (Trevor)

FPGABoy implements a timer with four selectable frequencies ranging from 4096 Hz to 262144 Hz. The timer can be controller programmatically by a set of memory mapped registers contained within the module. Games can directly read the timer's counter register, or use the generated timer overflow interrupt to implement time based events. In order to maintain proper timing this module is clocked on the CPU's 4.125 MHz clock, rather than the 33 MHz that most other modules use.

I/O Registers

The timer module has four memory mapped registers which can be accessed through the MMU.

Divider Register (DIV) – FF04 (R/W)

This register is incremented once every 256 clock cycles. Writing any value to this register causes it to reset to zero. Overflow of this register has no effect.

Timer Counter (TIMA) – FF05 (R/W)

This counter is incremented using one of four selectable frequencies, controlled by the TAC register. The timer can also be stopped and started at any time, also via the TAC register. When the value in this counter overflows, a timer overflow interrupt is requested. When this occurs, the value in the TMA register is loaded into the counter, rather than simply resetting to zero.

Timer Modulo (TMA) – FF06 (R/W)

The value to be loaded into the TIMA register when an overflow occurs.

Timer Control (TAC) – FF07 (R/W)

```
Bit 2    - Timer Stop  (0=Stop, 1=Start)
Bits 1-0 - Input Clock Select
           00:   4096 Hz    (~4194 Hz SGB)
           01: 262144 Hz  (~268400 Hz SGB)
           10:  65536 Hz   (~67110 Hz SGB)
           11:  16384 Hz   (~16780 Hz SGB)
```

This register controls the frequency at which the TIMA register increments, as well as enabling and disabling the timer altogether. It has no effect on the frequency or state of the DIV register.

Implementation

The timer module contains four simple dividers, one for each selectable frequency. On each clock cycle, if the selected divider overflows (as indicated by an enable signal), the timer counter (TIMA) is incremented. The divider register (DIV) is always incremented on overflow of the fourth divider (16384 Hz) regardless of the selected frequency.

Video Module (Trevor)

The video module is one of the most complicated aspects of the FPGABoy system. It is primarily responsible for rendering the graphical output of the game. It also performs several other tasks, including the generation of two interrupts, control of 11 memory mapped I/O registers and brokering of access to two separate areas of the memory map, video memory (VRAM) and sprite object attribute memory (OAM). To perform all of these tasks, it uses a 29 state finite state machine, as well as several smaller state machines to control the rendering mode.

I/O Registers

The video module has 11 memory mapped I/O registers that provide information about and control over the rendering state.

LCD Control (LDCD) – FF40 (R/W)

```
Bit 7 - LCD Display Enable             (0=Off, 1=On)
Bit 6 - Window Tile Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
Bit 5 - Window Display Enable          (0=Off, 1=On)
Bit 4 - BG & Window Tile Data Select   (0=8800-97FF, 1=8000-8FFF)
Bit 3 - BG Tile Map Display Select     (0=9800-9BFF, 1=9C00-9FFF)
Bit 2 - OBJ (Sprite) Size              (0=8x8, 1=8x16)
Bit 1 - OBJ (Sprite) Display Enable    (0=Off, 1=On)
Bit 0 - BG Display                     (0=Off, 1=On)
```

The LCDC register controls a number of high level rendering options, including enable/disable flags for each of the three renderable layers, the locations of tile and data maps in VRAM, and the ability to turn the display off altogether.

LCDC Status (STAT) – FF41 (R/W)

```
Bit 6 - LYC=LY Coincidence Interrupt (1=Enable) (Read/Write)
Bit 5 - Mode 2 OAM Interrupt         (1=Enable) (Read/Write)
Bit 4 - Mode 1 V-Blank Interrupt     (1=Enable) (Read/Write)
Bit 3 - Mode 0 H-Blank Interrupt     (1=Enable) (Read/Write)
Bit 2 - Coincidence Flag  (0:LYC<>LY, 1:LYC=LY) (Read Only)
Bit 1-0 - Mode Flag                             (Read Only)
          0: During H-Blank
          1: During V-Blank
          2: During Searching OAM-RAM
          3: During Transfering Data to LCD Driver
```

The STAT register can be used by games to determine the current rendering state, as well as to enable various interrupts. Writes to this register will only affect the upper five bits. Bits 2-0 are set by the video module to provide information about the current rendering mode. For more information about the various video modes see the modes and timing section below.

Background Scroll Y (SCY) – FF42 (R/W)
Background Scroll X (SCX) – FF43 (R/W)

The SCX and SCY registers determine the offset of the upper left corner of the background tile map. The background wraps, so even if these values are non zero the background will still be visible in all areas of the screen.

LCDC Y-Coordinate (LY) – FF44 (Read only)

The LY register contains the current line that is being rendered. If the line count is greater than 153, it resets to zero.

LY Compare (LYC) – FF45 (R/W)

A game may write a value to this register to utilize the LY coincidence interrupt. The video module continuously compares the value in this register to the current line count. If the values match and the LY coincidence interrupt is enabled, an interrupt will be generated.

Background Palette (BGP) – FF47 (R/W)
Object Palette 0 (OBP0) – FF48 (R/W)
Object Palette 1 (OBP1) – FF49 (R/W)

```
Bit 7-6 - Shade for Color Number 3
Bit 5-4 - Shade for Color Number 2
Bit 3-2 - Shade for Color Number 1
Bit 1-0 - Shade for Color Number 0
```

The BGP register defines the four pixel colors that are used for rendering background and window tiles. The OBP0 and OBP1 registers define the color palettes that can be used for

rendering sprites. Each color is a two bit value representing a different shade of grey, with 0 representing white and 3 representing black. To get an actual color from raw pixel data, the video module indexes into the appropriate color palette as if it were an array with four elements.

Window Position Y (WY) – FF4A (R/W)
Window Positon X (WX) – FF4B (R/W)*

The WY and WX registers control the upper left corner of the window tile map. Unlike the background tiles, window tiles do not wrap so this is truly an offset.

*The actual X offset of the window tile map is WX – 7. A value of 7 in this register means that the window tiles will be aligned to the left side of the screen.

Video Modes and Timing

When enabled, the video module continuously cycles through four different modes. Depending on which mode the module is in, some areas of memory may become inaccessible to the CPU. When the mode changes it may also cause several different interrupts to occur.

In order to meet the timing specifications required by the original GameBoy, FPGABoy's video module uses two different clocks. The first is the slower 4.125 MHz clock that runs the CPU. The second is a faster 33 MHz clock. While rendering and memory accesses are performed on the 33 MHz clock, all mode changes occur on the 4.125 MHz clock. The video module maintains two counters, a line count and a "pixel" count, which control the mode and are incremented on each tick of the 4.125 MHz clock. The pixel counter increments 456 times per line and the line count increments 154 times per frame. This results in an overall frame rate of approximately 60 frames per second.

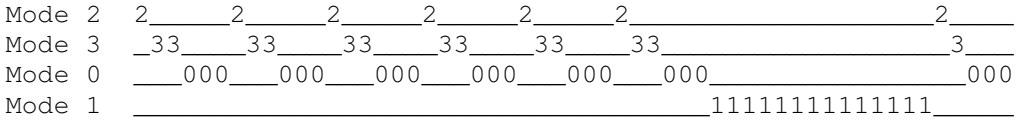


Figure 11: Video mode timing *(Source: http://nocash.emubase.de/pandocs.htm#lcdstatusregister)*

H-Blank – Mode 0

During the H-Blank period the CPU is free to access both the VRAM and OAM areas of memory. This occurs at the end of each rendered line and lasts for 204 cycles. When entering the H-Blank mode, the video module generates an LCDC status interrupt if the corresponding bit of the STAT register is set.

V-Blank – Mode 1

During the V-Blank mode the CPU is free to access both the VRAM and OAM areas of memory. This mode occurs when the line count is greater than 143 and lasts until line count returns to

zero, a total of 4560 cycles. When entering V-Blank mode, two interrupts may be generated. The first is the actual V-Blank interrupt which will always occur. The second is an option LCDC status interrupt which only occurs if the corresponding bit of the STAT register is set.

OAM Lock – Mode 2

During this period the CPU is unable to access the OAM area of memory. Any CPU reads and writes to this area will be ignored. The video module enters this mode for the first 80 cycles of each line. When entering this mode an LCDC status interrupt will be generated if the corresponding bit in the STAT register is set. In reality, FPGABoy does not make use of this mode for rendering so access to memory in the OAM region is only disabled for compatibility.

RAM Lock – Mode 3

During this period the CPU is unable to access both VRAM and OAM. Any CPU reads or writes to either one are ignored. The video module enters this mode immediately after the OAM Lock mode and lasts for 204 cycles. Entering into this mode does not generate an interrupt. All memory accesses required for rendering are performed during this time.

Video Rendering

Just like on the original GameBoy, FPGABoy renders each frame one line at a time. Each line contains 160 rendered pixels and each frame contains 144 rendered lines. During rendering, the current line is stored in a scanline buffer. This buffer allows multiple layers to be rendered separately without generating any actual display data. Once all layers have been rendered, the module reads the data out of the scanline and outputs it one pixel at a time.

While the video mode is controlled on the CPU's 4.125 MHz clock, the actual rendering process occurs on the 33 MHz clock. The principle reason for this was the extremely tight timing that would have been required for rendering with a 4.125 MHz clock. Since each tile and sprite requires been two and six memory access to retrieve the data required for rendering, it is much easier just to overclock the module rather than try to pipeline the process or reduce memory access time.

Video RAM Mappings

The video module uses the following mappings in Video RAM to locate tile data. Some of these regions overlap so special care must be taken to use the correct region as specified by the LCDC register.

Start Address	End Address	Mapping	Condition
0x8000	0x8FFF	BG and Window Tile ID Map	LDCD[4] -> 1
0x8800	0x97FF	BG and Window Tile ID Map*	LCDC[4] -> 0
0x9800	0x9BFF	BG Tile Data	LCDC[3] -> 0
0x9C00	0x9FFF	Background Tile Data	LCDC[3] -> 1

0x9800	0x9BFF	Window Tile Data	LCDC[6] -> 0
0x9C00	0x9FFF	Window Tile Data	LCDC[6] -> 1

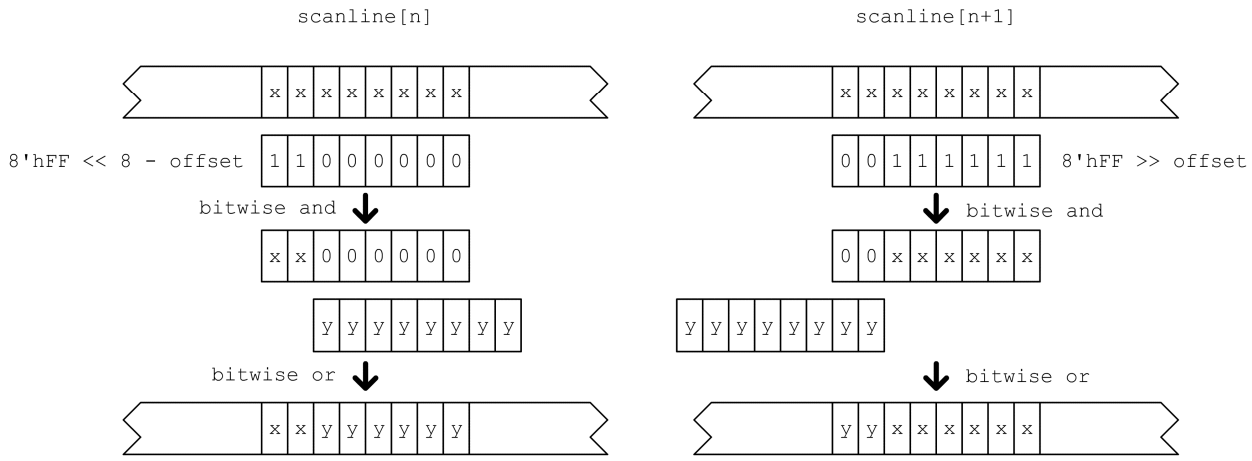
* This region uses signed offsets from -128 to 127 rather than unsigned offsets.

Scanline Buffer

The scanline buffer is implemented as a dual port block RAM, 8 bytes wide and with a depth of 20. Because each pixel is 2 bits, the video module actually has to maintain two scanline buffers, one for the higher order bits of each pixel and one for the lower order bits. These two buffers are combined after rendering is completed to produce the final output.

Using this format for the scanline has a number of important advantages. Each tile that is rendered, be it from a sprite, a window time or a background tile, is 8 pixels across. However, each of these layers can be offset from the left border by a different amount. This means that the video module must either be able to write individual pixels into the scanline, or it must be able to write partial bytes. Of these two options, individual pixels turned out to be extremely slow, both in terms of hardware synthesis and in required clock cycles. Thus, the scanline buffer was implemented as a byte array.

To solve the problem of writing partial bytes, the video module uses a simple masking and shifting technique to when writing non-aligned bytes into the scanline. By using this process we can effectively reduce the number of memory access required to write each byte by a factor of four. A dual port RAM is used so these two memory accesses can be run in parallel. This process is illustrated in the diagram below.



Background and Window Rendering

The background and window layers are both comprised of a grid of 8x8 tiles. The background can have up to 32 tiles in each row and these tiles wrap around if the SCX register is non-zero.

The window tiles do not wrap. Window tiles (if they exist) will always appear on top of the background.

Because the background and window layers have the same basic layout, they can be rendered at the same time. The video module will render 32 columns, one for each 8 pixel tile that can be displayed one each line. For each of these columns it is determined if the window is enabled and visible in that column. If so the window's tile id number (1 byte) is fetched from memory. If it is not and the background is, the background tile id number (1 byte) is fetched. In either case, the resulting tile id number is used to determine the memory address of the actual tile data (2 bytes). These bytes are then fetched and written into the high and low order scanline buffers using the masking / shifting technique. If neither the background nor the window is visible, null bytes are written into the buffer at the corresponding location. When rendering of all 32 background tile columns completed, the video module will begin to render the sprites. A simplified version of the state machine that controls this process is shown below.

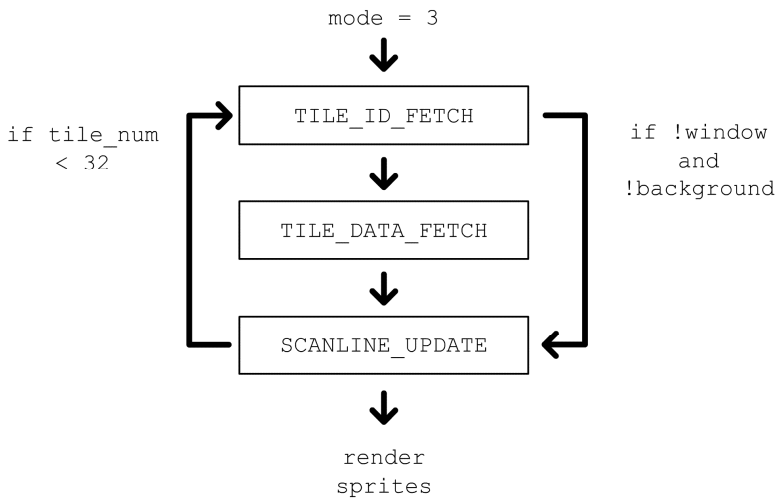


Figure 12: A simplified state diagram for rendering background tiles

Sprite Rendering

Sprites are rendered in a very similar manner to background tiles, but there are a few extra steps that must be taken. First, a lookup into OAM is performed for each sprite to determine its x and y positions (2 bytes). If those coordinates are determined to intersect with the line currently being rendered, two more attributes (2 bytes) are fetched from OAM which correspond to the sprite's location in the tile data map and a number of cosmetic attributes which determine which object palette should be used to color the sprite, whether that sprite appears above or behind the background and if the sprite should be flipped in either the x or y direction. Once these have been obtained, the actual sprite pixel data can be retrieved from VRAM (2 bytes). If the sprite does not intersect with the current line, the sprite is ignored.

Once all the sprite data has been read from memory, the sprite must be drawn. Unfortunately, the sprite cannot be drawn directly into the scanline. Instead, the video module must first determine the color of each pixel in the sprite by indexing into the appropriate object palette. The pixel value for the same location in the scanline must also be read and compared with that of the sprite. Depending on the attributes of the sprite and the color of the background pixel, the sprite's pixel may be masked in favor of the background pixel. This step is repeated for each pixel and the results are stored in a temporary 8-bit register until the high and low bits for all 8 pixels have been computed. At this point the data bytes are written into the scanlines, again using the shifting / masking technique. A simplified state diagram for this process is shown below.

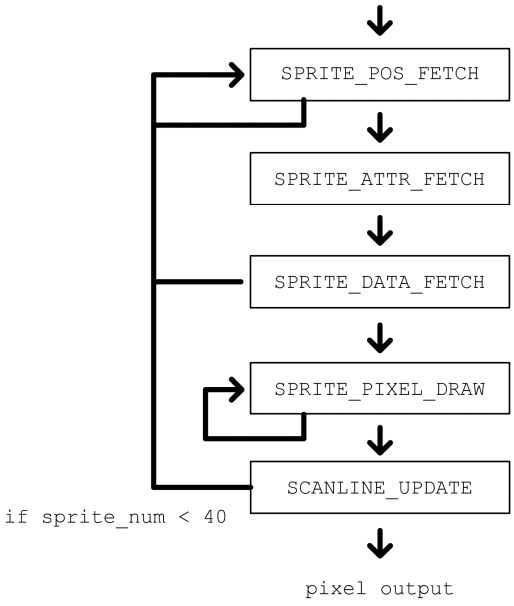


Figure 13: A simplified state diagram for rendering sprites

Pixel Output

Once the entire scanline has been computed, the video module outputs the results, one pixel at a time. This is done by reading out a single byte from the high and low order scanlines, and combining them. The only tricky part about this is that the bits have to be read backwards in order to achieve a correct image. This is because the furthest left pixel in each byte is actually the highest bit.

Video Converter (Oleg)

The video converter module interprets the output generated by the video module and converts it into a form such that it can be passed on to the VGA controller. Specifically, it converts from the FPGABoy's 160x144x2 resolution into VGA's 640x480x24 resolution. To perform the conversion smoothly, the module implements a double buffering system. That is, while the VGA controller is fetching color data from one buffer, the video module is writing into another.

This minimizes the amount of time during which there is no image on the screen and reduces flickering.

Double Buffering

The double buffering is implemented through the use of two different block RAMs, each of which is big enough to store 160x144x2 bits. An internal register keeps state of which of the two block RAMs acts as the front buffer and the inputs and outputs into the rams are multiplexed appropriately. Furthermore, the clocks that drive the block RAMs are also multiplexed as the pixel clock from the VGA controller that outputs to the screen and the clock from the video module are different. The difference in clocks poses no problem since only one type of operation happens for each clock. That is, the video module clock only times the writes while the pixel clock controls the timing of the reads. Since these operations happen on different memories no conflicts arise.

The RAM that is used as the back buffer takes as an input the address determined by the pixel count and line count that is output by the video module. The value that is written into that address also comes from the video module. When the video module completes a whole frame, it raises a vsync signal which signifies to the video converter that it is time to flip buffers. Thus the video converter flips the internal switch and swaps the block memories. Now the video module is able to restart writing into the new back buffer.

While the back buffer is being written by the vide module, the VGA controller reads from the front buffer. The address being read from is determined by the line and pixel count output by the VGA controller. In order to center the image and achieve scaling, the actual address is modified by dividing the pixel and line count by two and shifting it by an offset in the x and y directions. The shift amount is determined by two parameters in the module so the output can be placed anywhere on the screen. The VGA controller keeps reading from whichever front buffer is selected by the internal register to present a seamless image on the screen.

VGA Controller

The VGA controller used by the FPGABoy is identical to the one implemented in Lab 4. As an input it takes a clock and generates the proper output signals such as a pixel clock, delayed hsync and vsync, a blank signal and the line and pixel count for the screen. As stated above, the converter module uses the line and pixel counter to determine which address to read from the front buffer and to display on the screen. This component also facilitates the conversion of the 2-bit FPGABoy color to a 24-bit RGB color value. Since the game boy only supports four colors, the module remaps them onto different shades of gray. Table 6 illustrates the color mappings used between the FPGABoy output and the 24-bit VGA color.

Table 6. FPGABoy to RGB color mapping

FPGABoy Color	Color	RGB Color(24-Bit)
0x00		0xFFFFFF
0x01		0xAAAAAA
0x10		0x555555
0x11		0x000000

The disjoint nature of the two halves of the video converter made it essential in the debugging process. The main reason was that the VGA module could output the contents of the buffer even when the processor or the video module was halted, thus it made the video module much easier to test.

Cartridge Module (Oleg)

The cartridge module is intended to interface the FPGABoy with real game cartridges. The module has two components, one physical and one that was implemented solely in Verilog. The cartridge module is simply a bus that runs from the user I/O pins on the lab kit to the memory map module. Because of the block RAM abstraction used inside the memory module, it is trivial to use the cartridge module to remap the cartridge images stored internally on the block RAM. All that is necessary is to remap the first 32KB of address space along with read and write control signals onto the user I/O pins instead of the block RAM and the interfacing with the external cartridge is complete.

Cartridge Interface

In order to construct the cartridge interface we obtained an original GameBoy handheld device. After disassembling it the back plate with the cartridge slot was extracted and can be seen in Figure 14. Next, we soldered 32 wires to the pins that came out from the cartridge interface on the back of the GameBoy. Next these wires were connected to +5V, GND, and the user2[29:0] port.

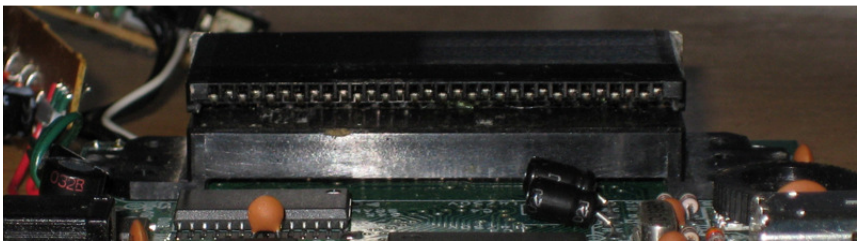


Figure 14. GameBoy back plate cartridge interface

This allowed us to attach real cartridges to the FPGABoy and insert them into our system. Unfortunately, due to lack of time we were unable to get this module to interface properly with the rest of the FPGABoy system. This is definitely something that can be worked upon in the future.

Testing and Debugging

One of the most difficult aspects of this project was to perform most debugging tasks. Because of the monolithic system structure, almost all the modules, except the video converter and input module had to be tested simultaneously. This made it very difficult to track down bugs as they could have originated from any component in the system. Furthermore, we had to aggregate our fixes and changes carefully in order to make sure we don't waste time re-synthesizing the project needlessly.

The most vital component of our testing and debugging system was our access to a high quality emulator [3]. It's ability to disassemble code, display contents of video memory, set breakpoints and view any of the GameBoy state imaginable provided an excellent reference point for finding any issues with our FPGABoy implementation. It is likely that without such a powerful tool this project would've resulted in failure.

Input Converter Module

The testing for the input converter module was done by first performing simulations in ModelSim. This was vital in order to verify that the output of the latch and clock signals were properly timed. Once that was tested, we implemented a simplified version of the lab kit test bench which mapped the eight button states to the LED's on the lab kit. After some toying with the timing of the data signal that was received from the game pad we were successful in making the LED lights light up when the respective button was pressed. This signified proper operation of the input converter module.

Memory Controller

The memory map module was tested over the course of the whole project. This was because the memory map was one of the first modules implemented as it was vital to the operation of the system. Because of that there were one or two incremental bug fixes, but overall it worked near perfectly from the start.

The second set of testing on the memory map module happened towards the end of the project when DMA was added to the system. This was tested by running a GameBoy game and placing a breakpoint when a routine in high RAM was entered. Execution in high RAM signifies the beginning of a DMA sequence. In fact, we determined the exact entry point of the procedure that performs the DMA initialization. Then, using the hex display we were able to output the data that was on the DMA bus and the address from which it was copying. When those values matched up with the reference as executed by a GameBoy emulator, we knew that the DMA operation was working properly. Thus the debugging of the memory controller was complete.

Video Converter Module

Much like the input converter module, the video converter was tested separately. Since our own VGA converter from the previous lab was utilized, there was some assurance that it worked well. Thus, all that was needed to be tested was the buffer flipping on vsync and the writing into the back buffer. A small test harness was established in its own lab kit test bench that would draw vertical stripes on the screen using different colors. Then, scan-line positioning was tested by displaying alternating color horizontal stripes. This type of testing revealed all the display artifacts. Using this information, most of the timing bugs that caused them were tracked down. The result was a very smooth converter module with hardly any flicker.

Video Module

The video module was initially tested using ModelSim. Contrived scenarios using preset register values and an emulator dump of video RAM were used to render a single line. The state and eventual pixel output was compared against a set of expect values to determine validity. Once the system had progressed to the point of mobile sprites and was actually playable, testing of the video module was don't mostly by play testing on the FPGA.

Game-play

One of the more fun aspects of testing was towards the end of the project development cycle. That testing involved playing the game and looking for various errors. Luckily, by this time our system was stable enough that there no visible errors that were caused by our modules. The other part of game play testing was to check our compatibility with other GameBoy games. This involved obtaining a variety of other images for testing and implementing a way to switch between games without having to re-synthesize.

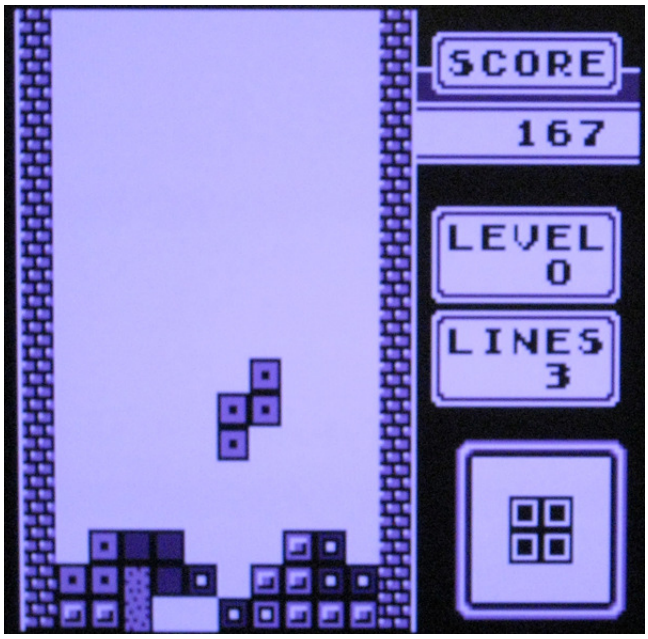


Figure 15. Game-play testing

As a result of game play testing we discovered that most of the other games had many compatibility issues. These issues were primarily caused by our lack of support for many of the advanced GameBoy features and thus were labeled outside of the scope of this project. An interesting point is that all of the games that were tested were from 1989 but only the first game, Tetris, was fully functional. This revealed the rate at which the advanced features of the GameBoy were exploited such a short time after its release. By implementing the rest of the GameBoy specification, the compatibility can definitely be improved in the future.

Conclusion and Future Work

Overall, the FPGABoy was a great success. We were able to implement a working system that is able to run and play GameBoy cartridges. There are significant compatibility issues with most games, but others such as Tetris appear to work near flawlessly. In the future there is a significant amount of work that could be done to improve the FPGABoy's compatibility, both in terms of rendering and fixing any remaining bugs in the TV80 core. Another step to facilitate better compatibility would adding support for external game cartridges. Our attempt to get this working proved it to be somewhat difficult. There are also two major GameBoy features that were not implemented in FPGABoy. Adding support for sound and serial connections would make this a truly complete system.

References

[1] kOOPa. nocash. "Everything You Always Wanted To Know About GAMEBOY," Pan Docs, 2001, <http://nocash.emubase.de/pandocs.htm#joypadinput>

[2] Neviksti. "Gameboy Bootstrap ROM," Gameboy Developement Wiki, 2008, <http://gameboygenius.8bitcollective.com/gbdev-wiki/articles/Gameboy_Bootstrap_ROM>

[3] "BGB Homepage," <http://bgb.bircd.org/>

[4] "TV80 Home Page." 15 May 2009 <http://ghutchis.googlepages.com/tv80homepage>.

Appendix A: FPGABoy modules

labkit.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module for Lab 4 (Spring 2007)
//
//
// Created: March 15, 2007
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////////

module labkit (
               vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
               vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
               vga_out_vsync,

               clock_27mhz, //clock1, clock2,

               disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_in,

               button0, button1, button2, button3, button_enter, button_right,
               button_left, button_down, button_up,

               switch,

               led, user1, user2);

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
          vga_out_hsync, vga_out_vsync;

   input  clock_27mhz;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter, button_right,
          button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

        inout wire[31:0] user1;
        inout wire[31:0] user2;

   //////////////////////////////////////////////////////////////////////////////////
   //
   // Final Project Components
   //
   //////////////////////////////////////////////////////////////////////////////////

   //
   // Core Clock: ~4.194304 MHz
   //

   wire coreclk, core_clock;
   DCM core_clock_dcm (.CLKIN(clock_27mhz), .CLKFX(coreclk));
        defparam core_clock_dcm.CLKFX_DIVIDE = 9;
        defparam core_clock_dcm.CLKFX_MULTIPLY = 11;
        defparam core_clock_dcm.CLK_FEEDBACK = "NONE";
        //defparam core_clock_dcm.CLKIN_PERIOD = 37.037;
   BUFG core_clock_buf (.I(coreclk), .O(core_clock));

        wire pixelclk, pixel_clock;
        DCM pixel_clock_dcm (.CLKIN(clock_27mhz), .CLKFX(pixelclk));
        defparam pixel_clock_dcm.CLKFX_DIVIDE = 6;
```

```
        defparam pixel_clock_dcm.CLKFX_MULTIPLY = 7;
        defparam pixel_clock_dcm.CLK_FEEDBACK = "NONE";
    BUFG pixel_clock_buf (.I(pixelclk), .O(pixel_clock));

        wire reset_init, reset;
        SRL16 reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(reset_init),
                 .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
        defparam reset_sr.INIT = 16'hFFFF;

        //
        // Hex Display
        //

        wire [63:0] display_data;

        display_16hex hex_display (reset, core_clock, display_data,
               disp_blank, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_out);

        //
        // Buttons + Switches
        //

        wire reset_sync, step_sync, left_sync, right_sync, save_sync, edit_sync;
        wire step_enable, breakpoint_enable, debug_enable;

        debounce debounce_reset_sync(reset_init, core_clock, !switch[0], reset_sync);
        debounce debounce_step_sync(reset_init, core_clock, switch[3], step_sync);
        debounce debounce_left_sync(reset_init, core_clock, !button_left, left_sync);
        debounce debounce_right_sync(reset_init, core_clock, !button_right, right_sync);
        debounce debounce_save_sync(reset_init, core_clock, !button_up, save_sync);
        //debounce debounce_edit_sync(reset_init, core_clock, switch[3], edit_sync);


        assign edit_sync = 1;

        debounce debounce_step_enable(reset_init, core_clock, switch[1], step_enable);
        debounce debounce_breakpoint_enable(reset_init, core_clock, switch[2],
breakpoint_enable);

        assign reset = (reset_init || reset_sync);

        //
        // CPU I/O Pins
        //

        wire reset_n, wait_n, int_n, nmi_n, busrq_n; // cpu inputs
        wire m1_n, mreq_n, iorq_n, rd_n, wr_n, rfsh_n, halt_n, busak_n; // cpu outputs

        wire[15:0] A; // cpu addr
        wire[7:0] di; // cpu data in
        wire[7:0] do; // cpu data out

        //
        // Breakpoint Module
        //

        wire[15:0] breakpoint_addr;
        wire[15:0] breakpoint_disp;

        breakpoint_module breakpoint(
               .reset(reset_init),
               .clock(core_clock),
               .button_left(left_sync),
               .button_right(right_sync),
               .breakpoint_save(save_sync),
               .breakpoint_edit(edit_sync),
               .bits(switch[7:4]),
               .display(breakpoint_disp),
               .breakpoint_addr(breakpoint_addr)
        );
```

```
wire[4:0] gb_state;

assign debug_enable =
       step_enable ||
       (breakpoint_enable && A == breakpoint_addr && !m1_n && (!iorq_n || !mreq_n)) ||
       (!button_down && gb_state == breakpoint_addr);

wire clock;
BUFGMUX clock_mux(.S(debug_enable), .O(clock),
                     .I0(core_clock), .I1(step_sync));

//assign clock = clock_27mhz;

//
// TV80 CPU
//

reg[2:0] clock_divider; // overflows every 8 cycles

wire clock_4mhz;
BUFG divided_clock_buf (.I(clock_divider[2]), .O(clock_4mhz));
//assign clock_4mhz = clock_27mhz;

wire [15:0] AF;
wire [15:0] BC;
wire [15:0] DE;
wire [15:0] HL;
wire [15:0] PC;
wire [15:0] SP;
wire IntE_FF1;
wire IntE_FF2;
wire INT_s;

tv80s tv80_core(
       .reset_n(reset_n),
       .clk(clock_4mhz),
       .wait_n(wait_n),
       .int_n(int_n),
       .nmi_n(nmi_n),
       .busrq_n(busrq_n),
       .m1_n(m1_n),
       .mreq_n(mreq_n),
       .iorq_n(iorq_n),
       .rd_n(rd_n),
       .wr_n(wr_n),
       .rfsh_n(rfsh_n),
       .halt_n(halt_n),
       .busak_n(busak_n),
       .A(A),
       .di(di),
       .do(do),
       .ACC(AF[15:8]),
       .F(AF[7:0]),
       .BC(BC),
       .DE(DE),
       .HL(HL),
       .PC(PC),
       .SP(SP),
       .IntE_FF1(IntE_FF1),
       .IntE_FF2(IntE_FF2),
       .INT_s(INT_s)
);

assign reset_n = !reset;
assign wait_n = 1'b1;
assign nmi_n = 1'b1;
assign busrq_n = 1'b1;

//
// Status LEDs + Core Debugging
```

```
//

assign display_data[63:48] = A[15:0];
assign display_data[47:40] = di[7:0];
assign display_data[39:32] = do[7:0];

assign led[0] = m1_n;
assign led[1] = mreq_n;
assign led[2] = iorq_n;
assign led[3] = int_n;
assign led[4] = halt_n;
assign led[5] = reset_n;
assign led[6] = rd_n;
assign led[7] = wr_n;

//
// MMU
//

wire[15:0] A_video;
wire[7:0] di_video;
wire rd_n_video;
wire wr_n_video;
wire[15:0] current_dma_addr;
wire[15:0] current_dma_data;

wire mem_enable_interrupt;
wire mem_enable_timer;
wire mem_enable_video;
wire mem_enable_sound;
wire mem_enable_input;

wire[7:0] do_mmu;
wire[7:0] do_interrupt;
wire[7:0] do_timer;
wire[7:0] do_video;
wire[7:0] do_sound;
wire[7:0] do_input;

memory_controller mmu(
        .reset(reset),
        .clock(clock),
        .rd_n(rd_n),
        .wr_n(wr_n),
        .A(A),
        .di(do), // TODO: make di and di names more specific
        .do_interrupt(do_interrupt),
        .do_timer(do_timer),
        .do_video(do_video),
        .do_sound(do_sound),
        .do_input(do_input),
        .A_video(A_video),
        .di_video(di_video),
        .rd_n_video(rd_n_video),
        .wr_n_video(wr_n_video),
        .button0(button0),
        .button1(button1),
        .button2(button2),
        .button3(button3),
        .button_enter(button_enter),
        .enable_interrupt(mem_enable_interrupt),
        .enable_timer(mem_enable_timer),
        .enable_video(mem_enable_video),
        .enable_sound(mem_enable_sound),
        .enable_input(mem_enable_input),
        .do(do_mmu),
        .current_dma_addr(current_dma_addr),
        .current_dma_data(current_dma_data),

        .cart_wr_n(user2[2]),
        .cart_rd_n(user2[3]),
```

```
        .cart_mreq_n(user2[4]),
        .cart_A(user2[20:5]),
        .cart_data(user2[28:21]),
        .cart_reset_n(user2[29])
);

//
// Interrupt Module
//

wire[4:0] int_req;
wire[4:0] int_ack;
wire[7:0] jump_addr;

interrupt_module interrupt(
        .reset(reset),
        .clock(clock),
        .mem_enable(mem_enable_interrupt),
        .rd_n(rd_n),
        .wr_n(wr_n),
        .m1_n(m1_n),
        .int_n(int_n),
        .iorq_n(iorq_n),
        .int_ack(int_ack),
        .int_req(int_req),
        .jump_addr(jump_addr),
        .A(A),
        .di(do),
        .do(do_interrupt)
);

assign int_req[3] = 0;
//assign int_req[4] = 0;

assign di = (!iorq_n && !m1_n) ? jump_addr : do_mmu;

//
// Timer Module
//

timer_module timer(
        .reset(reset),
        .clock(clock_4mhz),
        .mem_enable(mem_enable_timer),
        .rd_n(rd_n),
        .wr_n(wr_n),
        .int_ack(int_ack[2]),
        .int_req(int_req[2]),
        .A(A),
        .di(do),
        .do(do_timer)
);

//
// Video Module
//

wire gb_hsync, gb_vsync;
wire[1:0] gb_pixel_data;
wire[7:0] gb_pixel_count;
wire[7:0] gb_line_count;
wire[7:0] gb_pixel_cnt; // 0-454
wire[7:0] sprite_x_pos;
wire[7:0] sprite_y_pos;
wire[6:0] sprite_num;
wire[7:0] sprite_data1;
wire[7:0] sprite_data2;
wire[1:0] sprite_pixel;
wire[1:0] bg_pixel;
wire[7:0] oam_addrA;
wire gb_pixel_we;
```

```
video_module video(
        .reset(reset),
        .clock(clock),
        .mem_enable(mem_enable_video),
        .rd_n(rd_n_video),
        .wr_n(wr_n_video),
        .int_vblank_ack(int_ack[0]),
        .int_lcdc_ack(int_ack[1]),
        .A(A_video),
        .di(di_video),
        .int_vblank_req(int_req[0]),
        .int_lcdc_req(int_req[1]),
        .hsync(gb_hsync),
        .vsync(gb_vsync),
        .pixel_count(gb_pixel_cnt),
        .pixel_data(gb_pixel_data),
        .pixel_data_count(gb_pixel_count),
        .pixel_we(gb_pixel_we),
        .sprite_x_pos(sprite_x_pos),
        .sprite_y_pos(sprite_y_pos),
        .sprite_num(sprite_num),
        .sprite_data1(sprite_data1),
        .sprite_data2(sprite_data2),
        .sprite_pixel(sprite_pixel),
        .bg_pixel(bg_pixel),
        .oam_addrA(oam_addrA),
        .line_count(gb_line_count),
        .state(gb_state),
        .do(do_video)
);

wire fb;

video_converter converter(
        .reset(reset),
        .clock(pixel_clock),
        .gb_clock(clock),
        .pixel_data(gb_pixel_data),
        .gb_pixel_count(gb_pixel_count),
        .gb_line_count(gb_line_count),
        .gb_hsync(gb_hsync),
        .gb_vsync(gb_vsync),
        .gb_we(gb_pixel_we),
        .color( {vga_out_red, vga_out_green, vga_out_blue} ),
        .sync_b(vga_out_sync_b),
        .blank_b(vga_out_blank_b),
        .pixel_clock(vga_out_pixel_clock),
        .hsync(vga_out_hsync),
        .vsync(vga_out_vsync)
);

wire[7:0] bs;

input_controller ic(
        .reset(reset),
        .clock(clock_27mhz),
        .controller_data(user1[2]),
        .controller_latch(user1[1]),
        .controller_clock(user1[0]),
        .mem_enable(mem_enable_input),
        .rd_n(rd_n),
        .wr_n(wr_n),
        .int_ack(int_ack[4]),
        .int_req(int_req[4]),
        .A(A),
        .di(do),
        .do(do_input),
        .button_state(bs)
);
```

```
	reg[2:0] disp_select;

	assign display_data[31:0] =
		(disp_select == 0) ? { AF, BC } :
		(disp_select == 1) ? { DE, HL } :
		//(disp_select == 2) ? { 3'b0, gb_state, gb_line_count, 3'b0, gb_pixel_we, 3'b0,
gb_vsync, 3'b0, !vga_out_vsync, 3'b0, fb } :
		//(disp_select == 2) ? { 3'b0, bs[7], 3'b0, bs[6], 3'b0, bs[5], 3'b0, bs[4],
		//                                                              3'b0, bs[3], 3'b0,
bs[2], 3'b0, bs[1], 3'b0, bs[0] } :
		(disp_select == 2) ? { PC, SP } :
		(disp_select == 3) ? { 13'b0, IntE_FF2, IntE_FF1, INT_s, 3'b0, int_ack, 3'b0,
int_req } :
		//(disp_select == 3) ? { 3'b0, gb_state, 1'b0, sprite_num, sprite_y_pos,
oam_addrA} :
		{ 16'b0, breakpoint_disp };

	always @(posedge clock_27mhz)
	begin
		if (reset_init)
		begin
			disp_select <= 0;
		end
		else
		begin
			if (!button0)
				disp_select <= 0;
			if (!button1)
				disp_select <= 1;
			if (!button2)
				disp_select <= 2;
			if (!button3)
				disp_select <= 3;
			if (!button_enter)
				disp_select <= 4;
		end
	end

	always @(posedge clock)
	begin
		if (reset_init)
		begin
			clock_divider <= 0;
		end
		else
		begin
			clock_divider <=
				(switch[6]) ?
					clock_divider + 4 :
					clock_divider + 1;
		end
	end

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Debounce/Synchronize module
//
//
// Use your system clock for the clock input to produce a synchronous,
// debounced output
//
//////////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock, noisy, clean);
   parameter DELAY = 270000;   // .01 sec with a 27Mhz clock
   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
```

```
   reg new, clean;

   always @(posedge clock)
     if (reset)
       begin
          count <= 0;
          new <= noisy;
          clean <= noisy;
       end
     else if (noisy != new)
       begin
          new <= noisy;
          count <= 0;
       end
     else if (count == DELAY)
       clean <= new;
     else
       count <= count+1;

endmodule
```

breakpoint_module.v

```
`default_nettype none
`timescale 1ns / 1ps

module breakpoint_module(
        input wire reset,
        input wire clock,
        input wire button_left,
        input wire button_right,
        input wire breakpoint_save,
        input wire breakpoint_edit,
        input wire[3:0] bits,
        output reg[15:0] display,
        output reg[15:0] breakpoint_addr);

        wire blink_toggle; // 1 sec toggle w/ 27 MHz clock
        divider blink_divider(reset, clock, blink_toggle);

        reg[1:0] cursor;
        reg button_wait;
        reg blink_enable;

        always @(posedge clock)
        begin
                if (reset)
                begin
                        cursor <= 0;
                        button_wait <= 0;
                        blink_enable <= 0;
                        breakpoint_addr <= 0;
                end
                else
                begin
                        if (breakpoint_edit)
                        begin
                                if (button_left && !button_wait)
                                begin
                                        cursor <= cursor + 1;
                                        button_wait <= 1;
                                end
                                else if (button_right && !button_wait)
                                begin
                                        cursor <= cursor - 1;
                                        button_wait <= 1;
                                end
                                else if (!button_left && !button_right)
                                        button_wait <= 0;
```

```
                        if (blink_toggle)
                                blink_enable <= !blink_enable;

                        if (breakpoint_save)
                                case (cursor)
                                        2'b00: breakpoint_addr[3:0] <= bits;
                                        2'b01: breakpoint_addr[7:4] <= bits;
                                        2'b10: breakpoint_addr[11:8] <= bits;
                                        2'b11: breakpoint_addr[15:12] <= bits;
                                endcase
                        end
                        else
                        begin
                                blink_enable <= 0;
                        end
                end

                assign display =
                        ((cursor == 0 && blink_enable) ? 4'hF : breakpoint_addr[3:0]) |
                        ((cursor == 1 && blink_enable) ? 4'hF : breakpoint_addr[7:4]) << 4 |
                        ((cursor == 2 && blink_enable) ? 4'hF : breakpoint_addr[11:8]) << 8 |
                        ((cursor == 3 && blink_enable) ? 4'hF : breakpoint_addr[15:12]) << 12;

        end

endmodule
```

memory_controller.v

```
`default_nettype none
`timescale 1ns / 1ps

module memory_controller(
        input wire reset,
        input wire clock,
        input wire rd_n,
        input wire wr_n,
        input wire [15:0] A,
        input wire [7:0] di, // should be do from cpu
        input wire [7:0] do_interrupt,
        input wire [7:0] do_timer,
        input wire [7:0] do_video,
        input wire [7:0] do_sound,
        input wire [7:0] do_input,
        input wire button0,
        input wire button1,
        input wire button2,
        input wire button3,
        input wire button_enter,

        output wire [15:0] cart_A,
        output wire cart_wr_n,
        output wire cart_rd_n,
        output wire cart_mreq_n,
        output wire cart_reset_n,
        inout wire [7:0] cart_data,

        output wire [15:0] A_video,
        output wire [7:0] di_video,
        output wire rd_n_video,
        output wire wr_n_video,
        output wire enable_interrupt,
        output wire enable_timer,
        output wire enable_video,
        output wire enable_sound,
        output wire enable_input,
        output wire [7:0] do, // should be mux'd to cpu's di
        output wire [15:0] current_dma_addr,
        output wire [7:0] current_dma_data
        );
```

```
        // internal data out pins
        wire [7:0] do_boot_rom;
        wire [7:0] do_internal_ram;
        wire [7:0] do_high_ram;
        wire [7:0] do_int_table;

        // internal r/w enables
        wire enable_boot_rom;
        wire enable_internal_ram;
        wire enable_high_ram;
        wire enable_cartridge;
        wire enable_int_table;

        // remapped addresses
        wire [12:0] A_internal_ram;
        wire [6:0] A_high_ram;
        wire [6:0] A_int_table;

        // when $01 gets written into $FF50 the ROM is disabled
        reg rom_enable;

        // dma stuff
        reg dma_enable;
        reg dma_rd_n_video;
        reg dma_wr_n_video;
        reg[7:0] dma_addr;
        reg[7:0] dma_counter;
        wire[7:0] do_dma;

        // if dma is enabled, make address source as the dma address
        assign current_dma_addr = {dma_addr, 8'b0} + dma_counter;
        wire[15:0] A_temp = (dma_enable) ? { dma_addr, 8'b0 } + dma_counter : A;

        boot_rom br(A, clock, do_boot_rom);
        interrupt_table it(A, clock, do_int_table);
        internal_ram ir(A_internal_ram, clock, di, do_internal_ram, enable_internal_ram,
dma_enable ? 1 : (wr_n));
        high_ram hr(A_high_ram, clock, di, do_high_ram, enable_high_ram, dma_enable ? 1 :
(wr_n));

        reg[2:0] cart_select;

        wire [7:0] do_cartridge;
        wire [7:0] do_cartridge1;
        wire [7:0] do_cartridge2;
        wire [7:0] do_cartridge3;
        wire [7:0] do_cartridge4;
        wire [7:0] do_cartridge_ext;

        assign cart_data = (!cart_wr_n && !cart_mreq_n) ? di : 8'hZZ;
        assign do_cartridge_ext = (!cart_rd_n && !cart_mreq_n) ? cart_data : 8'hFF;

        // cartridges
        tetris_rom cr1(A_temp, clock, do_cartridge1);
        adjustris_rom cr2(A_temp, clock, do_cartridge2);
        sokoban_rom cr3(A_temp, clock, do_cartridge3);
        kwirk_rom cr4(A_temp, clock, do_cartridge4);

        // broken carts
        //sokoban_rom cr(A_temp, clock, do_cartridge);
        //kwirk_rom cr(A_temp, clock, do_cartridge);

        assign do_cartridge =
                (cart_select == 0) ? do_cartridge1 :
                (cart_select == 1) ? do_cartridge2 :
                (cart_select == 2) ? do_cartridge3 :
                (cart_select == 3) ? do_cartridge4 :
                (cart_select == 4) ? do_cartridge_ext : do_cartridge1;

        assign cart_A = A_temp;
```

```
assign cart_wr_n = dma_enable ? 1 : (wr_n);
assign cart_rd_n = rd_n;
assign cart_mreq_n = !(enable_cartridge && cart_select == 4);
assign cart_reset_n = !reset;

parameter DMA_WR_ENABLE = 0;
parameter DMA_WAIT = 1;
parameter DMA_WR_DISABLE = 2;
parameter DMA_COUNT = 3;
reg[1:0] dma_state;

always @ (posedge clock)
begin
	if(reset) begin
		rom_enable <= 1;
		dma_enable <= 0;
		dma_state <= DMA_WR_ENABLE;
		dma_counter <= 0;
		dma_rd_n_video <= 1;
		dma_wr_n_video <= 1;

		if (!button0)
			cart_select <= 0;
		else if (!button1)
			cart_select <= 1;
		else if (!button2)
			cart_select <= 2;
		else if (!button3)
			cart_select <= 3;
		else if (!button_enter)
			cart_select <= 4;
		else
			cart_select <= 0;
	end else begin

		if(dma_enable) begin
			case(dma_state)
			DMA_WR_ENABLE:
				begin
					dma_rd_n_video <= 1;
					dma_wr_n_video <= 0;
					dma_state <= DMA_WAIT;
				end
			DMA_WAIT: dma_state <= DMA_WR_DISABLE;
			DMA_WR_DISABLE:
				begin
					dma_rd_n_video <= 1;
					dma_wr_n_video <= 1;
					dma_state <= DMA_COUNT;
				end
			DMA_COUNT:
				begin
					dma_counter <= dma_counter + 1;
					if(dma_counter >= 159) begin
						dma_enable <= 0;
					end else begin
						dma_state <= DMA_WR_ENABLE;
					end
				end
			endcase

			// read from dma_addr * 100h
			// write to oam at FE00 + dma_counter
			// clamp dma counter at after transfering 160 bytes
			/*if(dma_counter < 160) begin
				dma_counter <= dma_counter + 1;

			end else begin
				dma_enable <= 0;
				dma_rd_n_video <= 1;
				dma_wr_n_video <= 1;
```

```
                end*/
            end

            if (!rd_n)
            begin

            end
            else if (!wr_n)
            begin

                case(A)
                    // dma
                    16'hFF46:
                    begin
                        if (!dma_enable)
                        begin
                            dma_enable <= 1;
                            dma_counter <= 0;
                            dma_addr <= di;
                            dma_state <= DMA_WR_ENABLE;
                        end
                    end

                    16'hFF50: if (di == 1) rom_enable <= 0;
                endcase
            end

        end
end

assign enable_boot_rom = rom_enable && A < 16'h0100;
assign enable_cartridge = A_temp >= 16'h0000 && A_temp < 16'h8000;
assign enable_internal_ram =
        (A_temp >= 16'hC000 && A_temp < 16'hE000) ||
        (A_temp >= 16'hE000 && A_temp < 16'hFE00); // echo of internal ram
assign enable_high_ram = A >= 16'hFF80 && A < 16'hFFFF;
assign enable_interrupt = A == 16'hFF0F || A == 16'hFFFF;
assign enable_timer = A >= 16'hFF04 && A <= 16'hFF07;
assign enable_video = (dma_enable) || // if dma is on, enable video
        (A >= 16'h8000 && A < 16'hA000) || // vram
        (A >= 16'hFE00 && A < 16'hFEA0) || // oam
        (A >= 16'hFF40 && A <= 16'hFF4B && A != 16'hFF46); // registers (except for DMA)
assign enable_sound = A >= 16'hFF10 && A <= 16'hFF3F; // there are some gaps here
assign enable_input = A == 16'hFF00;
assign enable_int_table = A >= 16'hFEA0 && A < 16'hFF00;

// remap addresses
assign A_internal_ram = (A_temp >= 16'hE000) ? A_temp - 16'hE000 : A_temp - 16'hC000;
assign A_high_ram = A - 16'hFF80;
assign A_int_table = A - 16'hFEA0;

assign do = (dma_enable) ? ((enable_high_ram) ? do_high_ram : 8'hFF) : (
        (enable_boot_rom) ? do_boot_rom :
        (enable_cartridge) ? do_cartridge :
        (enable_internal_ram) ? do_internal_ram :
        (enable_high_ram) ? do_high_ram :
        (enable_interrupt) ? do_interrupt :
        (enable_timer) ? do_timer :
        (enable_video) ? do_video :
        (enable_sound) ? do_sound :
        (enable_input) ? do_input :
        (enable_int_table) ? do_int_table : 8'hFF);

assign do_dma = (dma_enable) ? ((enable_boot_rom) ? do_boot_rom :
        (enable_cartridge) ? do_cartridge :
        (enable_internal_ram) ? do_internal_ram :
        8'hFF) : 8'hFF;

// video memory address
assign A_video = (dma_enable) ? 16'hFE00 + dma_counter : A;
assign di_video = (dma_enable) ? do_dma : di;
```

```
        assign current_dma_data = do_dma;
        assign rd_n_video = (dma_enable) ? dma_rd_n_video : rd_n;
        assign wr_n_video = (dma_enable) ? dma_wr_n_video : wr_n;

endmodule
```

interrupt_module.v

```
`default_nettype none
`timescale 1ns / 1ps

module interrupt_module(
        input wire reset,
        input wire clock,
        input wire m1_n,
        input wire iorq_n,
        input wire mem_enable,
        input wire rd_n,
        input wire wr_n,
        input wire[15:0] A,
        input wire[7:0] di,
        input wire[4:0] int_req,
        output wire int_n,
        output wire[7:0] jump_addr,
        output wire[7:0] do,
        output reg[4:0] int_ack);

        //////////////////////////////////////
        // Interrupt Registers
        //
        // IF - Interrupt Flag (FF0F)
        //   Bit 4: New Value on Selected Joypad Keyline(s) (rst 60)
        //   Bit 3: Serial I/O transfer end                 (rst 58)
        //   Bit 2: Timer Overflow                          (rst 50)
        //   Bit 1: LCD (see STAT)                          (rst 48)
        //   Bit 0: V-Blank                                 (rst 40)
        //
        // IE - Interrupt Enable (FFFF)
        //   Bit 4: New Value on Selected Joypad Keyline(s)
        //   Bit 3: Serial I/O transfer end
        //   Bit 2: Timer Overflow
        //   Bit 1: LCDC (see STAT)
        //   Bit 0: V-Blank
        //
        //   0 <= disable
        //   1 <= enable
        //
        //////////////////////////////////////

        reg[7:0] IF;
        reg[7:0] IE;

        parameter POLL_STATE = 0;
        parameter WAIT_STATE = 1;
        parameter ACK_STATE = 2;
        parameter CLEAR_STATE = 3;

        parameter VBLANK_INT = 0;
        parameter LCDC_INT = 1;
        parameter TIMER_INT = 2;
        parameter SERIAL_INT = 3;
        parameter INPUT_INT = 4;

        parameter VBLANK_JUMP = 8'hA0; // 8'h40;
        parameter LCDC_JUMP   = 8'hA8; // 8'h48;
        parameter TIMER_JUMP  = 8'hB0; // 8'h50;
        parameter SERIAL_JUMP = 8'hB8; // 8'h58;
        parameter INPUT_JUMP  = 8'hC0; // 8'h60;

        reg[1:0] state;
```

```
reg[2:0] interrupt;
reg[7:0] reg_out;

always @(posedge clock)
begin

	if (reset)
	begin
		//IF <= 0;
		IE <= 0;
		state <= POLL_STATE;
	end
	else
	begin

		// Read / Write for registers
		if (mem_enable)
		begin
			if (!wr_n)
			begin
				case (A)
					//16'hFF0F: IF <= di;
					16'hFFFF: IE <= di;
				endcase
			end
			else if (!rd_n)
			begin
				case (A)
					16'hFF0F: reg_out <= IF;
					16'hFFFF: reg_out <= IE;
				endcase
			end
		end

		case (state)
			POLL_STATE:
			begin
				if (IF[VBLANK_INT] && IE[VBLANK_INT])
				begin
					interrupt <= VBLANK_INT;
					state <= WAIT_STATE;
				end
				else if (IF[LCDC_INT] && IE[LCDC_INT])
				begin
					interrupt <= LCDC_INT;
					state <= WAIT_STATE;
				end
				else if (IF[TIMER_INT] && IE[TIMER_INT])
				begin
					interrupt <= TIMER_INT;
					state <= WAIT_STATE;
				end
				else if (IF[SERIAL_INT] && IE[SERIAL_INT])
				begin
					interrupt <= SERIAL_INT;
					state <= WAIT_STATE;
				end
				else if (IF[INPUT_INT] && IE[INPUT_INT])
				begin
					interrupt <= INPUT_INT;
					state <= WAIT_STATE;
				end
			end
			WAIT_STATE:
			begin
				if (!m1_n && !iorq_n)
					state <= ACK_STATE;
			end
			ACK_STATE:
			begin
				int_ack[interrupt] <= 1;
```

```
                                state <= CLEAR_STATE;
                        end
                        CLEAR_STATE:
                        begin
                                int_ack[interrupt] <= 0;
                                if (m1_n || iorq_n)
                                        state <= POLL_STATE;
                        end
                endcase

        end

        assign IF = int_req; // this makes the value read only

end


assign do = (mem_enable) ? reg_out : 8'hFF;
assign int_n = (state == WAIT_STATE) ? 0 : 1; // active low
assign jump_addr =
                (interrupt == VBLANK_INT) ? VBLANK_JUMP :
                (interrupt == LCDC_INT) ? LCDC_JUMP :
                (interrupt == TIMER_INT) ? TIMER_JUMP :
                (interrupt == SERIAL_INT) ? SERIAL_JUMP :
                (interrupt == INPUT_INT) ? INPUT_JUMP : 8'hZZ;

endmodule
```

timer_module.v

```
`default_nettype none
`timescale 1ns / 1ps

module timer_module(
        input wire reset,
        input wire clock,
        input wire mem_enable,
        input wire rd_n,
        input wire wr_n,
        input wire int_ack,
        input wire[15:0] A,
        input wire[7:0] di,
        output reg int_req,
        output reg[7:0] do);

   ////////////////////////////////////////////////
        // Timer Registers
        //
        // DIV - Divider Register (FF04)
        //   Increments 16384 times a second
        //
        // TIMA - Timer Counter (FF05)
        //   Increments at frequency specified by TAC
        //
        // TMA - Timer Modulo (FF06)
        //   Value to load into TIMA on overflow
        //
        // TAC - Timer Control (FF07)
        //   Bit 2: 0 <= stop, 1 <= start
        //   Bit 1-0: 00 <= 4.096 KHz
        //            01 <= 262.144 KHz
        //            10 <= 65.536 KHz
        //            11 <= 16.384 KHz
   ////////////////////////////////////////////////

        reg[7:0] DIV;
        reg[7:0] TIMA;
        reg[7:0] TMA;
        reg[7:0] TAC;
```

```
reg[7:0] reg_out;

parameter MAX_TIMER = 8'hFF;

reg enable;
wire e0, e1, e2, e3;

divider #(1024) d0(reset, clock, e0);
divider #(16) d1(reset, clock, e1);
divider #(64) d2(reset, clock, e2);
divider #(256) d3(reset, clock, e3);

always @(posedge clock)
begin
	if (reset)
	begin
		DIV <= 8'h0;
		TIMA <= 8'h0;
		TMA <= 8'h0;
		TAC <= 8'h0;
		int_req <= 0;
		reg_out <= 0;
	end
	else
	begin

		// Read / Write for registers
		if (mem_enable)
		begin
			if (!wr_n)
			begin
				case (A)
					16'hFF04: DIV <= 0;
					16'hFF05: TIMA <= di;
					16'hFF06: TMA <= di;
					16'hFF07: TAC <= di;
				endcase
			end
			else if (!rd_n)
			begin
				case (A)
					16'hFF04: reg_out <= DIV;
					16'hFF05: reg_out <= TIMA;
					16'hFF06: reg_out <= TMA;
					16'hFF07: reg_out <= TAC;
				endcase
			end
		end

		// Clear overflow interrupt
		if (int_ack)
			int_req <= 0;

		// Increment timers
		if (enable)
		begin
			if (TIMA == MAX_TIMER)
				int_req <= 1;
			TIMA <= (TIMA == MAX_TIMER) ? TMA : TIMA + 1;
		end

		if (e3)
		begin
			DIV <= DIV + 1;
		end

	end

	assign do = (mem_enable) ? reg_out : 8'hZZ;
	assign enable =
		(TAC[2] == 0) ? 0 :
```

```
                        (TAC[1:0] == 0) ? e0 :
                        (TAC[1:0] == 1) ? e1 :
                        (TAC[1:0] == 2) ? e2 :
                        (TAC[1:0] == 3) ? e3 : 0;

        end

endmodule
```

video_module.v

```
`default_nettype none
`timescale 1ns / 1ps

module video_module(
        input wire reset,
        input wire clock,
        input wire mem_enable,
        input wire rd_n,
        input wire wr_n,
        input wire int_vblank_ack,
        input wire int_lcdc_ack,
        input wire[15:0] A,
        input wire[7:0] di,
        output reg int_vblank_req,
        output reg int_lcdc_req,
        output wire hsync,
        output wire vsync,
        output reg[7:0] line_count,
        output reg[8:0] pixel_count,
        output reg[7:0] pixel_data_count,
        output reg[1:0] pixel_data,
        output reg pixel_we,
        output reg[1:0] mode,
        output reg[4:0] state,
        output wire[7:0] do,
        output wire[31:0] debug_out,
        output reg[7:0] sprite_y_pos,
        output reg[7:0] sprite_x_pos,
        output reg[6:0] sprite_num,
        output reg[7:0] sprite_data1,
        output reg[7:0] sprite_data2,
        output reg[1:0] sprite_pixel,
        output reg[1:0] bg_pixel,
        output reg[7:0] oam_addrA);

        ////////////////////////////////
        //
        // Video Registers
        //
        // LCDC - LCD Control (FF40) R/W
        //
        //   Bit 7 - LCD Control Operation
    //      0: Stop completeLY (no picture on screen)
    //      1: operation
    //    Bit 6 - Window Screen Display Data Select
    //      0: $9800-$9BFF
    //      1: $9C00-$9FFF
    //    Bit 5 - Window Display
    //      0: off
    //      1: on
    //    Bit 4 - BG Character Data Select
    //      0: $8800-$97FF
    //      1: $8000-$8FFF <- Same area as OBJ
    //    Bit 3 - BG Screen Display Data Select
    //      0: $9800-$9BFF
    //      1: $9C00-$9FFF
    //    Bit 2 - OBJ Construction
    //      0: 8*8
    //      1: 8*16
```

```
//   Bit 1 - Window priority bit
//     0: window overlaps all sprites
//     1: window onLY overlaps sprites whose
//        priority bit is set to 1
//   Bit 0 - BG Display
//     0: off
//     1: on
     //
     // STAT - LCDC Status (FF41) R/W
     //
     //    Bits 6-3 - Interrupt Selection By LCDC Status
     //    Bit 6 - LYC=LY Coincidence          (1=Select)
//    Bit 5 - Mode 2: OAM-Search          (1=Enabled)
//    Bit 4 - Mode 1: V-Blank             (1=Enabled)
//    Bit 3 - Mode 0: H-Blank             (1=Enabled)
//    Bit 2 - Coincidence Flag
//      0: LYC not equal to LCDC LY
//      1: LYC = LCDC LY
//    Bit 1-0 - Mode Flag (Current STATus of the LCD controller)
//      0: During H-Blank. Entire Display Ram can be accessed.
//      1: During V-Blank. Entire Display Ram can be accessed.
//      2: During Searching OAM-RAM. OAM cannot be accessed.
//      3: During Transfering Data to LCD Driver. CPU cannot
//         access OAM and display RAM during this period.
     //
     //    The following are typical when the display is enabled:
     //
     //         Mode 0  000___000___000___000___000___000___000_________________  H-Blank
     //         Mode 1  _____________________________________________11111111111__  V-Blank
     //         Mode 2  ___2_____2_____2_____2_____2_____2_____2_____________2_  OAM
     //         Mode 3  ____33____33____33____33____33____33____33_____________3  Transfer
     //
     //
//    The Mode Flag goes through the values 00, 02,
//    and 03 at a cycle of about 109uS. 00 is present
//    about 49uS, 02 about 20uS, and 03 about 40uS. This
//    is interrupted every 16.6ms by the VBlank (01).
//    The mode flag stays set at 01 for 1.1 ms.
     //
     //    Mode 0 is present between 201-207 clks, 2 about 77-83 clks,
     //    and 3 about 169-175 clks. A complete cycle through these
     //    states takes 456 clks. VBlank lasts 4560 clks. A complete
     //    screen refresh occurs every 70224 clks.)
     //
     // SCY - Scroll Y (FF42) R/W
     //    Vertical scroll of background
     //
     // SCX - Scroll X (FF43) R/W
     //    Horizontal scroll of background
     //
     // LY - LCDC Y-Coordinate (FF44) R
     //    The LY indicates the vertical line to which
//    the present data is transferred to the LCD
//    Driver. The LY can take on any value between
//    0 through 153. The values between 144 and 153
//    indicate the V-Blank period. Writing will
//    reset the counter.
     //
//    This is just a RASTER register. The current
//    line is thrown into here. But since there are
//    no RASTERS on an LCD display it's called the
//    LCDC Y-Coordinate.
     //
     // LYC - LY Compare (FF45) R/W
     //    The LYC compares itself with the LY. If the
//    values are the same it causes the STAT to set
//    the coincident flag.
     //
     // DMA (FF46)
     //    Implemented in the MMU
     //
```

```
    // BGP - BG Palette Data (FF47) W
    //
    //    Bit 7-6 - Data for Dot Data 11
//    Bit 5-4 - Data for Dot Data 10
//    Bit 3-2 - Data for Dot Data 01
//    Bit 1-0 - Data for Dot Data 00
    //
//    This selects the shade of gray you what for
//    your BG pixel. Since each pixel uses 2 bits,
//    the corresponding shade will be selected
//    from here. The Background Color (00) lies at
//    Bits 1-0, just put a value from 0-$3 to
//    change the color.
    //
    // OBP0 - Object Palette 0 Data (FF48) W
    //    This selects the colors for sprite palette 0.
//    It works exactLY as BGP ($FF47).
//    See BGP for details.
//
    // OBP1 - Object Palette 1 Data (FF49) W
    //    This selects the colors for sprite palette 1.
//    It works exactLY as BGP ($FF47).
//    See BGP for details.
    //
    // WY - Window Y Position (FF4A) R/W
    //    0 <= WY <= 143
    //
    // WX - Window X Position + 7 (FF4B) R/W
    //    7 <= WX <= 166
//
    ////////////////////////////////////

    reg[7:0] LCDC;
    wire[7:0] STAT;
    reg[7:0] SCY;
    reg[7:0] SCX;
    reg[7:0] LYC;
    reg[7:0] BGP;
    reg[7:0] OBP0;
    reg[7:0] OBP1;
    reg[7:0] WY;
    reg[7:0] WX;

    // temp registers for r/rw mixtures
    reg[4:0] STAT_w;

    wire vram_enable, oam_enable, reg_enable;
    reg[12:0] vram_addrA;
    reg[12:0] vram_addrB;
    wire[7:0] vram_outA;
    wire[7:0] vram_outB;

    //reg[7:0] oam_addrA;
    reg[7:0] oam_addrB;
    wire[7:0] oam_outA;
    wire[7:0] oam_outB;

    reg[7:0] reg_out;

    wire vram_we_n;
    wire oam_we_n;

    reg[4:0] scanline1_addrA;
    reg[4:0] scanline1_addrB;
    reg[4:0] scanline2_addrA;
    reg[4:0] scanline2_addrB;
    reg[7:0] scanline1_inA;
    reg[7:0] scanline1_inB;
    reg[7:0] scanline2_inA;
    reg[7:0] scanline2_inB;
    wire[7:0] scanline1_outA;
```

```
wire[7:0] scanline1_outB;
wire[7:0] scanline2_outA;
wire[7:0] scanline2_outB;

reg scanlineA_we;
reg scanlineB_we;

wire clock_enable;
divider #(8) clock_divider(reset, clock, clock_enable);

sprite_ram oam(
        oam_addrA,
        oam_addrB,
        clock,
        clock,
        di,
        oam_outA,
        oam_outB,
        oam_we_n
);

//vram_rom vram(vram_addr, vram_addrB, clock, clock, vram_outA, vram_outB);

video_ram vram(
        vram_addrA,
        vram_addrB,
        clock,
        clock,
        di,
        vram_outA,
        vram_outB,
        vram_we_n
);

scanline_ram scanline1 (
        scanline1_addrA,
        scanline1_addrB,
        clock,
        clock,
        scanline1_inA,
        scanline1_inB,
        scanline1_outA,
        scanline1_outB,
        scanlineA_we,
        scanlineB_we
);

scanline_ram scanline2 (
        scanline2_addrA,
        scanline2_addrB,
        clock,
        clock,
        scanline2_inA,
        scanline2_inB,
        scanline2_outA,
        scanline2_outB,
        scanlineA_we,
        scanlineB_we
);

//reg[7:0] oam[159:0]; // oam is reg array to speed up memory accesses
//reg[7:0] scanline1[19:0];
//reg[7:0] scanline2[19:0];

// timing params -- see STAT register
parameter PIXELS = 456;
parameter LINES = 154;
parameter HACTIVE_VIDEO = 160;
parameter HBLANK_PERIOD = 41;
parameter OAM_ACTIVE = 80;
parameter RAM_ACTIVE = 172;
```

```
parameter VACTIVE_VIDEO = 144;
parameter VBLANK_PERIOD = 10;

//reg[1:0] mode;
parameter HBLANK_MODE = 0;
parameter VBLANK_MODE = 1;
parameter OAM_LOCK_MODE = 2;
parameter RAM_LOCK_MODE = 3;

//reg[3:0] state;
parameter IDLE_STATE = 0;

parameter BG_ADDR_STATE = 1;
parameter BG_ADDR_WAIT_STATE = 2;
parameter BG_DATA_STATE = 3;
parameter BG_DATA_WAIT_STATE = 4;
parameter BG_PIXEL_COMPUTE_STATE = 8;
parameter BG_PIXEL_READ_STATE = 9;
parameter BG_PIXEL_WAIT_STATE = 10;
parameter BG_PIXEL_WRITE_STATE = 11;
parameter BG_PIXEL_HOLD_STATE = 12;

parameter SPRITE_POS_STATE = 13;
parameter SPRITE_POS_WAIT_STATE = 14;
parameter SPRITE_ATTR_STATE = 15;
parameter SPRITE_ATTR_WAIT_STATE = 16;
parameter SPRITE_DATA_STATE = 17;
parameter SPRITE_DATA_WAIT_STATE = 18;
parameter SPRITE_PIXEL_COMPUTE_STATE = 19;
parameter SPRITE_PIXEL_READ_STATE = 20;
parameter SPRITE_PIXEL_WAIT_STATE = 21;
parameter SPRITE_PIXEL_DRAW_STATE = 22;
parameter SPRITE_PIXEL_DATA_STATE = 23;
parameter SPRITE_WRITE_STATE = 24;
parameter SPRITE_HOLD_STATE = 25;

parameter PIXEL_WAIT_STATE = 26;
parameter PIXEL_READ_STATE = 27;
parameter PIXEL_READ_WAIT_STATE = 28;
parameter PIXEL_OUT_STATE = 29;
parameter PIXEL_OUT_HOLD_STATE = 30;
parameter PIXEL_INCREMENT_STATE = 31;

wire[7:0] next_line_count;
wire[8:0] next_pixel_count;

reg[7:0] tile_x_pos;
reg[7:0] tile_y_pos;
reg[4:0] tile_byte_pos1;
reg[4:0] tile_byte_pos2;
reg[3:0] tile_byte_offset1;
reg[3:0] tile_byte_offset2;
reg[7:0] tile_data1;
reg[7:0] tile_data2;
reg render_background;

//reg[7:0] sprite_x_pos;
//reg[7:0] sprite_y_pos;
//reg[7:0] sprite_data1;
//reg[7:0] sprite_data2;
reg[7:0] sprite_location;
reg[7:0] sprite_attributes;
//reg[1:0] sprite_pixel;
//reg[1:0] bg_pixel;
reg[2:0] sprite_pixel_num;
reg[7:0] sprite_palette;
reg[4:0] sprite_y_size;

reg[4:0] tile_col_num; // increments from 0 -> 31
//reg[6:0] sprite_num; // increments from 0 -> 39
```

```
always @(posedge clock)
begin
	if (reset)
	begin
		// initialize registers
		LCDC <= 8'h00;  //91
		SCY  <= 8'h00;  //4f
		SCX  <= 8'h00;
		LYC  <= 8'h00;
		BGP  <= 8'hFC;  //fc
		OBP0 <= 8'h00;
		OBP1 <= 8'h00;
		WY   <= 8'h00;
		WX   <= 8'h00;

		// reset internal registers
		int_vblank_req <= 0;
		int_lcdc_req <= 0;
		mode <= 0;
		state <= 0;
		STAT_w <= 0;

		pixel_count <= 0;
		line_count <= 0;

		vram_addrA <= 0;
		vram_addrB <= 0;
	end
	else
	begin

		// memory r/w
		if (mem_enable)
		begin
			if (!rd_n)
			begin
				case (A)
					16'hFF40: reg_out <= LCDC;
					16'hFF41: reg_out <= STAT;
					16'hFF42: reg_out <= SCY;
					16'hFF43: reg_out <= SCX;
					16'hFF44: reg_out <= line_count;
					16'hFF45: reg_out <= LYC;
					16'hFF47: reg_out <= BGP;
					16'hFF48: reg_out <= OBP0;
					16'hFF49: reg_out <= OBP1;
					16'hFF4A: reg_out <= WX;
					16'hFF4B: reg_out <= WY;
				endcase
			end
			else if (!wr_n)
			begin
				case (A)
					16'hFF40: LCDC <= di;
					16'hFF41: STAT_w[4:0] <= di[7:3];
					16'hFF42: SCY <= di;
					16'hFF43: SCX <= di;
					//16'hFF44: line_count <= 0; // TODO: reset counter
					16'hFF45: LYC <= di;
					16'hFF47: BGP <= di;
					16'hFF48: OBP0 <= di;
					16'hFF49: OBP1 <= di;
					16'hFF4A: WX <= di;
					16'hFF4B: WY <= di;
				endcase
			end
		end

		// clear interrupts
		if (int_vblank_ack)
			int_vblank_req <= 0;
```

```
if (int_lcdc_ack)
	int_lcdc_req <= 0;

if (LCDC[7]) // grapics enabled
begin

	//////////////////////////////
	// STAT INTERRUPTS AND MODE //
	//////////////////////////////

	// vblank -- mode 1
	if (line_count >= VACTIVE_VIDEO)
	begin
		if (mode != VBLANK_MODE)
		begin
			int_vblank_req <= 1;
			if (STAT[4])
				int_lcdc_req <= 1;
		end
		mode <= VBLANK_MODE;
	end
	// oam lock -- mode 2
	else if (pixel_count < OAM_ACTIVE)
	begin
		if (STAT[5] && mode != OAM_LOCK_MODE)
			int_lcdc_req <= 1;
		mode <= OAM_LOCK_MODE;
	end
	// ram + oam lock -- mode 3
	else if (pixel_count < OAM_ACTIVE + RAM_ACTIVE)
	begin
		mode <= RAM_LOCK_MODE;
		// does not generate an interrupt
	end
	// hblank -- mode 0
	else
	begin
		if (STAT[3] && mode != HBLANK_MODE)
			int_lcdc_req <= 1;
		mode <= HBLANK_MODE;
	end

	// lyc interrupt
	if (pixel_count == 0 && line_count == LYC)
	begin
		// stat bit set automatically
		if (STAT[6])
			int_lcdc_req <= 1;
	end

	//////////////////////
	// RENDER GRAPHICS //
	//////////////////////

	case (state)
		IDLE_STATE:
		begin
			if (mode == RAM_LOCK_MODE)
			begin
				tile_col_num <= 0;
				sprite_num <= 0;
				pixel_data_count <= 0;
				state <= BG_ADDR_STATE;
			end
		end

		////////////////
		// BACKGROUND //
		////////////////

		BG_ADDR_STATE:
```

```
                                        begin
                                                // disable writes
                                                scanlineA_we <= 0;
                                                scanlineB_we <= 0;

                                                if (LCDC[5] && WY <= line_count) // enable window
                                                begin
                                                        tile_x_pos <= {tile_col_num, 3'b0} + (WX -
7);
                                                        tile_y_pos <= (line_count - WY);
                                                        vram_addrA <=
                                                                {(line_count - WY) >> 3, 5'b0} + //
(tile_y_pos[7:3] << 5)
                                                                (({tile_col_num, 3'b0} + (WX - 7)) >>
3) + // (tile_x_pos[7:3])
                                                                ((LCDC[6]) ? 16'h1C00 : 16'h1800);
                                                        render_background <= 1;
                                                        state <= BG_ADDR_WAIT_STATE;
                                                end
                                                else if (LCDC[0]) // enable background
                                                begin
                                                        tile_x_pos <= {tile_col_num, 3'b0} + (SCX);
                                                        tile_y_pos <= (SCY + line_count);
                                                        vram_addrA <=
                                                                {(SCY + line_count) >> 3, 5'b0} +
                                                                (({tile_col_num, 3'b0} + (SCX)) >> 3)
+
                                                                ((LCDC[3]) ? 16'h1C00 : 16'h1800);
                                                        render_background <= 1;
                                                        state <= BG_ADDR_WAIT_STATE;
                                                end
                                                else
                                                begin
                                                        tile_x_pos <= {tile_col_num, 3'b0};
                                                        tile_y_pos <= line_count;
                                                        render_background <= 0;
                                                        state <= BG_PIXEL_COMPUTE_STATE;
                                                end
                                        end

                                        BG_ADDR_WAIT_STATE:
                                        begin
                                                state <= BG_DATA_STATE;
                                        end

                                        BG_DATA_STATE:
                                        begin
                                                //tile_id_num <= vram_outA;
                                                vram_addrA <=
                                                        (LCDC[4]) ?
                                                                16'h0000 + { vram_outA, 4'b0 } + {
tile_y_pos[2:0], 1'b0 } :
                                                                (( { vram_outA, 4'b0 } + {
tile_y_pos[2:0], 1'b0 }) < 128) ?
                                                                        16'h1000 + { vram_outA, 4'b0
} + { tile_y_pos[2:0], 1'b0 } :
                                                                        16'h1000 - (~({ vram_outA,
4'b0 } + { tile_y_pos[2:0], 1'b0 }) + 1);
                                                vram_addrB <=
                                                        (LCDC[4]) ?
                                                                16'h0000 + { vram_outA, 4'b0 } + {
tile_y_pos[2:0], 1'b0 } + 1 :
                                                                (( { vram_outA, 4'b0 } + {
tile_y_pos[2:0], 1'b0 } + 1 ) < 128) ?
                                                                        16'h1000 + { vram_outA, 4'b0
} + { tile_y_pos[2:0], 1'b0 } + 1 :
                                                                        16'h1000 - (~({ vram_outA,
4'b0 } + { tile_y_pos[2:0], 1'b0 } + 1) + 1);
                                                state <= BG_DATA_WAIT_STATE;
                                        end
```

```
BG_DATA_WAIT_STATE:
begin
	state <= BG_PIXEL_COMPUTE_STATE;
end

BG_PIXEL_COMPUTE_STATE:
begin
	tile_data1 <= vram_outA;
	tile_data2 <= vram_outB;
	tile_byte_pos1 <= tile_x_pos >> 3;
	tile_byte_pos2 <= ((tile_x_pos + 8) & 8'hFF) >> 3;
	tile_byte_offset1 <= tile_x_pos[2:0];
	tile_byte_offset2 <= 8 - tile_x_pos[2:0];
	state <= BG_PIXEL_READ_STATE;
end

BG_PIXEL_READ_STATE:
begin
	scanline1_addrA <= tile_byte_pos1;
	scanline1_addrB <= tile_byte_pos2;
	scanline2_addrA <= tile_byte_pos1;
	scanline2_addrB <= tile_byte_pos2;
	state <= BG_PIXEL_WAIT_STATE;
end

BG_PIXEL_WAIT_STATE:
begin
	state <= BG_PIXEL_WRITE_STATE;
end

BG_PIXEL_WRITE_STATE:
begin
	// first byte
	scanline1_inA <=
		(render_background) ?
			(scanline1_outA &
			(8'hFF << tile_byte_offset2) |
			(tile_data1 >> tile_byte_offset1)) :
		0;
	scanline2_inA <=
		(render_background) ?
			(scanline2_outA &
			(8'hFF << tile_byte_offset2) |
			(tile_data2 >> tile_byte_offset1)) :
		0;

	// second byte
	scanline1_inB <=
		(render_background) ?
			(scanline1_outB &
			~(8'hFF << tile_byte_offset2) |
			(tile_data1 << tile_byte_offset2)) :
		0;
	scanline2_inB <=
		(render_background) ?
			(scanline2_outB &
			~(8'hFF << tile_byte_offset2) |
			(tile_data2 << tile_byte_offset2)) :
		0;

	// enable writes
	scanlineA_we <= (tile_byte_pos1 < 20) ? 1 : 0;
	scanlineB_we <= (tile_byte_pos2 < 20) ? 1 : 0;
	state <= BG_PIXEL_HOLD_STATE;
end

BG_PIXEL_HOLD_STATE:
begin
	// increment col
	if (tile_col_num == 31)
		state <= SPRITE_POS_STATE;
```

```
                else
                begin
                        tile_col_num <= tile_col_num + 1;
                        state <= BG_ADDR_STATE;
                end
        end

        //////////////
        // SPRITES //
        //////////////

        SPRITE_POS_STATE:
        begin
                // disable writes
                scanlineA_we <= 0;
                scanlineB_we <= 0;
                sprite_y_size <= LCDC[2] ? 16 : 8;

                oam_addrA <= { sprite_num, 2'b00 }; // y pos
                oam_addrB <= { sprite_num, 2'b01 }; // x pos
                state <= SPRITE_POS_WAIT_STATE;
        end

        SPRITE_POS_WAIT_STATE:
        begin
                state <= SPRITE_ATTR_STATE;
        end

        SPRITE_ATTR_STATE:
        begin
                sprite_y_pos <= oam_outA - 16;
                sprite_x_pos <= oam_outB - 8;
                if (line_count >= (oam_outA - 16) && line_count <
(oam_outA - 16) + sprite_y_size)
                begin
                        oam_addrA <= { sprite_num, 2'b10 }; // tile
addr
                        oam_addrB <= { sprite_num, 2'b11 }; //
attributes
                        state <= SPRITE_ATTR_WAIT_STATE;
                end
                else
                begin
                        state <= SPRITE_HOLD_STATE;
                end
        end

        SPRITE_ATTR_WAIT_STATE:
        begin
                state <= SPRITE_DATA_STATE;
        end

        SPRITE_DATA_STATE:
        begin
                sprite_attributes <= oam_outB;
                vram_addrA <=
                        { (oam_outB[6]) ?
                                ((line_count - sprite_y_pos) -
sprite_y_size) * -1 :
                                (line_count - sprite_y_pos), 1'b0 } +
                        { oam_outA, 4'b0 };
                vram_addrB <=
                        { (oam_outB[6]) ?
                                ((line_count - sprite_y_pos) -
sprite_y_size) * -1 :
                                (line_count - sprite_y_pos), 1'b0 } +
                        { oam_outA, 4'b0 } + 1;
                state <= SPRITE_DATA_WAIT_STATE;
        end

        SPRITE_DATA_WAIT_STATE:
```

```
                                        begin
                                                state <= SPRITE_PIXEL_COMPUTE_STATE;
                                        end

                                        SPRITE_PIXEL_COMPUTE_STATE:
                                        begin
                                                tile_data1 <= vram_outA;
                                                tile_data2 <= vram_outB;
                                                tile_byte_pos1 <= sprite_x_pos >> 3;
                                                tile_byte_pos2 <= (sprite_x_pos >> 3) + 1;
                                                tile_byte_offset1 <= sprite_x_pos[2:0];
                                                tile_byte_offset2 <= 8 - sprite_x_pos[2:0];
                                                state <= SPRITE_PIXEL_READ_STATE;
                                        end

                                        SPRITE_PIXEL_READ_STATE:
                                        begin
                                                scanline1_addrA <= tile_byte_pos1;
                                                scanline1_addrB <= tile_byte_pos2;
                                                scanline2_addrA <= tile_byte_pos1;
                                                scanline2_addrB <= tile_byte_pos2;
                                                state <= SPRITE_PIXEL_WAIT_STATE;
                                        end

                                        SPRITE_PIXEL_WAIT_STATE:
                                        begin
                                                sprite_pixel_num <= 0;
                                                sprite_palette <= (sprite_attributes[4]) ? OBP1 :
OBP0;
                                                state <= SPRITE_PIXEL_DRAW_STATE;
                                        end

                                        SPRITE_PIXEL_DRAW_STATE:
                                        begin
                                                sprite_pixel <=
                                                        (sprite_palette >>
                                                                { tile_data2[sprite_pixel_num],
tile_data1[sprite_pixel_num], 1'b0 } ) & 2'b11;
                                                bg_pixel <=
                                                        (BGP >> (sprite_pixel_num <
tile_byte_offset2) ?
                                                                { scanline2_outA[sprite_pixel_num +
tile_byte_offset1],

        scanline1_outA[sprite_pixel_num + tile_byte_offset1], 1'b0 } :
                                                                { scanline2_outB[sprite_pixel_num -
tile_byte_offset1],

        scanline1_outB[sprite_pixel_num - tile_byte_offset1], 1'b0 } ) & 2'b11;
                                                state <= SPRITE_PIXEL_DATA_STATE;
                                        end

                                        SPRITE_PIXEL_DATA_STATE:
                                        begin
                                                if (sprite_pixel == 2'b00 || (sprite_attributes[7]
&& bg_pixel != 2'b00))
                                                begin
                                                        sprite_data1 <=
                                                                (sprite_data1 & ~(8'h01 <<
sprite_pixel_num)) |
                                                                (bg_pixel[0] << sprite_pixel_num);
                                                        sprite_data2 <=
                                                                (sprite_data2 & ~(8'h01 <<
sprite_pixel_num)) |
                                                                (bg_pixel[1] << sprite_pixel_num);
                                                end
                                                else
                                                begin
                                                        sprite_data1 <=
                                                                (sprite_data1 & ~(8'h01 <<
sprite_pixel_num)) |
```

```
                                                        (sprite_pixel[0] <<
sprite_pixel_num);
                                                sprite_data2 <=
                                                        (sprite_data2 & ~(8'h01 <<
sprite_pixel_num)) |
                                                        (sprite_pixel[1] <<
sprite_pixel_num);
                                        end

                                        if (sprite_pixel_num < 7)
                                        begin
                                                sprite_pixel_num <= sprite_pixel_num + 1;
                                                state <= SPRITE_PIXEL_DRAW_STATE;
                                        end
                                        else
                                        begin
                                                state <= SPRITE_WRITE_STATE;
                                        end
                                end

                                SPRITE_WRITE_STATE:
                                begin
                                        // first byte
                                        scanline1_inA <=
                                                (scanline1_outA &
                                                (8'hFF << tile_byte_offset2) |
                                                (sprite_data1 >> tile_byte_offset1));
                                        scanline2_inA <=
                                                (scanline2_outA &
                                                (8'hFF << tile_byte_offset2) |
                                                (sprite_data2 >> tile_byte_offset1));

                                        // second byte
                                        scanline1_inB <=
                                                (scanline1_outB &
                                                ~(8'hFF << tile_byte_offset2) |
                                                (sprite_data1 << tile_byte_offset2));
                                        scanline2_inB <=
                                                (scanline2_outB &
                                                ~(8'hFF << tile_byte_offset2) |
                                                (sprite_data2 << tile_byte_offset2));

                                        // enable writes
                                        scanlineA_we <= (tile_byte_pos1 < 20) ? 1 : 0;
                                        scanlineB_we <= (tile_byte_pos2 < 20 &&
tile_byte_pos2 > tile_byte_pos1) ? 1 : 0; // don't wrap
                                        state <= SPRITE_HOLD_STATE;
                                end

                                SPRITE_HOLD_STATE:
                                begin
                                        if (sprite_num == 39)
                                                state <= PIXEL_WAIT_STATE;
                                        else
                                        begin
                                                sprite_num <= sprite_num + 1;
                                                state <= SPRITE_POS_STATE;
                                        end
                                end

                                //////////////
                                // DISPLAY //
                                //////////////

                                PIXEL_WAIT_STATE:
                                begin
                                        // disable writes
                                        scanlineA_we <= 0;
                                        scanlineB_we <= 0;

                                        if (mode == HBLANK_MODE)
```

```
                                        state <= PIXEL_READ_STATE;
                                end

                                PIXEL_READ_STATE:
                                begin
                                        scanline1_addrA <= pixel_data_count >> 3;
                                        scanline2_addrA <= pixel_data_count >> 3;
                                        state <= PIXEL_READ_WAIT_STATE;
                                end

                                PIXEL_READ_WAIT_STATE:
                                begin
                                        state <= PIXEL_OUT_STATE;
                                end

                                PIXEL_OUT_STATE:
                                begin
                                        pixel_data <=
                                                (BGP >>
                                                        { scanline2_outA[7 -
pixel_data_count[2:0]],
                                                                scanline1_outA[7 -
pixel_data_count[2:0]], 1'b0} ) & 2'b11;
                                        pixel_we <= 1;
                                        state <= PIXEL_OUT_HOLD_STATE;
                                end

                                PIXEL_OUT_HOLD_STATE:
                                begin
                                        pixel_we <= 0;
                                        state <= PIXEL_INCREMENT_STATE;
                                end

                                PIXEL_INCREMENT_STATE:
                                begin
                                        if (pixel_data_count < 160)
                                        begin
                                                pixel_data_count <= pixel_data_count + 1;
                                                if (pixel_data_count[2:0] == 7)
                                                        state <= PIXEL_READ_STATE;
                                                else
                                                        state <= PIXEL_OUT_STATE;
                                        end
                                        else
                                                state <= IDLE_STATE;
                                end
                        endcase

                end
                else
                begin
                        mode <= HBLANK_MODE;
                end

                // failsafe -- if we somehow exceed the allotted cycles for rendering
                if (mode != RAM_LOCK_MODE && state < PIXEL_WAIT_STATE && state >
IDLE_STATE)
                        state <= PIXEL_WAIT_STATE;

                if (mode < RAM_LOCK_MODE)
                        vram_addrA <= A - 16'h8000;
                if (mode < OAM_LOCK_MODE)
                        oam_addrA <= A - 16'hFE00;

                if (clock_enable)
                begin
                        pixel_count <= next_pixel_count;
                        line_count <= next_line_count;
                end
        end
    end
```

```
        assign next_pixel_count = (LCDC[7]) ?
                ((pixel_count == PIXELS - 1) ? 0 : pixel_count + 1) : 0;

        assign next_line_count = (LCDC[7]) ?
                ((pixel_count == PIXELS - 1) ?
                        ((line_count == LINES - 1) ? 0 : line_count + 1) : line_count) : 0;

        assign hsync = (pixel_count > OAM_ACTIVE + RAM_ACTIVE + HACTIVE_VIDEO) ? 1 : 0;
        assign vsync = (line_count > VACTIVE_VIDEO) ? 1 : 0;

        assign vram_enable = mem_enable && (A >= 16'h8000 && A < 16'hA000);
        assign oam_enable = mem_enable && (A >= 16'hFE00 && A < 16'hFEA0);
        assign reg_enable = mem_enable && !vram_enable && !oam_enable;

        assign vram_we_n = !(vram_enable && !wr_n && mode != RAM_LOCK_MODE);
        assign oam_we_n = !(oam_enable && !wr_n && mode != RAM_LOCK_MODE && mode !=
OAM_LOCK_MODE);

        assign STAT[7:3] = STAT_w[4:0]; // r/w
        assign STAT[2] = (line_count == LYC) ? 1 : 0; // LYC Coincidence flag
        assign STAT[1:0] = mode; // read only -- set internally

        assign debug_out = (vram_addrA << 16) | (line_count << 8) | (vram_outA);

        assign do =
                (vram_enable) ? vram_outA :
                (oam_enable) ? oam_outA :
                (reg_enable) ? reg_out : 8'hFF;

endmodule
```

video_converter.v

```
`default_nettype none
`timescale 1ns / 1ps

module video_converter(
        input wire reset,
        input wire clock,
        input wire gb_clock,

        // run on gb_clock
        input wire[1:0] pixel_data,
        input wire[7:0] gb_pixel_count,
        input wire[7:0] gb_line_count,
        input wire gb_hsync,   // active high
        input wire gb_vsync,   // active high
        input wire gb_we,

        // run on clock
        output wire[23:0] color,
        output wire sync_b,
        output wire blank_b,
        output wire pixel_clock,
        output wire hsync,
        output wire vsync
        );

        // game boy screen size
        parameter GB_SCREEN_WIDTH = 10'd160;
        parameter GB_SCREEN_HEIGHT = 10'd144;

        // toggle for which is the front buffer
        // 0 -> buffer1 is front buffer
        // 1 -> buffer2 is front buffer
        reg front_buffer;

        //wire[14:0] read_addr;
```

```
wire[14:0] write_addr;
wire[1:0] read_data;
//wire[1:0] write_data;

wire[14:0] b1_addr;
wire b1_clk;
wire[1:0] b1_din;
wire[1:0] b1_dout;
wire b1_we;    // active high

wire[14:0] b2_addr;
wire b2_clk;
wire[1:0] b2_din;
wire[1:0] b2_dout;
wire b2_we;    // active high

reg[1:0] last_pixel_data;
reg[14:0] last_write_addr;

reg write_enable;

assign b1_we = front_buffer ? (gb_we) : 0;
assign b2_we = front_buffer ? 0 : (gb_we);

assign read_data = (front_buffer) ? b2_dout : b1_dout;
//assign pixel_data = (front_buffer) ? b1_din : b2_din;
assign b1_din = (front_buffer) ? pixel_data : 0;
assign b2_din = (front_buffer) ? 0 : pixel_data;

BUFGMUX clock_mux_b1(.S(front_buffer), .O(b1_clk),
                     .I0(clock), .I1(gb_clock));
BUFGMUX clock_mux_b2(.S(front_buffer), .O(b2_clk),
                     .I0(gb_clock), .I1(clock));

// internal buffer ram
frame_buffer buffer1(
        b1_addr,
        b1_clk,
        b1_din,
        b1_dout,
        b1_we
);

frame_buffer buffer2(
        b2_addr,
        b2_clk,
        b2_din,
        b2_dout,
        b2_we
);

reg gb_last_vsync;
reg gb_last_hsync;

// handle writing into the back_buffer
always @ (posedge gb_clock)
begin
        if(reset) begin
                front_buffer <= 0;
                gb_last_vsync <= 0;
        end else begin
                gb_last_vsync <= gb_vsync;
        end

        // detect positive vsync edge
        if(~gb_last_vsync && gb_vsync) begin
                front_buffer <= ~front_buffer;
        end
end

// detect vsync edge
```

```
        // handle output to the vga module
        wire my_hsync, my_vsync;
        wire [9:0] pixel_count, line_count;
        vga_controller vgac(clock, reset, my_hsync, my_vsync, pixel_count, line_count);

        // write to our current counter
        //assign write_addr = my_line_count * 160 + my_pixel_count;
        assign write_addr = gb_line_count * 160 + gb_pixel_count;

        parameter X_OFFSET = 160;
        parameter Y_OFFSET = 76;

        // read from where the vga wants to read
        wire[14:0] buffer_pos = ((line_count - Y_OFFSET) >> 1) * 160 + ((pixel_count - X_OFFSET)
>> 1);

        assign b1_addr = (front_buffer) ? write_addr : buffer_pos;
        assign b2_addr = (front_buffer) ? buffer_pos : write_addr;

        // delay the outputs by two clocks
        reg [2:0] hdelay, vdelay;

        always @ (posedge clock) begin
                if(reset) begin
                        hdelay <= 3'b111;
                        vdelay <= 3'b111;
                end else begin
                        hdelay <= {hdelay[1:0], my_hsync};
                        vdelay <= {vdelay[1:0], my_vsync};
                end
        end

        // generate a gameboy color
        // 00 -> white
        // 01 -> light gray
        // 10 -> dark gray
        // 11 -> black

        wire [7:0] my_color = (pixel_count >= X_OFFSET && line_count >= Y_OFFSET && pixel_count <
X_OFFSET + 320 && line_count < Y_OFFSET + 288) ? (read_data == 2'b00) ? 8'b11111111 :
                                                                        ((read_data == 2'b01) ? 8'b10101010
:
                                                                        ((read_data == 2'b10) ? 8'b01010101
: 8'b00000000)) : 8'b00000000;

        assign color[7:0] = my_color;
        assign color[15:8] = my_color;
        assign color[23:16] = my_color;

   assign sync_b = hdelay[2] ^ vdelay[2];
   assign blank_b = (pixel_count[9:0] < 640) && (line_count[9:0] < 480);
   assign pixel_clock = ~clock;
   assign hsync = hdelay[2];
   assign vsync = vdelay[2];
endmodule
```

input_controller.v

```
`timescale 1ns / 1ps

module input_controller(
        input wire reset,
        input wire clock,

        input wire controller_data,
        output reg controller_latch,
        output reg controller_clock,
        output reg[7:0] button_state,
```

```
	output wire[7:0] reg_state,
	input wire mem_enable,

	// active low
	input wire rd_n,
	input wire wr_n,
	input wire int_ack,
	input wire[15:0] A,
	input wire[7:0] di,
	output reg int_req,
	output wire[7:0] do
	);

	reg[3:0] read_count;
	wire read_pulse;
	reg[7:0] last_button_state;

	// inverted, 0 means button is pressed

	reg[7:0] P1;
	reg[7:0] reg_out;

	assign reg_state = reg_out;

	//56250
	divider#(.DELAY(56250)) khz(
		.reset(reset),
		.clock(clock),
		.enable(read_pulse));

	always @(posedge clock)
	begin
		if(reset) begin
			button_state <= 8'b0;
			controller_latch <= 0;
			controller_clock <= 0;
			read_count <= 0;
			reg_out <= 8'b11001111;
			P1 <= 8'b11001111;

			int_req <= 0;
		end else begin
			last_button_state <= button_state;

			if(|(last_button_state & ~(button_state))) begin
				// button pressed, do interrupt
				int_req <= 1;
			end else if(int_ack) begin
				int_req <= 0;
			end

			// see if button was pressed
			if(~wr_n) begin
				if(A == 16'hFF00) begin
					P1 <= { 2'b0, di[5:4], 4'b0 };
				end
			end else if(~rd_n) begin
				if(A == 16'hFF00) begin
					// A, B, select, start, up, down, left, right
					if(P1[5] == 0 && P1[4] == 0) begin
						reg_out <= 8'hFF;
					end else if(P1[5] == 0) begin
						// select button keys
						reg_out <= {2'b11, P1[5:4], button_state[3],
button_state[2], button_state[1], button_state[0]};
					end else if(P1[4] == 0) begin
						// select direction keys
						reg_out <= {2'b11, P1[5:4], button_state[5],
button_state[4], button_state[6], button_state[7]};
					end else begin
						reg_out <= 8'hFF;
```

```
                        end
                end
        end

        if(read_pulse) begin
                if(read_count >= 8) begin
                        read_count <= 0;
                        controller_latch <= 1;
                        controller_clock <= 0;
                end else begin
                        controller_latch <= 0;
                        controller_clock <= ~controller_clock;

                        if(~controller_clock) begin
                                read_count <= read_count + 1;
                        end
                end
        end

        if(~controller_clock) begin
                // button order is
                // A, B, select, start, up, down, left, right
                case(read_count)
                        0: button_state[0] <= controller_data;
                        1: button_state[1] <= controller_data;
                        2: button_state[2] <= controller_data;
                        3: button_state[3] <= controller_data;
                        4: button_state[4] <= controller_data;
                        5: button_state[5] <= controller_data;
                        6: button_state[6] <= controller_data;
                        7: button_state[7] <= controller_data;
                endcase
        end
    end
end

assign do = (mem_enable) ? reg_out : 8'hFF;

endmodule
```

Appendix B: TV80 Core

tv80s.v

```
//
// TV80 8-Bit Microprocessor Core
// Based on the VHDL T80 core by Daniel Wallner (jesus@opencores.org)
//
// Copyright (c) 2004 Guy Hutchison (ghutchis@opencores.org)
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included
// in all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
// EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
// MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
// IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
// CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
// TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
// SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

```
module tv80s (/*AUTOARG*/
  // Outputs
  m1_n, mreq_n, iorq_n, rd_n, wr_n, rfsh_n, halt_n, busak_n, A, do,
  BC, DE, HL, ACC, F, PC, SP, IntE_FF1, IntE_FF2, INT_s,
  // Inputs
  reset_n, clk, wait_n, int_n, nmi_n, busrq_n, di
  );

  parameter Mode = 3;    // 0 => Z80, 1 => Fast Z80, 2 => 8080, 3 => GB
  parameter T2Write = 1; // 0 => wr_n active in T3, /=0 => wr_n active in T2
  parameter IOWait  = 1; // 0 => Single cycle I/O, 1 => Std I/O cycle


  input         reset_n;
  input         clk;
  input         wait_n;
  input         int_n;
  input         nmi_n;
  input         busrq_n;
  output        m1_n;
  output        mreq_n;
  output        iorq_n;
  output        rd_n;
  output        wr_n;
  output        rfsh_n;
  output        halt_n;
  output        busak_n;
  output [15:0] A;
  input [7:0]   di;
  output [7:0]  do;
  output [15:0] BC;
  output [15:0] DE;
  output [15:0] HL;
  output [7:0]  F;
  output [7:0]  ACC;
  output [15:0] PC;
  output [15:0] SP;
  output        IntE_FF1;
  output        IntE_FF2;
  output        INT_s;

  reg           mreq_n;
  reg           iorq_n;
  reg           rd_n;
  reg           wr_n;

  wire          cen;
  wire          intcycle_n;
  wire          no_read;
  wire          write;
  wire          iorq;
  reg [7:0]     di_reg;
  wire [6:0]    mcycle;
  wire [6:0]    tstate;

  assign    cen = 1;

  tv80_core #(Mode, IOWait) i_tv80_core
    (
     .cen (cen),
     .m1_n (m1_n),
     .iorq (iorq),
     .no_read (no_read),
     .write (write),
     .rfsh_n (rfsh_n),
     .halt_n (halt_n),
     .wait_n (wait_n),
     .int_n (int_n),
     .nmi_n (nmi_n),
     .reset_n (reset_n),
     .busrq_n (busrq_n),
```

```
    .busak_n (busak_n),
    .clk (clk),
    .IntE (),
    .stop (),
    .A (A),
    .dinst (di),
    .di (di_reg),
    .do (do),
    .mc (mcycle),
    .ts (tstate),
    .intcycle_n (intcycle_n),
        .BC (BC),
        .DE (DE),
        .HL (HL),
        .F (F),
        .ACC (ACC),
        .PC (PC),
    .SP (SP),
    .IntE_FF1(IntE_FF1),
    .IntE_FF2(IntE_FF2),
    .INT_s(INT_s)
    );

  always @(posedge clk)
    begin
      if (!reset_n)
        begin
          rd_n   <= #1 1'b1;
          wr_n   <= #1 1'b1;
          iorq_n <= #1 1'b1;
          mreq_n <= #1 1'b1;
          di_reg <= #1 0;
        end
      else
        begin
          rd_n <= #1 1'b1;
          wr_n <= #1 1'b1;
          iorq_n <= #1 1'b1;
          mreq_n <= #1 1'b1;
          if (mcycle[0])
            begin
              if (tstate[1] || (tstate[2] && wait_n == 1'b0))
                begin
                  rd_n <= #1 ~ intcycle_n;
                  mreq_n <= #1 ~ intcycle_n;
                  iorq_n <= #1 intcycle_n;
                end
            `ifdef TV80_REFRESH
              if (tstate[3])
            mreq_n <= #1 1'b0;
            `endif
            end // if (mcycle[0])
          else
            begin
              if ((tstate[1] || (tstate[2] && wait_n == 1'b0)) && no_read == 1'b0 && write ==
1'b0)
                begin
                  rd_n <= #1 1'b0;
                  iorq_n <= #1 ~ iorq;
                  mreq_n <= #1 iorq;
                end
              if (T2Write == 0)
                begin
                  if (tstate[2] && write == 1'b1)
                    begin
                      wr_n <= #1 1'b0;
                      iorq_n <= #1 ~ iorq;
                      mreq_n <= #1 iorq;
                    end
                end
              else
```

```
                begin
                  if ((tstate[1] || (tstate[2] && wait_n == 1'b0)) && write == 1'b1)
                    begin
                      wr_n <= #1 1'b0;
                      iorq_n <= #1 ~ iorq;
                      mreq_n <= #1 iorq;
                    end
                end // else: !if(T2write == 0)

            end // else: !if(mcycle[0])

          if (tstate[2] && wait_n == 1'b1)
            di_reg <= #1 di;
        end // else: !if(!reset_n)
    end // always @ (posedge clk or negedge reset_n)

endmodule // t80s
```

tv80_core.v

```
//
// TV80 8-Bit Microprocessor Core
// Based on the VHDL T80 core by Daniel Wallner (jesus@opencores.org)
//
// Copyright (c) 2004 Guy Hutchison (ghutchis@opencores.org)
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included
// in all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
// EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
// MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
// IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
// CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
// TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
// SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

module tv80_core (/*AUTOARG*/
  // Outputs
  m1_n, iorq, no_read, write, rfsh_n, halt_n, busak_n, A, do, mc, ts,
  intcycle_n, IntE, stop, BC, DE, HL, F, ACC, PC, SP, IntE_FF1, IntE_FF2, INT_s,
  // Inputs
  reset_n, clk, cen, wait_n, int_n, nmi_n, busrq_n, dinst, di
  );
  // Beginning of automatic inputs (from unused autoinst inputs)
  // End of automatics

  parameter Mode = 3;   // 0 => Z80, 1 => Fast Z80, 2 => 8080, 3 => GB
  parameter IOWait = 1; // 0 => Single cycle I/O, 1 => Std I/O cycle
  /*
  parameter Flag_C = 0;
  parameter Flag_N = 1;
  parameter Flag_P = 2;
  parameter Flag_X = 3;
  parameter Flag_H = 4;
  parameter Flag_Y = 5;
  parameter Flag_Z = 6;
  parameter Flag_S = 7;
  */
  parameter Flag_C = 4;
  parameter Flag_N = 0;
  parameter Flag_P = 1;
  parameter Flag_X = 2;
```

```
parameter Flag_H = 5;
parameter Flag_Y = 6;
parameter Flag_Z = 7;
parameter Flag_S = 3;


input     reset_n;
input     clk;
input     cen;
input     wait_n;
input     int_n;
input     nmi_n;
input     busrq_n;
output    m1_n;
output    iorq;
output    no_read;
output    write;
output    rfsh_n;
output    halt_n;
output    busak_n;
output [15:0] A;
input [7:0]   dinst;
input [7:0]   di;
output [7:0]  do;
output [6:0]  mc;
output [6:0]  ts;
output        intcycle_n;
output        IntE;
output        stop;
output [15:0] BC;
output [15:0] DE;
output [15:0] HL;
output [7:0]  F;
output [7:0]  ACC;
output [15:0] PC;
output [15:0] SP;
output        IntE_FF1;
output        IntE_FF2;
output        INT_s;

reg    m1_n;
reg    iorq;
reg    rfsh_n;
reg    halt_n;
reg    busak_n;
reg [15:0] A;
reg [7:0]  do;
reg [6:0]  mc;
reg [6:0]  ts;
reg   intcycle_n;
reg   IntE;
reg   stop;

parameter     aNone    = 3'b111;
parameter     aBC      = 3'b000;
parameter     aDE      = 3'b001;
parameter     aXY      = 3'b010;
parameter     aIOA     = 3'b100;
parameter     aSP      = 3'b101;
parameter     aZI      = 3'b110;

// Registers
reg [7:0]     ACC, F;
reg [7:0]     Ap, Fp;
reg [7:0]     I;
reg [7:0]     R;
reg [15:0]    SP, PC;
reg [7:0]     RegDIH;
reg [7:0]     RegDIL;
wire [15:0]   RegBusA;
wire [15:0]   RegBusB;
```

```
wire [15:0]   RegBusC;
reg [2:0]     RegAddrA_r;
reg [2:0]     RegAddrA;
reg [2:0]     RegAddrB_r;
reg [2:0]     RegAddrB;
reg [2:0]     RegAddrC;
reg           RegWEH;
reg           RegWEL;
reg           Alternate;

// Help Registers
reg [15:0]    TmpAddr;        // Temporary address register
reg [7:0]     IR;             // Instruction register
reg [1:0]     ISet;           // Instruction set selector
reg [15:0]    RegBusA_r;

reg [15:0]    ID16;
reg [7:0]     Save_Mux;

reg [6:0]     tstate;
reg [6:0]     mcycle;
reg           last_mcycle, last_tstate;
reg           IntE_FF1;
reg           IntE_FF2;
reg           Halt_FF;
reg           BusReq_s;
reg           BusAck;
reg           ClkEn;
reg           NMI_s;
reg           INT_s;
reg [1:0]     IStatus;

reg [7:0]     DI_Reg;
reg           T_Res;
reg [1:0]     XY_State;
reg [2:0]     Pre_XY_F_M;
reg           NextIs_XY_Fetch;
reg           XY_Ind;
reg           No_BTR;
reg           BTR_r;
reg           Auto_Wait;
reg           Auto_Wait_t1;
reg           Auto_Wait_t2;
reg           IncDecZ;

// ALU signals
reg [7:0]     BusB;
reg [7:0]     BusA;
wire [7:0]    ALU_Q;
wire [7:0]    F_Out;

// Registered micro code outputs
reg [4:0]     Read_To_Reg_r;
reg           Arith16_r;
reg           Z16_r;
reg [3:0]     ALU_Op_r;
reg           Save_ALU_r;
reg           PreserveC_r;
reg [2:0]     mcycles;

// Micro code outputs
wire [2:0]    mcycles_d;
wire [2:0]    tstates;
reg           IntCycle;
reg           NMICycle;
wire          Inc_PC;
wire          Inc_WZ;
wire [3:0]    IncDec_16;
wire [1:0]    Prefix;
wire          Read_To_Acc;
wire          Read_To_Reg;
```

```
wire [3:0]     Set_BusB_To;
wire [3:0]     Set_BusA_To;
wire [3:0]     ALU_Op;
wire           Save_ALU;
wire           PreserveC;
wire           Arith16;
wire [2:0]     Set_Addr_To;
wire           Jump;
wire           JumpE;
wire           JumpXY;
wire           Call;
wire           RstP;
wire           LDZ;
wire           LDW;
wire           LDSPHL;
wire           iorq_i;
wire [2:0]     Special_LD;
wire           ExchangeDH;
wire           ExchangeRp;
wire           ExchangeAF;
wire           ExchangeRS;
wire           I_DJNZ;
wire           I_CPL;
wire           I_CCF;
wire           I_SCF;
wire           I_RETN;
wire           I_BT;
wire           I_BC;
wire           I_BTR;
wire           I_RLD;
wire           I_RRD;
wire           I_INRC;
wire           SetDI;
wire           SetEI;
wire [1:0]     IMode;
wire           Halt;

reg [15:0]     PC16;
reg [15:0]     PC16_B;
reg [15:0]     SP16, SP16_A, SP16_B;
reg [15:0]     ID16_B;
reg            Oldnmi_n;

tv80_mcode #(Mode, Flag_C, Flag_N, Flag_P, Flag_X, Flag_H, Flag_Y, Flag_Z, Flag_S) i_mcode
  (
   .IR                  (IR),
   .ISet                (ISet),
   .MCycle              (mcycle),
   .F                   (F),
   .NMICycle            (NMICycle),
   .IntCycle            (IntCycle),
   .MCycles             (mcycles_d),
   .TStates             (tstates),
   .Prefix              (Prefix),
   .Inc_PC              (Inc_PC),
   .Inc_WZ              (Inc_WZ),
   .IncDec_16           (IncDec_16),
   .Read_To_Acc         (Read_To_Acc),
   .Read_To_Reg         (Read_To_Reg),
   .Set_BusB_To         (Set_BusB_To),
   .Set_BusA_To         (Set_BusA_To),
   .ALU_Op              (ALU_Op),
   .Save_ALU            (Save_ALU),
   .PreserveC           (PreserveC),
   .Arith16             (Arith16),
   .Set_Addr_To         (Set_Addr_To),
   .IORQ                (iorq_i),
   .Jump                (Jump),
   .JumpE               (JumpE),
   .JumpXY              (JumpXY),
   .Call                (Call),
```

```
   .RstP                (RstP),
   .LDZ                 (LDZ),
   .LDW                 (LDW),
   .LDSPHL              (LDSPHL),
   .Special_LD          (Special_LD),
   .ExchangeDH          (ExchangeDH),
   .ExchangeRp          (ExchangeRp),
   .ExchangeAF          (ExchangeAF),
   .ExchangeRS          (ExchangeRS),
   .I_DJNZ              (I_DJNZ),
   .I_CPL               (I_CPL),
   .I_CCF               (I_CCF),
   .I_SCF               (I_SCF),
   .I_RETN              (I_RETN),
   .I_BT                (I_BT),
   .I_BC                (I_BC),
   .I_BTR               (I_BTR),
   .I_RLD               (I_RLD),
   .I_RRD               (I_RRD),
   .I_INRC              (I_INRC),
   .SetDI               (SetDI),
   .SetEI               (SetEI),
   .IMode               (IMode),
   .Halt                (Halt),
   .NoRead              (no_read),
   .Write               (write)
   );

tv80_alu #(Mode, Flag_C, Flag_N, Flag_P, Flag_X, Flag_H, Flag_Y, Flag_Z, Flag_S) i_alu
  (
   .Arith16             (Arith16_r),
   .Z16                 (Z16_r),
   .ALU_Op              (ALU_Op_r),
   .IR                  (IR[5:0]),
   .ISet                (ISet),
   .BusA                (BusA),
   .BusB                (BusB),
   .F_In                (F),
   .Q                   (ALU_Q),
   .F_Out               (F_Out)
   );

function [6:0] number_to_bitvec;
  input [2:0] num;
  begin
    case (num)
      1 : number_to_bitvec = 7'b0000001;
      2 : number_to_bitvec = 7'b0000010;
      3 : number_to_bitvec = 7'b0000100;
      4 : number_to_bitvec = 7'b0001000;
      5 : number_to_bitvec = 7'b0010000;
      6 : number_to_bitvec = 7'b0100000;
      7 : number_to_bitvec = 7'b1000000;
      default : number_to_bitvec = 7'bx;
    endcase // case(num)
  end
endfunction // number_to_bitvec

always @(/*AUTOSENSE*/mcycle or mcycles or tstate or tstates)
  begin
    case (mcycles)
      1 : last_mcycle = mcycle[0];
      2 : last_mcycle = mcycle[1];
      3 : last_mcycle = mcycle[2];
      4 : last_mcycle = mcycle[3];
      5 : last_mcycle = mcycle[4];
      6 : last_mcycle = mcycle[5];
      7 : last_mcycle = mcycle[6];
      default : last_mcycle = 1'bx;
    endcase // case(mcycles)
```

```
    case (tstates)
      0 : last_tstate = tstate[0];
      1 : last_tstate = tstate[1];
      2 : last_tstate = tstate[2];
      3 : last_tstate = tstate[3];
      4 : last_tstate = tstate[4];
      5 : last_tstate = tstate[5];
      6 : last_tstate = tstate[6];
      default : last_tstate = 1'bx;
    endcase
  end // always @ (...


always @(/*AUTOSENSE*/ALU_Q or BusAck or BusB or DI_Reg
         or ExchangeRp or IR or Save_ALU_r or Set_Addr_To or XY_Ind
         or XY_State or cen or last_tstate or mcycle)
  begin
    ClkEn = cen && ~ BusAck;

    if (last_tstate)
      T_Res = 1'b1;
    else T_Res = 1'b0;

    if (XY_State != 2'b00 && XY_Ind == 1'b0 &&
        ((Set_Addr_To == aXY) ||
         (mcycle[0] && IR == 8'b11001011) ||
         (mcycle[0] && IR == 8'b00110110)))
      NextIs_XY_Fetch = 1'b1;
    else
      NextIs_XY_Fetch = 1'b0;

    if (ExchangeRp)
      Save_Mux = BusB;
    else if (!Save_ALU_r)
      Save_Mux = DI_Reg;
    else
      Save_Mux = ALU_Q;
  end // always @ *

always @ (posedge clk)
  begin
    if (reset_n == 1'b0 )
      begin
        PC <= #1 0;  // Program Counter
        A <= #1 0;
        TmpAddr <= #1 0;
        IR <= #1 8'b00000000;
        ISet <= #1 2'b00;
        XY_State <= #1 2'b00;
        IStatus <= #1 2'b00;
        mcycles <= #1 3'b000;
        do <= #1 8'b00000000;

        ACC <= #1 8'hFF;
        F <= #1 8'hFF;
        Ap <= #1 8'hFF;
        Fp <= #1 8'hFF;
        I <= #1 8'hFE; // 0
        `ifdef TV80_REFRESH
        R <= #1 0;
        `endif
        SP <= #1 16'hFFFF;
        Alternate <= #1 1'b0;

        Read_To_Reg_r <= #1 5'b00000;
        Arith16_r <= #1 1'b0;
        BTR_r <= #1 1'b0;
        Z16_r <= #1 1'b0;
        ALU_Op_r <= #1 4'b0000;
        Save_ALU_r <= #1 1'b0;
        PreserveC_r <= #1 1'b0;
```

```
          XY_Ind <= #1 1'b0;
        end
      else
        begin

          if (ClkEn == 1'b1 )
            begin

              ALU_Op_r <= #1 4'b0000;
              Save_ALU_r <= #1 1'b0;
              Read_To_Reg_r <= #1 5'b00000;

              mcycles <= #1 mcycles_d;

              if (IMode != 2'b11 )
                begin
                  IStatus <= #1 IMode;
                end

              Arith16_r <= #1 Arith16;
              PreserveC_r <= #1 PreserveC;
              if (ISet == 2'b10 && ALU_Op[2] == 1'b0 && ALU_Op[0] == 1'b1 && mcycle[2] )
                begin
                  Z16_r <= #1 1'b1;
                end
              else
                begin
                  Z16_r <= #1 1'b0;
                end

              if (mcycle[0] && (tstate[1] | tstate[2] | tstate[3] ))
                begin
                  // mcycle == 1 && tstate == 1, 2, || 3
                  if (tstate[2] && wait_n == 1'b1 )
                    begin
                      `ifdef TV80_REFRESH
                      if (Mode < 2 )
                        begin
                          A[7:0] <= #1 R;
                          A[15:8] <= #1 I;
                          R[6:0] <= #1 R[6:0] + 1;
                        end
                      `endif
                      if (Jump == 1'b0 && Call == 1'b0 && NMICycle == 1'b0 && IntCycle == 1'b0 &&
~ (Halt_FF == 1'b1 || Halt == 1'b1) )
                        begin
                          PC <= #1 PC16;
                        end

                      if (IntCycle == 1'b1 && IStatus == 2'b01 )
                        begin
                          IR <= #1 8'b11111111;
                        end
                      else if (Halt_FF == 1'b1 || (IntCycle == 1'b1 && IStatus == 2'b10) ||
NMICycle == 1'b1 )
                        begin
                          IR <= #1 8'b00000000;
                        end
                      else
                        begin
                          IR <= #1 dinst;
                        end

                      ISet <= #1 2'b00;
                      if (Prefix != 2'b00 )
                        begin
                          if (Prefix == 2'b11 )
                            begin
                              if (IR[5] == 1'b1 )
                                begin
                                  XY_State <= #1 2'b10;
```

```
                end
              else
                begin
                  XY_State <= #1 2'b01;
                end
            end
          else
            begin
              if (Prefix == 2'b10 )
                begin
                  XY_State <= #1 2'b00;
                  XY_Ind <= #1 1'b0;
                end
              ISet <= #1 Prefix;
            end
        end
      else
        begin
          XY_State <= #1 2'b00;
          XY_Ind <= #1 1'b0;
        end
    end // if (tstate == 2 && wait_n == 1'b1 )


  end
else
  begin
    // either (mcycle > 1) OR (mcycle == 1 AND tstate > 3)

    if (mcycle[5] )
      begin
        XY_Ind <= #1 1'b1;
        if (Prefix == 2'b01 )
          begin
            ISet <= #1 2'b01;
          end
      end

    if (T_Res == 1'b1 )
      begin
        BTR_r <= #1 (I_BT || I_BC || I_BTR) && ~ No_BTR;
        if (Jump == 1'b1 )
          begin
            A[15:8] <= #1 DI_Reg;
            A[7:0] <= #1 TmpAddr[7:0];
            PC[15:8] <= #1 DI_Reg;
            PC[7:0] <= #1 TmpAddr[7:0];
          end
        else if (JumpXY == 1'b1 )
          begin
            A <= #1 RegBusC;
            PC <= #1 RegBusC;
          end else if (Call == 1'b1 || RstP == 1'b1 )
            begin
              A <= #1 TmpAddr;
              PC <= #1 TmpAddr;
            end
          else if (last_mcycle && NMICycle == 1'b1 )
            begin
              A <= #1 16'b0000000001100110;
              PC <= #1 16'b0000000001100110;
            end
          else if (mcycle[2] && IntCycle == 1'b1 && IStatus == 2'b10 )
            begin
              A[15:8] <= #1 I;
              A[7:0] <= #1 TmpAddr[7:0];
              PC[15:8] <= #1 I;
              PC[7:0] <= #1 TmpAddr[7:0];
            end
          else
            begin
```

```
case (Set_Addr_To)
  aXY :
    begin
      if (XY_State == 2'b00 )
        begin
          A <= #1 RegBusC;
        end
      else
        begin
          if (NextIs_XY_Fetch == 1'b1 )
            begin
              A <= #1 PC;
            end
          else
            begin
              A <= #1 TmpAddr;
            end
        end // else: !if(XY_State == 2'b00 )
    end // case: aXY

  aIOA :
    begin
      if (Mode == 3 )
        begin
          // Memory map I/O on GBZ80
          A[15:8] <= #1 8'hFF;
        end
      else if (Mode == 2 )
        begin
          // Duplicate I/O address on 8080
          A[15:8] <= #1 DI_Reg;
        end
      else
        begin
          A[15:8] <= #1 ACC;
        end
      A[7:0] <= #1 DI_Reg;
    end // case: aIOA


  aSP :
    begin
      A <= #1 SP;
    end

  aBC :
    begin
      if (Mode == 3 && iorq_i == 1'b1 )
        begin
          // Memory map I/O on GBZ80
          A[15:8] <= #1 8'hFF;
          A[7:0] <= #1 RegBusC[7:0];
        end
      else
        begin
          A <= #1 RegBusC;
        end
    end // case: aBC

  aDE :
    begin
      A <= #1 RegBusC;
    end

  aZI :
    begin
      if (Inc_WZ == 1'b1 )
        begin
          A <= #1 TmpAddr + 1;
        end
      else
```

```
                    begin
                      A[15:8] <= #1 DI_Reg;
                      A[7:0] <= #1 TmpAddr[7:0];
                    end
                end // case: aZI

              default   :
                begin
                  A <= #1 PC;
                end
            endcase // case(Set_Addr_To)

        end // else: !if(mcycle[2] && IntCycle == 1'b1 && IStatus == 2'b10 )


    Save_ALU_r <= #1 Save_ALU;
    ALU_Op_r <= #1 ALU_Op;

    if (I_CPL == 1'b1 )
      begin
        // CPL
        ACC <= #1 ~ ACC;
        F[Flag_Y] <= #1 ~ ACC[5];
        F[Flag_H] <= #1 1'b1;
        F[Flag_X] <= #1 ~ ACC[3];
        F[Flag_N] <= #1 1'b1;
      end
    if (I_CCF == 1'b1 )
      begin
        // CCF
        F[Flag_C] <= #1 ~ F[Flag_C];
        F[Flag_Y] <= #1 ACC[5];
        F[Flag_H] <= #1 F[Flag_C];
        F[Flag_X] <= #1 ACC[3];
        F[Flag_N] <= #1 1'b0;
      end
    if (I_SCF == 1'b1 )
      begin
        // SCF
        F[Flag_C] <= #1 1'b1;
        F[Flag_Y] <= #1 ACC[5];
        F[Flag_H] <= #1 1'b0;
        F[Flag_X] <= #1 ACC[3];
        F[Flag_N] <= #1 1'b0;
      end
  end // if (T_Res == 1'b1 )


if (tstate[2] && wait_n == 1'b1 )
  begin
    if (ISet == 2'b01 && mcycle[6] )
      begin
        IR <= #1 dinst;
      end
    if (JumpE == 1'b1 )
      begin
        PC <= #1 PC16;
      end
    else if (Inc_PC == 1'b1 )
      begin
        //PC <= #1 PC + 1;
        PC <= #1 PC16;
      end
    if (BTR_r == 1'b1 )
      begin
        //PC <= #1 PC - 2;
        PC <= #1 PC16;
      end
    if (RstP == 1'b1 )
      begin
        TmpAddr <= #1 { 10'h0, IR[5:3], 3'h0 };
```

```
            //TmpAddr <= #1 (others =>1'b0);
            //TmpAddr[5:3] <= #1 IR[5:3];
          end
      end
    if (tstate[3] && mcycle[5] )
      begin
        TmpAddr <= #1 SP16;
      end

    if ((tstate[2] && wait_n == 1'b1) || (tstate[4] && mcycle[0]) )
      begin
        if (IncDec_16[2:0] == 3'b111 )
          begin
            SP <= #1 SP16;
          end
      end

    if (LDSPHL == 1'b1 )
      begin
        SP <= #1 RegBusC;
      end
    if (ExchangeAF == 1'b1 )
      begin
        Ap <= #1 ACC;
        ACC <= #1 Ap;
        Fp <= #1 F;
        F <= #1 Fp;
      end
    if (ExchangeRS == 1'b1 )
      begin
        Alternate <= #1 ~ Alternate;
      end
  end // else: !if(mcycle  == 3'b001 && tstate(2) == 1'b0 )


if (tstate[3] )
  begin
    if (LDZ == 1'b1 )
      begin
        TmpAddr[7:0] <= #1 DI_Reg;
      end
    if (LDW == 1'b1 )
      begin
        TmpAddr[15:8] <= #1 DI_Reg;
      end

    if (Special_LD[2] == 1'b1 )
      begin
        case (Special_LD[1:0])
          2'b00 :
            begin
              ACC <= #1 I;
              F[Flag_P] <= #1 IntE_FF2;
            end

          2'b01 :
            begin
              ACC <= #1 R;
              F[Flag_P] <= #1 IntE_FF2;
            end

          2'b10 :
            I <= #1 ACC;

          `ifdef TV80_REFRESH
          default :
            R <= #1 ACC;
          `else
          default : ;
          `endif
        endcase
```

```
      end
  end // if (tstate == 3 )


if ((I_DJNZ == 1'b0 && Save_ALU_r == 1'b1) || ALU_Op_r == 4'b1001 )
  begin
    if (Mode == 3 )
      begin
        F[6] <= #1 F_Out[6];
        F[5] <= #1 F_Out[5];
        F[7] <= #1 F_Out[7];
        if (PreserveC_r == 1'b0 )
          begin
            F[4] <= #1 F_Out[4];
          end
      end
    else
      begin
        F[7:1] <= #1 F_Out[7:1];
        if (PreserveC_r == 1'b0 )
          begin
            F[Flag_C] <= #1 F_Out[0];
          end
      end
  end // if ((I_DJNZ == 1'b0 && Save_ALU_r == 1'b1) || ALU_Op_r == 4'b1001 )

if (T_Res == 1'b1 && I_INRC == 1'b1 )
  begin
    F[Flag_H] <= #1 1'b0;
    F[Flag_N] <= #1 1'b0;
    if (DI_Reg[7:0] == 8'b00000000 )
      begin
        F[Flag_Z] <= #1 1'b1;
      end
    else
      begin
        F[Flag_Z] <= #1 1'b0;
      end
    F[Flag_S] <= #1 DI_Reg[7];
    F[Flag_P] <= #1 ~ (^DI_Reg[7:0]);
  end // if (T_Res == 1'b1 && I_INRC == 1'b1 )


if (tstate[1] && Auto_Wait_t1 == 1'b0 )
  begin
    do <= #1 BusB;
    if (I_RLD == 1'b1 )
      begin
        do[3:0] <= #1 BusA[3:0];
        do[7:4] <= #1 BusB[3:0];
      end
    if (I_RRD == 1'b1 )
      begin
        do[3:0] <= #1 BusB[7:4];
        do[7:4] <= #1 BusA[3:0];
      end
  end

if (T_Res == 1'b1 )
  begin
    Read_To_Reg_r[3:0] <= #1 Set_BusA_To;
    Read_To_Reg_r[4] <= #1 Read_To_Reg;
    if (Read_To_Acc == 1'b1 )
      begin
        Read_To_Reg_r[3:0] <= #1 4'b0111;
        Read_To_Reg_r[4] <= #1 1'b1;
      end
  end

if (tstate[1] && I_BT == 1'b1 )
  begin
```

```
              F[Flag_X] <= #1 ALU_Q[3];
              F[Flag_Y] <= #1 ALU_Q[1];
              F[Flag_H] <= #1 1'b0;
              F[Flag_N] <= #1 1'b0;
            end
          if (I_BC == 1'b1 || I_BT == 1'b1 )
            begin
              F[Flag_P] <= #1 IncDecZ;
            end

          if ((tstate[1] && Save_ALU_r == 1'b0 && Auto_Wait_t1 == 1'b0) ||
              (Save_ALU_r == 1'b1 && ALU_Op_r != 4'b0111) )
            begin
              case (Read_To_Reg_r)
                5'b10111 :
                  ACC <= #1 Save_Mux;
                5'b10110 :
                  do <= #1 Save_Mux;
                5'b11000 :
                  SP[7:0] <= #1 Save_Mux;
                5'b11001 :
                  SP[15:8] <= #1 Save_Mux;
                5'b11011 :
                  F <= #1 Save_Mux;
              endcase
            end // if ((tstate == 1 && Save_ALU_r == 1'b0 && Auto_Wait_t1 == 1'b0) ||...
        end // if (ClkEn == 1'b1 )
    end // else: !if(reset_n == 1'b0 )
  end


//-------------------------------------------------------------------------
//
// BC('), DE('), HL('), IX && IY
//
//-------------------------------------------------------------------------
always @ (posedge clk)
  begin
    if (ClkEn == 1'b1 )
      begin
        // Bus A / Write
        RegAddrA_r <= #1  { Alternate, Set_BusA_To[2:1] };
        if (XY_Ind == 1'b0 && XY_State != 2'b00 && Set_BusA_To[2:1] == 2'b10 )
          begin
            RegAddrA_r <= #1 { XY_State[1],  2'b11 };
          end

        // Bus B
        RegAddrB_r <= #1 { Alternate, Set_BusB_To[2:1] };
        if (XY_Ind == 1'b0 && XY_State != 2'b00 && Set_BusB_To[2:1] == 2'b10 )
          begin
            RegAddrB_r <= #1 { XY_State[1],  2'b11 };
          end

        // Address from register
        RegAddrC <= #1 { Alternate,  Set_Addr_To[1:0] };
        // Jump (HL), LD SP,HL
        if ((JumpXY == 1'b1 || LDSPHL == 1'b1) )
          begin
            RegAddrC <= #1 { Alternate, 2'b10 };
          end
        if (((JumpXY == 1'b1 || LDSPHL == 1'b1) && XY_State != 2'b00) || (mcycle[5]) )
          begin
            RegAddrC <= #1 { XY_State[1],  2'b11 };
          end

        if (I_DJNZ == 1'b1 && Save_ALU_r == 1'b1 && Mode < 2 )
          begin
            IncDecZ <= #1 F_Out[Flag_Z];
          end
        if ((tstate[2] || (tstate[3] && mcycle[0])) && IncDec_16[2:0] == 3'b100 )
```

```
            begin
              if (ID16 == 0 )
                begin
                  IncDecZ <= #1 1'b0;
                end
              else
                begin
                  IncDecZ <= #1 1'b1;
                end
            end

          RegBusA_r <= #1 RegBusA;
        end

    end // always @ (posedge clk)


  always @(/*AUTOSENSE*/Alternate or ExchangeDH or IncDec_16
           or RegAddrA_r or RegAddrB_r or XY_State or mcycle or tstate)
    begin
      if ((tstate[2] || (tstate[3] && mcycle[0] && IncDec_16[2] == 1'b1)) && XY_State == 2'b00)
        RegAddrA = { Alternate, IncDec_16[1:0] };
      else if ((tstate[2] || (tstate[3] && mcycle[0] && IncDec_16[2] == 1'b1)) && IncDec_16[1:0]
== 2'b10)
        RegAddrA = { XY_State[1], 2'b11 };
      else if (ExchangeDH == 1'b1 && tstate[3])
        RegAddrA = { Alternate, 2'b10 };
      else if (ExchangeDH == 1'b1 && tstate[4])
        RegAddrA = { Alternate, 2'b01 };
      else
        RegAddrA = RegAddrA_r;

      if (ExchangeDH == 1'b1 && tstate[3])
        RegAddrB = { Alternate, 2'b01 };
      else
        RegAddrB = RegAddrB_r;
    end // always @ *


  always @(/*AUTOSENSE*/ALU_Op_r or Auto_Wait_t1 or ExchangeDH
           or IncDec_16 or Read_To_Reg_r or Save_ALU_r or mcycle
           or tstate or wait_n)
    begin
      RegWEH = 1'b0;
      RegWEL = 1'b0;
      if ((tstate[1] && Save_ALU_r == 1'b0 && Auto_Wait_t1 == 1'b0) ||
          (Save_ALU_r == 1'b1 && ALU_Op_r != 4'b0111) )
        begin
          case (Read_To_Reg_r)
            5'b10000 , 5'b10001 , 5'b10010 , 5'b10011 , 5'b10100 , 5'b10101 :
              begin
                RegWEH = ~ Read_To_Reg_r[0];
                RegWEL = Read_To_Reg_r[0];
              end
          endcase // case(Read_To_Reg_r)

        end // if ((tstate == 1 && Save_ALU_r == 1'b0 && Auto_Wait_t1 == 1'b0) ||...


      if (ExchangeDH == 1'b1 && (tstate[3] || tstate[4]) )
        begin
          RegWEH = 1'b1;
          RegWEL = 1'b1;
        end

      if (IncDec_16[2] == 1'b1 && ((tstate[2] && wait_n == 1'b1 && mcycle != 3'b001) ||
(tstate[3] && mcycle[0])) )
        begin
          case (IncDec_16[1:0])
            2'b00 , 2'b01 , 2'b10 :
              begin
```

```
                RegWEH = 1'b1;
                RegWEL = 1'b1;
              end
          endcase
        end
    end // always @ *


  always @(/*AUTOSENSE*/ExchangeDH or ID16 or IncDec_16 or RegBusA_r
           or RegBusB or Save_Mux or mcycle or tstate)
    begin
      RegDIH = Save_Mux;
      RegDIL = Save_Mux;

      if (ExchangeDH == 1'b1 && tstate[3] )
        begin
          RegDIH = RegBusB[15:8];
          RegDIL = RegBusB[7:0];
        end
      else if (ExchangeDH == 1'b1 && tstate[4] )
        begin
          RegDIH = RegBusA_r[15:8];
          RegDIL = RegBusA_r[7:0];
        end
      else if (IncDec_16[2] == 1'b1 && ((tstate[2] && mcycle != 3'b001) || (tstate[3] &&
mcycle[0])) )
        begin
          RegDIH = ID16[15:8];
          RegDIL = ID16[7:0];
        end
    end

  tv80_reg i_reg
    (
     .clk                 (clk),
     .CEN                 (ClkEn),
     .WEH                 (RegWEH),
     .WEL                 (RegWEL),
     .AddrA               (RegAddrA),
     .AddrB               (RegAddrB),
     .AddrC               (RegAddrC),
     .DIH                 (RegDIH),
     .DIL                 (RegDIL),
     .DOAH                (RegBusA[15:8]),
     .DOAL                (RegBusA[7:0]),
     .DOBH                (RegBusB[15:8]),
     .DOBL                (RegBusB[7:0]),
     .DOCH                (RegBusC[15:8]),
     .DOCL                (RegBusC[7:0]),
         .BC                  (BC),
         .DE                  (DE),
         .HL                  (HL)
     );

  //-------------------------------------------------------------------------
  //
  // Buses
  //
  //-------------------------------------------------------------------------

  always @ (posedge clk)
    begin
      if (ClkEn == 1'b1 )
        begin
          case (Set_BusB_To)
            4'b0111 :
              BusB <= #1 ACC;
            4'b0000 , 4'b0001 , 4'b0010 , 4'b0011 , 4'b0100 , 4'b0101 :
              begin
                if (Set_BusB_To[0] == 1'b1 )
                  begin
```

```
                BusB <= #1 RegBusB[7:0];
              end
            else
              begin
                BusB <= #1 RegBusB[15:8];
              end
          end
        4'b0110 :
          BusB <= #1 DI_Reg;
        4'b1000 :
          BusB <= #1 SP[7:0];
        4'b1001 :
          BusB <= #1 SP[15:8];
        4'b1010 :
          BusB <= #1 8'b00000001;
        4'b1011 :
          BusB <= #1 F;
        4'b1100 :
          BusB <= #1 PC[7:0];
        4'b1101 :
          BusB <= #1 PC[15:8];
        4'b1110 :
          BusB <= #1 8'b00000000;
        default :
          BusB <= #1 8'hxx;
      endcase

      case (Set_BusA_To)
        4'b0111 :
          BusA <= #1 ACC;
        4'b0000 , 4'b0001 , 4'b0010 , 4'b0011 , 4'b0100 , 4'b0101 :
          begin
            if (Set_BusA_To[0] == 1'b1 )
              begin
                BusA <= #1 RegBusA[7:0];
              end
            else
              begin
                BusA <= #1 RegBusA[15:8];
              end
          end
        4'b0110 :
          BusA <= #1 DI_Reg;
        4'b1000 :
          BusA <= #1 SP[7:0];
        4'b1001 :
          BusA <= #1 SP[15:8];
        4'b1010 :
          BusA <= #1 8'b00000000;
        default :
          BusB <= #1  8'hxx;
      endcase
    end
  end

  //-------------------------------------------------------------------------
  //
  // Generate external control signals
  //
  //-------------------------------------------------------------------------
`ifdef TV80_REFRESH
  always @ (posedge clk)
    begin
      if (reset_n == 1'b0 )
        begin
          rfsh_n <= #1 1'b1;
        end
      else
        begin
          if (cen == 1'b1 )
            begin
```

```
            if (mcycle[0] && ((tstate[2]  && wait_n == 1'b1) || tstate[3]) )
              begin
                rfsh_n <= #1 1'b0;
              end
            else
              begin
                rfsh_n <= #1 1'b1;
              end
          end
      end
  end
`endif

  always @(/*AUTOSENSE*/BusAck or Halt_FF or I_DJNZ or IntCycle
           or IntE_FF1 or di or iorq_i or mcycle or tstate)
    begin
      mc = mcycle;
      ts = tstate;
      DI_Reg = di;
      halt_n = ~ Halt_FF;
      busak_n = ~ BusAck;
      intcycle_n = ~ IntCycle;
      IntE = IntE_FF1;
      iorq = iorq_i;
      stop = I_DJNZ;
    end

  //-------------------------------------------------------------------------
  //
  // Syncronise inputs
  //
  //-------------------------------------------------------------------------

  always @ (posedge clk)
    begin : sync_inputs

      if (reset_n == 1'b0 )
        begin
          BusReq_s <= #1 1'b0;
          INT_s <= #1 1'b0;
          NMI_s <= #1 1'b0;
          Oldnmi_n <= #1 1'b0;
        end
      else
        begin
          if (cen == 1'b1 )
            begin
              BusReq_s <= #1 ~ busrq_n;
              INT_s <= #1 ~ int_n;
              if (NMICycle == 1'b1 )
                begin
                  NMI_s <= #1 1'b0;
                end
              else if (nmi_n == 1'b0 && Oldnmi_n == 1'b1 )
                begin
                  NMI_s <= #1 1'b1;
                end
              Oldnmi_n <= #1 nmi_n;
            end
        end
    end

  //-------------------------------------------------------------------------
  //
  // Main state machine
  //
  //-------------------------------------------------------------------------

  always @ (posedge clk)
    begin
      if (reset_n == 1'b0 )
```

```
  begin
    mcycle <= #1 7'b0000001;
    tstate <= #1 7'b0000001;
    Pre_XY_F_M <= #1 3'b000;
    Halt_FF <= #1 1'b0;
    BusAck <= #1 1'b0;
    NMICycle <= #1 1'b0;
    IntCycle <= #1 1'b0;
    IntE_FF1 <= #1 1'b0;
    IntE_FF2 <= #1 1'b0;
    No_BTR <= #1 1'b0;
    Auto_Wait_t1 <= #1 1'b0;
    Auto_Wait_t2 <= #1 1'b0;
    m1_n <= #1 1'b1;
  end
else
  begin
    if (cen == 1'b1 )
      begin
        if (T_Res == 1'b1 )
          begin
            Auto_Wait_t1 <= #1 1'b0;
          end
        else
          begin
            Auto_Wait_t1 <= #1 Auto_Wait || iorq_i;
          end
        Auto_Wait_t2 <= #1 Auto_Wait_t1;
        No_BTR <= #1 (I_BT && (~ IR[4] || ~ F[Flag_P])) ||
                  (I_BC && (~ IR[4] || F[Flag_Z] || ~ F[Flag_P])) ||
                  (I_BTR && (~ IR[4] || F[Flag_Z]));
        if (tstate[2] )
          begin
            if (SetEI == 1'b1 )
              begin
                IntE_FF1 <= #1 1'b1;
                IntE_FF2 <= #1 1'b1;
              end
            if (I_RETN == 1'b1 )
              begin
                IntE_FF1 <= #1 IntE_FF2;
              end
          end
        if (tstate[3] )
          begin
            if (SetDI == 1'b1 )
              begin
                IntE_FF1 <= #1 1'b0;
                IntE_FF2 <= #1 1'b0;
              end
          end
        if (IntCycle == 1'b1 || NMICycle == 1'b1 )
          begin
            Halt_FF <= #1 1'b0;
          end
        if (mcycle[0] && tstate[2] && wait_n == 1'b1 )
          begin
            m1_n <= #1 1'b1;
          end
        if (BusReq_s == 1'b1 && BusAck == 1'b1 )
          begin
          end
        else
          begin
            BusAck <= #1 1'b0;
            if (tstate[2] && wait_n == 1'b0 )
              begin
              end
            else if (T_Res == 1'b1 )
              begin
                if (Halt == 1'b1 )
```

```
                        begin
                          Halt_FF <= #1 1'b1;
                        end
                      if (BusReq_s == 1'b1 )
                        begin
                          BusAck <= #1 1'b1;
                        end
                      else
                        begin
                          tstate <= #1 7'b0000010;
                          if (NextIs_XY_Fetch == 1'b1 )
                            begin
                              mcycle <= #1 7'b0100000;
                              Pre_XY_F_M <= #1 mcycle;
                              if (IR == 8'b00110110 && Mode == 0 )
                                begin
                                  Pre_XY_F_M <= #1 3'b010;
                                end
                            end
                          else if ((mcycle[6]) || (mcycle[5] && Mode == 1 && ISet != 2'b01) )
                            begin
                              mcycle <= #1 number_to_bitvec(Pre_XY_F_M + 1);
                            end
                          else if ((last_mcycle) ||
                                   No_BTR == 1'b1 ||
                                   (mcycle[1] && I_DJNZ == 1'b1 && IncDecZ == 1'b1) )
                            begin
                              m1_n <= #1 1'b0;
                              mcycle <= #1 7'b0000001;
                              IntCycle <= #1 1'b0;
                              NMICycle <= #1 1'b0;
                              if (NMI_s == 1'b1 && Prefix == 2'b00 )
                                begin
                                  NMICycle <= #1 1'b1;
                                  IntE_FF1 <= #1 1'b0;
                                end
                              else if ((IntE_FF1 == 1'b1 && INT_s == 1'b1) && Prefix == 2'b00 &&
SetEI == 1'b0 )
                                begin
                                  IntCycle <= #1 1'b1;
                                  IntE_FF1 <= #1 1'b0;
                                  IntE_FF2 <= #1 1'b0;
                                end
                            end
                          else
                            begin
                              mcycle <= #1 { mcycle[5:0], mcycle[6] };
                            end
                        end
                    end
                  else
                    begin   // verilog has no "nor" operator
                      if ( ~(Auto_Wait == 1'b1 && Auto_Wait_t2 == 1'b0) &&
                           ~(IOWait == 1 && iorq_i == 1'b1 && Auto_Wait_t1 == 1'b0) )
                        begin
                          tstate <= #1 { tstate[5:0], tstate[6] };
                        end
                    end
                end
              if (tstate[0])
                begin
                  m1_n <= #1 1'b0;
                end
            end
        end
    end

  always @(/*AUTOSENSE*/BTR_r or DI_Reg or IncDec_16 or JumpE or PC
           or RegBusA or RegBusC or SP or tstate)
    begin
      if (JumpE == 1'b1 )
```

```
        begin
          PC16_B = { {8{DI_Reg[7]}}, DI_Reg };
        end
      else if (BTR_r == 1'b1 )
        begin
          PC16_B = -2;
        end
      else
        begin
          PC16_B = 1;
        end

      if (tstate[3])
        begin
          SP16_A = RegBusC;
          SP16_B = { {8{DI_Reg[7]}}, DI_Reg };
        end
      else
        begin
          // suspect that ID16 and SP16 could be shared
          SP16_A = SP;

          if (IncDec_16[3] == 1'b1)
            SP16_B = -1;
          else
            SP16_B = 1;
        end

      if (IncDec_16[3])
        ID16_B = -1;
      else
        ID16_B = 1;

      ID16 = RegBusA + ID16_B;
      PC16 = PC + PC16_B;
      SP16 = SP16_A + SP16_B;
    end // always @ *


  always @(/*AUTOSENSE*/IntCycle or NMICycle or mcycle)
    begin
      Auto_Wait = 1'b0;
      if (IntCycle == 1'b1 || NMICycle == 1'b1 )
        begin
          if (mcycle[0] )
            begin
              Auto_Wait = 1'b1;
            end
        end
    end // always @ *

// synopsys dc_script_begin
// set_attribute current_design "revision" "$Id: tv80_core.v,v 1.5 2005-01-26 18:55:47 ghutchis
Exp $" -type string -quiet
// synopsys dc_script_end
endmodule // T80
```

tv80_mcode.v

```
//
// TV80 8-Bit Microprocessor Core
// Based on the VHDL T80 core by Daniel Wallner (jesus@opencores.org)
//
// Copyright (c) 2004,2007 Guy Hutchison (ghutchis@opencores.org)
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
```

```
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included
// in all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
// EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
// MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
// IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
// CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
// TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
// SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

module tv80_mcode
  (/*AUTOARG*/
  // Outputs
  MCycles, TStates, Prefix, Inc_PC, Inc_WZ, IncDec_16, Read_To_Reg,
  Read_To_Acc, Set_BusA_To, Set_BusB_To, ALU_Op, Save_ALU, PreserveC,
  Arith16, Set_Addr_To, IORQ, Jump, JumpE, JumpXY, Call, RstP, LDZ,
  LDW, LDSPHL, Special_LD, ExchangeDH, ExchangeRp, ExchangeAF,
  ExchangeRS, I_DJNZ, I_CPL, I_CCF, I_SCF, I_RETN, I_BT, I_BC, I_BTR,
  I_RLD, I_RRD, I_INRC, SetDI, SetEI, IMode, Halt, NoRead, Write,
  // Inputs
  IR, ISet, MCycle, F, NMICycle, IntCycle
  );

  parameter             Mode   = 3;
  parameter             Flag_C = 0;
  parameter             Flag_N = 1;
  parameter             Flag_P = 2;
  parameter             Flag_X = 3;
  parameter             Flag_H = 4;
  parameter             Flag_Y = 5;
  parameter             Flag_Z = 6;
  parameter             Flag_S = 7;

  input [7:0]           IR;
  input [1:0]           ISet                    ;
  input [6:0]           MCycle                  ;
  input [7:0]           F                       ;
  input                 NMICycle                ;
  input                 IntCycle                ;
  output [2:0]          MCycles                 ;
  output [2:0]          TStates                 ;
  output [1:0]          Prefix                  ; // None,BC,ED,DD/FD
  output                Inc_PC                  ;
  output                Inc_WZ                  ;
  output [3:0]          IncDec_16               ; // BC,DE,HL,SP   0 is inc
  output                Read_To_Reg             ;
  output                Read_To_Acc             ;
  output [3:0]          Set_BusA_To     ; // B,C,D,E,H,L,DI/DB,A,SP(L),SP(M),0,F
  output [3:0]          Set_BusB_To     ; // B,C,D,E,H,L,DI,A,SP(L),SP(M),1,F,PC(L),PC(M),0
  output [3:0]          ALU_Op                  ;
  output                Save_ALU                ;
  output                PreserveC               ;
  output                Arith16                 ;
  output [2:0]          Set_Addr_To             ; // aNone,aXY,aIOA,aSP,aBC,aDE,aZI
  output                IORQ                    ;
  output                Jump                    ;
  output                JumpE                   ;
  output                JumpXY                  ;
  output                Call                    ;
  output                RstP                    ;
  output                LDZ                     ;
  output                LDW                     ;
  output                LDSPHL                  ;
  output [2:0]          Special_LD              ; // A,I;A,R;I,A;R,A;None
  output                ExchangeDH              ;
  output                ExchangeRp              ;
  output                ExchangeAF              ;
```

```
output               ExchangeRS             ;
output               I_DJNZ                 ;
output               I_CPL                  ;
output               I_CCF                  ;
output               I_SCF                  ;
output               I_RETN                 ;
output               I_BT                   ;
output               I_BC                   ;
output               I_BTR                  ;
output               I_RLD                  ;
output               I_RRD                  ;
output               I_INRC                 ;
output               SetDI                  ;
output               SetEI                  ;
output [1:0]         IMode                  ;
output               Halt                   ;
output               NoRead                 ;
output               Write   ;

// regs
reg [2:0]            MCycles                ;
reg [2:0]            TStates                ;
reg [1:0]            Prefix                 ; // None,BC,ED,DD/FD
reg                  Inc_PC                 ;
reg                  Inc_WZ                 ;
reg [3:0]            IncDec_16              ; // BC,DE,HL,SP   0 is inc
reg                  Read_To_Reg            ;
reg                  Read_To_Acc            ;
reg [3:0]            Set_BusA_To     ; // B,C,D,E,H,L,DI/DB,A,SP(L),SP(M),0,F
reg [3:0]            Set_BusB_To     ; // B,C,D,E,H,L,DI,A,SP(L),SP(M),1,F,PC(L),PC(M),0
reg [3:0]            ALU_Op                 ;
reg                  Save_ALU               ;
reg                  PreserveC              ;
reg                  Arith16                ;
reg [2:0]            Set_Addr_To            ; // aNone,aXY,aIOA,aSP,aBC,aDE,aZI
reg                  IORQ                   ;
reg                  Jump                   ;
reg                  JumpE                  ;
reg                  JumpXY                 ;
reg                  Call                   ;
reg                  RstP                   ;
reg                  LDZ                    ;
reg                  LDW                    ;
reg                  LDSPHL                 ;
reg [2:0]            Special_LD             ; // A,I;A,R;I,A;R,A;None
reg                  ExchangeDH             ;
reg                  ExchangeRp             ;
reg                  ExchangeAF             ;
reg                  ExchangeRS             ;
reg                  I_DJNZ                 ;
reg                  I_CPL                  ;
reg                  I_CCF                  ;
reg                  I_SCF                  ;
reg                  I_RETN                 ;
reg                  I_BT                   ;
reg                  I_BC                   ;
reg                  I_BTR                  ;
reg                  I_RLD                  ;
reg                  I_RRD                  ;
reg                  I_INRC                 ;
reg                  SetDI                  ;
reg                  SetEI                  ;
reg [1:0]            IMode                  ;
reg                  Halt                   ;
reg                  NoRead                 ;
reg                  Write   ;

parameter            aNone   = 3'b111;
parameter            aBC     = 3'b000;
parameter            aDE     = 3'b001;
parameter            aXY     = 3'b010;
```

```
parameter             aIOA    = 3'b100;
parameter             aSP     = 3'b101;
parameter             aZI     = 3'b110;
//    constant aNone  : std_logic_vector[2:0] = 3'b000;
//    constant aXY    : std_logic_vector[2:0] = 3'b001;
//    constant aIOA   : std_logic_vector[2:0] = 3'b010;
//    constant aSP    : std_logic_vector[2:0] = 3'b011;
//    constant aBC    : std_logic_vector[2:0] = 3'b100;
//    constant aDE    : std_logic_vector[2:0] = 3'b101;
//    constant aZI    : std_logic_vector[2:0] = 3'b110;

function is_cc_true;
  input [7:0] F;
  input [2:0] cc;
  begin
    if (Mode == 3 )
      begin
        case (cc)
          3'b000  : is_cc_true = F[7] == 1'b0; // NZ
          3'b001  : is_cc_true = F[7] == 1'b1; // Z
          3'b010  : is_cc_true = F[4] == 1'b0; // NC
          3'b011  : is_cc_true = F[4] == 1'b1; // C
          3'b100  : is_cc_true = 0;
          3'b101  : is_cc_true = 0;
          3'b110  : is_cc_true = 0;
          3'b111  : is_cc_true = 0;
        endcase
      end
    else
      begin
        case (cc)
          3'b000  : is_cc_true = F[6] == 1'b0; // NZ
          3'b001  : is_cc_true = F[6] == 1'b1; // Z
          3'b010  : is_cc_true = F[0] == 1'b0; // NC
          3'b011  : is_cc_true = F[0] == 1'b1; // C
          3'b100  : is_cc_true = F[2] == 1'b0; // PO
          3'b101  : is_cc_true = F[2] == 1'b1; // PE
          3'b110  : is_cc_true = F[7] == 1'b0; // P
          3'b111  : is_cc_true = F[7] == 1'b1; // M
        endcase
      end
  end
endfunction // is_cc_true


reg [2:0] DDD;
reg [2:0] SSS;
reg [1:0] DPAIR;

always @ (/*AUTOSENSE*/F or IR or ISet or IntCycle or MCycle
          or NMICycle)
  begin
    DDD = IR[5:3];
    SSS = IR[2:0];
    DPAIR = IR[5:4];

    MCycles = 3'b001;
    if (MCycle[0] )
      begin
        TStates = 3'b100;
      end
    else
      begin
        TStates = 3'b011;
      end
    Prefix = 2'b00;
    Inc_PC = 1'b0;
    Inc_WZ = 1'b0;
    IncDec_16 = 4'b0000;
    Read_To_Acc = 1'b0;
    Read_To_Reg = 1'b0;
```

```
Set_BusB_To = 4'b0000;
Set_BusA_To = 4'b0000;
ALU_Op = { 1'b0, IR[5:3] };
Save_ALU = 1'b0;
PreserveC = 1'b0;
Arith16 = 1'b0;
IORQ = 1'b0;
Set_Addr_To = aNone;
Jump = 1'b0;
JumpE = 1'b0;
JumpXY = 1'b0;
Call = 1'b0;
RstP = 1'b0;
LDZ = 1'b0;
LDW = 1'b0;
LDSPHL = 1'b0;
Special_LD = 3'b000;
ExchangeDH = 1'b0;
ExchangeRp = 1'b0;
ExchangeAF = 1'b0;
ExchangeRS = 1'b0;
I_DJNZ = 1'b0;
I_CPL = 1'b0;
I_CCF = 1'b0;
I_SCF = 1'b0;
I_RETN = 1'b0;
I_BT = 1'b0;
I_BC = 1'b0;
I_BTR = 1'b0;
I_RLD = 1'b0;
I_RRD = 1'b0;
I_INRC = 1'b0;
SetDI = 1'b0;
SetEI = 1'b0;
IMode = 2'b10; //2'b11
Halt = 1'b0;
NoRead = 1'b0;
Write = 1'b0;

case (ISet)
  2'b00  :
    begin

      //------------------------------------------------------------------------------
      //
      //  Unprefixed instructions
      //
      //------------------------------------------------------------------------------

      casex (IR)
        // 8 BIT LOAD GROUP
        8'b01xxxxxx :
          begin
            if (IR[5:0] == 6'b110110)
              Halt = 1'b1;
            else if (IR[2:0] == 3'b110)
              begin
                // LD r,(HL)
                MCycles = 3'b010;
                if (MCycle[0])
                  Set_Addr_To = aXY;
                if (MCycle[1])
                  begin
                    Set_BusA_To[2:0] = DDD;
                    Read_To_Reg = 1'b1;
                  end
              end // if (IR[2:0] == 3'b110)
            else if (IR[5:3] == 3'b110)
              begin
                // LD (HL),r
                MCycles = 3'b010;
```

```
        if (MCycle[0])
          begin
            Set_Addr_To = aXY;
            Set_BusB_To[2:0] = SSS;
            Set_BusB_To[3] = 1'b0;
          end
        if (MCycle[1])
          Write = 1'b1;
      end // if (IR[5:3] == 3'b110)
    else
      begin
        Set_BusB_To[2:0] = SSS;
        ExchangeRp = 1'b1;
        Set_BusA_To[2:0] = DDD;
        Read_To_Reg = 1'b1;
      end // else: !if(IR[5:3] == 3'b110)
  end // case: 8'b01xxxxxx

8'b00xxx110 :
  begin
    if (IR[5:3] == 3'b110)
      begin
        // LD (HL),n
        MCycles = 3'b011;
        if (MCycle[1])
          begin
            Inc_PC = 1'b1;
            Set_Addr_To = aXY;
            Set_BusB_To[2:0] = SSS;
            Set_BusB_To[3] = 1'b0;
          end
        if (MCycle[2])
          Write = 1'b1;
      end // if (IR[5:3] == 3'b110)
    else
      begin
        // LD r,n
        MCycles = 3'b010;
        if (MCycle[1])
          begin
            Inc_PC = 1'b1;
            Set_BusA_To[2:0] = DDD;
            Read_To_Reg = 1'b1;
          end
      end
  end

8'b00001010  :
  begin
    // LD A,(BC)
    MCycles = 3'b010;
    if (MCycle[0])
      Set_Addr_To = aBC;
    if (MCycle[1])
      Read_To_Acc = 1'b1;
  end // case: 8'b00001010

8'b00011010  :
  begin
    // LD A,(DE)
    MCycles = 3'b010;
    if (MCycle[0])
      Set_Addr_To = aDE;
    if (MCycle[1])
      Read_To_Acc = 1'b1;
  end // case: 8'b00011010

8'b00111010  :
  begin
    if (Mode == 3 )
      begin
```

```
        // LDD A,(HL)
        MCycles = 3'b010;
        if (MCycle[0])
          Set_Addr_To = aXY;
        if (MCycle[1])
          begin
            Read_To_Acc = 1'b1;
            IncDec_16 = 4'b1110;
          end
      end
    else
      begin
        // LD A,(nn)
        MCycles = 3'b100;
        if (MCycle[1])
          begin
            Inc_PC = 1'b1;
            LDZ = 1'b1;
          end
        if (MCycle[2])
          begin
            Set_Addr_To = aZI;
            Inc_PC = 1'b1;
          end
        if (MCycle[3])
          begin
            Read_To_Acc = 1'b1;
          end
      end // else: !if(Mode == 3 )
  end // case: 8'b00111010

8'b00000010  :
  begin
    // LD (BC),A
    MCycles = 3'b010;
    if (MCycle[0])
      begin
        Set_Addr_To = aBC;
        Set_BusB_To = 4'b0111;
      end
    if (MCycle[1])
      begin
        Write = 1'b1;
      end
  end // case: 8'b00000010

8'b00010010  :
  begin
    // LD (DE),A
    MCycles = 3'b010;
    case (1'b1) // MCycle
      MCycle[0] :
        begin
          Set_Addr_To = aDE;
          Set_BusB_To = 4'b0111;
        end
      MCycle[1] :
        Write = 1'b1;
      default :;
    endcase // case(MCycle)
  end // case: 8'b00010010

8'b00110010  :
  begin
    if (Mode == 3 )
      begin
        // LDD (HL),A
        MCycles = 3'b010;
        case (1'b1) // MCycle
          MCycle[0] :
            begin
```

```
              Set_Addr_To = aXY;
              Set_BusB_To = 4'b0111;
            end
          MCycle[1] :
            begin
              Write = 1'b1;
              IncDec_16 = 4'b1110;
            end
          default :;
        endcase // case(MCycle)

      end
    else
      begin
        // LD (nn),A
        MCycles = 3'b100;
        case (1'b1) // MCycle
          MCycle[1] :
            begin
              Inc_PC = 1'b1;
              LDZ = 1'b1;
            end
          MCycle[2] :
            begin
              Set_Addr_To = aZI;
              Inc_PC = 1'b1;
              Set_BusB_To = 4'b0111;
            end
          MCycle[3] :
            begin
              Write = 1'b1;
            end
          default :;
        endcase
      end // else: !if(Mode == 3 )
  end // case: 8'b00110010


// 16 BIT LOAD GROUP
8'b00000001,8'b00010001,8'b00100001,8'b00110001  :
  begin
    // LD dd,nn
    MCycles = 3'b011;
    case (1'b1) // MCycle
      MCycle[1] :
        begin
          Inc_PC = 1'b1;
          Read_To_Reg = 1'b1;
          if (DPAIR == 2'b11 )
            begin
              Set_BusA_To[3:0] = 4'b1000;
            end
          else
            begin
              Set_BusA_To[2:1] = DPAIR;
              Set_BusA_To[0] = 1'b1;
            end
        end // case: 2

      MCycle[2] :
        begin
          Inc_PC = 1'b1;
          Read_To_Reg = 1'b1;
          if (DPAIR == 2'b11 )
            begin
              Set_BusA_To[3:0] = 4'b1001;
            end
          else
            begin
              Set_BusA_To[2:1] = DPAIR;
              Set_BusA_To[0] = 1'b0;
```

```
            end
          end // case: 3

        default :;
      endcase // case(MCycle)
    end // case: 8'b00000001,8'b00010001,8'b00100001,8'b00110001

  8'b00101010  :
    begin
      if (Mode == 3 )
        begin
          // LDI A,(HL)
          MCycles = 3'b010;
          case (1'b1) // MCycle
            MCycle[0] :
              Set_Addr_To = aXY;
            MCycle[1] :
              begin
                Read_To_Acc = 1'b1;
                IncDec_16 = 4'b0110;
              end

            default :;
          endcase
        end
      else
        begin
          // LD HL,(nn)
          MCycles = 3'b101;
          case (1'b1) // MCycle
            MCycle[1] :
              begin
                Inc_PC = 1'b1;
                LDZ = 1'b1;
              end
            MCycle[2] :
              begin
                Set_Addr_To = aZI;
                Inc_PC = 1'b1;
                LDW = 1'b1;
              end
            MCycle[3] :
              begin
                Set_BusA_To[2:0] = 3'b101; // L
                Read_To_Reg = 1'b1;
                Inc_WZ = 1'b1;
                Set_Addr_To = aZI;
              end
            MCycle[4] :
              begin
                Set_BusA_To[2:0] = 3'b100; // H
                Read_To_Reg = 1'b1;
              end
            default :;
          endcase
        end // else: !if(Mode == 3 )
    end // case: 8'b00101010

  8'b00100010  :
    begin
      if (Mode == 3 )
        begin
          // LDI (HL),A
          MCycles = 3'b010;
          case (1'b1) // MCycle
            MCycle[0] :
              begin
                Set_Addr_To = aXY;
                Set_BusB_To = 4'b0111;
              end
            MCycle[1] :
```

```
            begin
              Write = 1'b1;
              IncDec_16 = 4'b0110;
            end
          default :;
        endcase
      end
    else
      begin
        // LD (nn),HL
        MCycles = 3'b101;
        case (1'b1) // MCycle
          MCycle[1] :
            begin
              Inc_PC = 1'b1;
              LDZ = 1'b1;
            end

          MCycle[2] :
            begin
              Set_Addr_To = aZI;
              Inc_PC = 1'b1;
              LDW = 1'b1;
              Set_BusB_To = 4'b0101; // L
            end

          MCycle[3] :
            begin
              Inc_WZ = 1'b1;
              Set_Addr_To = aZI;
              Write = 1'b1;
              Set_BusB_To = 4'b0100; // H
            end
          MCycle[4] :
            Write = 1'b1;
          default :;
        endcase
      end // else: !if(Mode == 3 )
  end // case: 8'b00100010

8'b11111001  :
  begin
    // LD SP,HL
    TStates = 3'b110;
    LDSPHL = 1'b1;
  end

8'b11xx0101 :
  begin
    // PUSH qq
    MCycles = 3'b011;
    case (1'b1) // MCycle
      MCycle[0] :
        begin
          TStates = 3'b101;
          IncDec_16 = 4'b1111;
          Set_Addr_To = aSP;
          if (DPAIR == 2'b11 )
            begin
              Set_BusB_To = 4'b0111;
            end
          else
            begin
              Set_BusB_To[2:1] = DPAIR;
              Set_BusB_To[0] = 1'b0;
              Set_BusB_To[3] = 1'b0;
            end
        end // case: 1

      MCycle[1] :
        begin
```

```
          IncDec_16 = 4'b1111;
          Set_Addr_To = aSP;
          if (DPAIR == 2'b11 )
            begin
              Set_BusB_To = 4'b1011;
            end
          else
            begin
              Set_BusB_To[2:1] = DPAIR;
              Set_BusB_To[0] = 1'b1;
              Set_BusB_To[3] = 1'b0;
            end
          Write = 1'b1;
        end // case: 2

      MCycle[2] :
        Write = 1'b1;
      default :;
    endcase // case(MCycle)
  end // case: 8'b11000101,8'b11010101,8'b11100101,8'b11110101

8'b11xx0001 :
  begin
    // POP qq
    MCycles = 3'b011;
    case (1'b1) // MCycle
      MCycle[0] :
        Set_Addr_To = aSP;
      MCycle[1] :
        begin
          IncDec_16 = 4'b0111;
          Set_Addr_To = aSP;
          Read_To_Reg = 1'b1;
          if (DPAIR == 2'b11 )
            begin
              Set_BusA_To[3:0] = 4'b1011;
            end
          else
            begin
              Set_BusA_To[2:1] = DPAIR;
              Set_BusA_To[0] = 1'b1;
            end
        end // case: 2

      MCycle[2] :
        begin
          IncDec_16 = 4'b0111;
          Read_To_Reg = 1'b1;
          if (DPAIR == 2'b11 )
            begin
              Set_BusA_To[3:0] = 4'b0111;
            end
          else
            begin
              Set_BusA_To[2:1] = DPAIR;
              Set_BusA_To[0] = 1'b0;
            end
        end // case: 3

      default :;
    endcase // case(MCycle)
  end // case: 8'b11000001,8'b11010001,8'b11100001,8'b11110001


// EXCHANGE, BLOCK TRANSFER AND SEARCH GROUP
8'b11101011  :
  begin
    if (Mode != 3 )
      begin
        // EX DE,HL
        ExchangeDH = 1'b1;
```

```
      end
  end

8'b00001000  :
  begin
    if (Mode == 3 )
      begin
        // LD (nn),SP
        MCycles = 3'b101;
        case (1'b1) // MCycle
          MCycle[1] :
            begin
              Inc_PC = 1'b1;
              LDZ = 1'b1;
            end

          MCycle[2] :
            begin
              Set_Addr_To = aZI;
              Inc_PC = 1'b1;
              LDW = 1'b1;
              Set_BusB_To = 4'b1000;
            end

          MCycle[3] :
            begin
              Inc_WZ = 1'b1;
              Set_Addr_To = aZI;
              Write = 1'b1;
              Set_BusB_To = 4'b1001;
            end

          MCycle[4] :
            Write = 1'b1;
          default :;
        endcase
      end
    else if (Mode < 2 )
      begin
        // EX AF,AF'
        ExchangeAF = 1'b1;
      end
  end // case: 8'b00001000

8'b11011001  :
  begin
    if (Mode == 3 )
      begin
        // RETI
        MCycles = 3'b011;
        case (1'b1) // MCycle
          MCycle[0] :
            Set_Addr_To = aSP;
          MCycle[1] :
            begin
              IncDec_16 = 4'b0111;
              Set_Addr_To = aSP;
              LDZ = 1'b1;
            end

          MCycle[2] :
            begin
              Jump = 1'b1;
              IncDec_16 = 4'b0111;
              //I_RETN = 1'b1; // GameBoy -- Remove to reenable FF1
              SetEI = 1'b1;
            end
          default :;
        endcase
      end
    else if (Mode < 2 )
```

```
      begin
        // EXX
        ExchangeRS = 1'b1;
      end
  end // case: 8'b11011001

8'b11100011  :
  begin
    if (Mode != 3 )
      begin
        // EX (SP),HL
        MCycles = 3'b101;
        case (1'b1) // MCycle
          MCycle[0] :
            Set_Addr_To = aSP;
          MCycle[1] :
            begin
              Read_To_Reg = 1'b1;
              Set_BusA_To = 4'b0101;
              Set_BusB_To = 4'b0101;
              Set_Addr_To = aSP;
            end
          MCycle[2] :
            begin
              IncDec_16 = 4'b0111;
              Set_Addr_To = aSP;
              TStates = 3'b100;
              Write = 1'b1;
            end
          MCycle[3] :
            begin
              Read_To_Reg = 1'b1;
              Set_BusA_To = 4'b0100;
              Set_BusB_To = 4'b0100;
              Set_Addr_To = aSP;
            end
          MCycle[4] :
            begin
              IncDec_16 = 4'b1111;
              TStates = 3'b101;
              Write = 1'b1;
            end

          default :;
        endcase
      end // if (Mode != 3 )
  end // case: 8'b11100011


// 8 BIT ARITHMETIC AND LOGICAL GROUP
8'b10xxxxxx :
  begin
    if (IR[2:0] == 3'b110)
      begin
        // ADD A,(HL)
        // ADC A,(HL)
        // SUB A,(HL)
        // SBC A,(HL)
        // AND A,(HL)
        // OR A,(HL)
        // XOR A,(HL)
        // CP A,(HL)
        MCycles = 3'b010;
        case (1'b1) // MCycle
          MCycle[0] :
            Set_Addr_To = aXY;
          MCycle[1] :
            begin
              Read_To_Reg = 1'b1;
              Save_ALU = 1'b1;
              Set_BusB_To[2:0] = SSS;
```

```
                        Set_BusA_To[2:0] = 3'b111;
                      end

                    default :;
                  endcase // case(MCycle)
                end // if (IR[2:0] == 3'b110)
              else
                begin
                  // ADD A,r
                  // ADC A,r
                  // SUB A,r
                  // SBC A,r
                  // AND A,r
                  // OR A,r
                  // XOR A,r
                  // CP A,r
                  Set_BusB_To[2:0] = SSS;
                  Set_BusA_To[2:0] = 3'b111;
                  Read_To_Reg = 1'b1;
                  Save_ALU = 1'b1;
                end // else: !if(IR[2:0] == 3'b110)
            end // case:
8'b10000000,8'b10000001,8'b10000010,8'b10000011,8'b10000100,8'b10000101,8'b10000111,...

          8'b11xxx110 :
            begin
              // ADD A,n
              // ADC A,n
              // SUB A,n
              // SBC A,n
              // AND A,n
              // OR A,n
              // XOR A,n
              // CP A,n
              MCycles = 3'b010;
              if (MCycle[1] )
                begin
                  Inc_PC = 1'b1;
                  Read_To_Reg = 1'b1;
                  Save_ALU = 1'b1;
                  Set_BusB_To[2:0] = SSS;
                  Set_BusA_To[2:0] = 3'b111;
                end
            end

          8'b00xxx100 :
            begin
              if (IR[5:3] == 3'b110)
                begin
                  // INC (HL)
                  MCycles = 3'b011;
                  case (1'b1) // MCycle
                    MCycle[0] :
                      Set_Addr_To = aXY;
                    MCycle[1] :
                      begin
                        TStates = 3'b100;
                        Set_Addr_To = aXY;
                        Read_To_Reg = 1'b1;
                        Save_ALU = 1'b1;
                        PreserveC = 1'b1;
                        ALU_Op = 4'b0000;
                        Set_BusB_To = 4'b1010;
                        Set_BusA_To[2:0] = DDD;
                      end // case: 2

                    MCycle[2] :
                      Write = 1'b1;
                    default :;
                  endcase // case(MCycle)
                end // case: 8'b00110100
```

```
    else
      begin
        // INC r
        Set_BusB_To = 4'b1010;
        Set_BusA_To[2:0] = DDD;
        Read_To_Reg = 1'b1;
        Save_ALU = 1'b1;
        PreserveC = 1'b1;
        ALU_Op = 4'b0000;
      end
  end

8'b00xxx101 :
  begin
    if (IR[5:3] == 3'b110)
      begin
        // DEC (HL)
        MCycles = 3'b011;
        case (1'b1) // MCycle
          MCycle[0] :
            Set_Addr_To = aXY;
          MCycle[1] :
            begin
              TStates = 3'b100;
              Set_Addr_To = aXY;
              ALU_Op = 4'b0010;
              Read_To_Reg = 1'b1;
              Save_ALU = 1'b1;
              PreserveC = 1'b1;
              Set_BusB_To = 4'b1010;
              Set_BusA_To[2:0] = DDD;
            end // case: 2

          MCycle[2] :
            Write = 1'b1;
          default :;
        endcase // case(MCycle)
      end
    else
      begin
        // DEC r
        Set_BusB_To = 4'b1010;
        Set_BusA_To[2:0] = DDD;
        Read_To_Reg = 1'b1;
        Save_ALU = 1'b1;
        PreserveC = 1'b1;
        ALU_Op = 4'b0010;
      end
  end

// GENERAL PURPOSE ARITHMETIC AND CPU CONTROL GROUPS
8'b00100111  :
  begin
    // DAA
    Set_BusA_To[2:0] = 3'b111;
    Read_To_Reg = 1'b1;
    ALU_Op = 4'b1100;
    Save_ALU = 1'b1;
  end

8'b00101111  :
  // CPL
  I_CPL = 1'b1;

8'b00111111  :
  // CCF
  I_CCF = 1'b1;

8'b00110111  :
  // SCF
  I_SCF = 1'b1;
```

```
8'b00000000  :
  begin
    if (NMICycle == 1'b1 )
      begin
        // NMI
        MCycles = 3'b011;
        case (1'b1) // MCycle
          MCycle[0] :
            begin
              TStates = 3'b101;
              IncDec_16 = 4'b1111;
              Set_Addr_To = aSP;
              Set_BusB_To = 4'b1101;
            end

          MCycle[1] :
            begin
              TStates = 3'b100;
              Write = 1'b1;
              IncDec_16 = 4'b1111;
              Set_Addr_To = aSP;
              Set_BusB_To = 4'b1100;
            end

          MCycle[2] :
            begin
              TStates = 3'b100;
              Write = 1'b1;
            end

          default :;
        endcase // case(MCycle)

      end
    else if (IntCycle == 1'b1 )
      begin
        // INT (IM 2)
        MCycles = 3'b101;
        case (1'b1) // MCycle
          MCycle[0] :
            begin
              LDZ = 1'b1;
              TStates = 3'b101;
              IncDec_16 = 4'b1111;
              Set_Addr_To = aSP;
              Set_BusB_To = 4'b1101;
            end

          MCycle[1] :
            begin
              TStates = 3'b100;
              Write = 1'b1;
              IncDec_16 = 4'b1111;
              Set_Addr_To = aSP;
              Set_BusB_To = 4'b1100;
            end

          MCycle[2] :
            begin
              TStates = 3'b100;
              Write = 1'b1;
            end

          MCycle[3] :
            begin
              Inc_PC = 1'b1;
              LDZ = 1'b1;
            end

          MCycle[4] :
```

```
            Jump = 1'b1;
          default :;
        endcase
      end
  end // case: 8'b00000000

8'b11110011  :
  // DI
  SetDI = 1'b1;

8'b11111011  :
  // EI
  SetEI = 1'b1;

// 16 BIT ARITHMETIC GROUP
8'b00xx1001  :
  begin
    // ADD HL,ss
    MCycles = 3'b011;
    case (1'b1) // MCycle
      MCycle[1] :
        begin
          NoRead = 1'b1;
          ALU_Op = 4'b0000;
          Read_To_Reg = 1'b1;
          Save_ALU = 1'b1;
          Set_BusA_To[2:0] = 3'b101;
          case (IR[5:4])
            0,1,2  :
              begin
                Set_BusB_To[2:1] = IR[5:4];
                Set_BusB_To[0] = 1'b1;
              end

            default :
              Set_BusB_To = 4'b1000;
          endcase // case(IR[5:4])

          TStates = 3'b100;
          Arith16 = 1'b1;
        end // case: 2

      MCycle[2] :
        begin
          NoRead = 1'b1;
          Read_To_Reg = 1'b1;
          Save_ALU = 1'b1;
          ALU_Op = 4'b0001;
          Set_BusA_To[2:0] = 3'b100;
          case (IR[5:4])
            0,1,2  :
              begin
                Set_BusB_To[2:1] = IR[5:4];
              end

            default :
              Set_BusB_To = 4'b1001;
          endcase
          Arith16 = 1'b1;
        end // case: 3

      default :;
    endcase // case(MCycle)
  end // case: 8'b00001001,8'b00011001,8'b00101001,8'b00111001

8'b00xx0011 :
  begin
    // INC ss
    TStates = 3'b110;
    IncDec_16[3:2] = 2'b01;
    IncDec_16[1:0] = DPAIR;
```

```
  end

8'b00xx1011 :
  begin
    // DEC ss
    TStates = 3'b110;
    IncDec_16[3:2] = 2'b11;
    IncDec_16[1:0] = DPAIR;
  end

// ROTATE AND SHIFT GROUP
8'b00000111,
    // RLCA
    8'b00010111,
    // RLA
    8'b00001111,
    // RRCA
    8'b00011111 :
      // RRA
      begin
        Set_BusA_To[2:0] = 3'b111;
        ALU_Op = 4'b1000;
        Read_To_Reg = 1'b1;
        Save_ALU = 1'b1;
      end // case: 8'b00000111,...


// JUMP GROUP
8'b11000011  :
  begin
    // JP nn
    MCycles = 3'b011;
    if (MCycle[1])
      begin
        Inc_PC = 1'b1;
        LDZ = 1'b1;
      end

    if (MCycle[2])
      begin
        Inc_PC = 1'b1;
        Jump = 1'b1;
      end

  end // case: 8'b11000011

8'b11xxx010  :
  begin
    if (IR[5] == 1'b1 && Mode == 3 )
      begin
        case (IR[4:3])
          2'b00  :
            begin
              // LD ($FF00+C),A
              MCycles = 3'b010;
              case (1'b1) // MCycle
                MCycle[0] :
                  begin
                    IORQ = 1'b1; // GameBoy -- Added (was in wrong mcycle)
                    Set_Addr_To = aBC;
                    Set_BusB_To   = 4'b0111;
                  end
                MCycle[1] :
                  begin
                    Write = 1'b1;
                    //IORQ = 1'b1; // GameBoy -- Removed (wrong mcycle)
                  end

                default :;
              endcase // case(MCycle)
            end // case: 2'b00
```

```
      2'b01  :
        begin
          // LD (nn),A
          MCycles = 3'b100;
          case (1'b1) // MCycle
            MCycle[1] :
              begin
                Inc_PC = 1'b1;
                LDZ = 1'b1;
              end

            MCycle[2] :
              begin
                Set_Addr_To = aZI;
                Inc_PC = 1'b1;
                Set_BusB_To = 4'b0111;
              end

            MCycle[3] :
              Write = 1'b1;
            default :;
          endcase // case(MCycle)
        end // case: default :...

      2'b10  :
        begin
          // LD A,($FF00+C)
          MCycles = 3'b010;
          case (1'b1) // MCycle
            MCycle[0] :
              Set_Addr_To = aBC;
            MCycle[1] :
              begin
                Read_To_Acc = 1'b1;
                IORQ = 1'b1;
              end
            default :;
          endcase // case(MCycle)
        end // case: 2'b10

      2'b11  :
        begin
          // LD A,(nn)
          MCycles = 3'b100;
          case (1'b1) // MCycle
            MCycle[1] :
              begin
                Inc_PC = 1'b1;
                LDZ = 1'b1;
              end
            MCycle[2] :
              begin
                Set_Addr_To = aZI;
                Inc_PC = 1'b1;
              end
            MCycle[3] :
              Read_To_Acc = 1'b1;
            default :;
          endcase // case(MCycle)
        end
    endcase
  end
else
  begin
    // JP cc,nn
    MCycles = 3'b011;
    case (1'b1) // MCycle
      MCycle[1] :
        begin
          Inc_PC = 1'b1;
```

```
                        LDZ = 1'b1;
                      end
                    MCycle[2] :
                      begin
                        Inc_PC = 1'b1;
                        if (is_cc_true(F, IR[5:3]) )
                          begin
                            Jump = 1'b1;
                          end
                      end

                    default :;
                  endcase
              end // else: !if(DPAIR == 2'b11 )
          end // case:
8'b11000010,8'b11001010,8'b11010010,8'b11011010,8'b11100010,8'b11101010,8'b11110010,8'b11111010

        8'b00011000  :
          begin
            if (Mode != 2 )
              begin
                // JR e
                MCycles = 3'b011;
                case (1'b1) // MCycle
                  MCycle[1] :
                    Inc_PC = 1'b1;
                  MCycle[2] :
                    begin
                      NoRead = 1'b1;
                      JumpE = 1'b1;
                      TStates = 3'b101;
                    end
                  default :;
                endcase
              end // if (Mode != 2 )
          end // case: 8'b00011000

        // Conditional relative jumps (JR [C/NC/Z/NZ], e)
        8'b001xx000  :
          begin
            if (Mode != 2 )
              begin
                MCycles = 3'd3;
                case (1'b1) // MCycle
                  MCycle[1] :
                    begin
                      Inc_PC = 1'b1;

                      case (IR[4:3])
                        0 : MCycles = (F[Flag_Z]) ? 3'd2 : 3'd3;
                        1 : MCycles = (!F[Flag_Z]) ? 3'd2 : 3'd3;
                        2 : MCycles = (F[Flag_C]) ? 3'd2 : 3'd3;
                        3 : MCycles = (!F[Flag_C]) ? 3'd2 : 3'd3;
                      endcase
                    end

                  MCycle[2] :
                    begin
                      NoRead = 1'b1;
                      JumpE = 1'b1;
                      TStates = 3'd5;
                    end
                  default :;
                endcase
              end // if (Mode != 2 )
          end // case: 8'b00111000

        8'b11101001  :
          // JP (HL)
          JumpXY = 1'b1;
```

```
8'b00010000  :
  begin
    if (Mode == 3 )
      begin
        I_DJNZ = 1'b1;
      end
    else if (Mode < 2 )
      begin
        // DJNZ,e
        MCycles = 3'b011;
        case (1'b1) // MCycle
          MCycle[0] :
            begin
              TStates = 3'b101;
              I_DJNZ = 1'b1;
              Set_BusB_To = 4'b1010;
              Set_BusA_To[2:0] = 3'b000;
              Read_To_Reg = 1'b1;
              Save_ALU = 1'b1;
              ALU_Op = 4'b0010;
            end
          MCycle[1] :
            begin
              I_DJNZ = 1'b1;
              Inc_PC = 1'b1;
            end
          MCycle[2] :
            begin
              NoRead = 1'b1;
              JumpE = 1'b1;
              TStates = 3'b101;
            end
          default :;
        endcase
      end // if (Mode < 2 )
  end // case: 8'b00010000


// CALL AND RETURN GROUP
8'b11001101  :
  begin
    // CALL nn
    MCycles = 3'b101;
    case (1'b1) // MCycle
      MCycle[1] :
        begin
          Inc_PC = 1'b1;
          LDZ = 1'b1;
        end
      MCycle[2] :
        begin
          IncDec_16 = 4'b1111;
          Inc_PC = 1'b1;
          TStates = 3'b100;
          Set_Addr_To = aSP;
          LDW = 1'b1;
          Set_BusB_To = 4'b1101;
        end
      MCycle[3] :
        begin
          Write = 1'b1;
          IncDec_16 = 4'b1111;
          Set_Addr_To = aSP;
          Set_BusB_To = 4'b1100;
        end
      MCycle[4] :
        begin
          Write = 1'b1;
          Call = 1'b1;
        end
      default :;
```

```
                endcase // case(MCycle)
              end // case: 8'b11001101

            8'b11xxx100  :
              begin
                if (IR[5] == 1'b0 || Mode != 3 )
                  begin
                    // CALL cc,nn
                    MCycles = 3'b101;
                    case (1'b1) // MCycle
                      MCycle[1] :
                        begin
                          Inc_PC = 1'b1;
                          LDZ = 1'b1;
                        end
                      MCycle[2] :
                        begin
                          Inc_PC = 1'b1;
                          LDW = 1'b1;
                          if (is_cc_true(F, IR[5:3]) )
                            begin
                              IncDec_16 = 4'b1111;
                              Set_Addr_To = aSP;
                              TStates = 3'b100;
                              Set_BusB_To = 4'b1101;
                            end
                          else
                            begin
                              MCycles = 3'b011;
                            end // else: !if(is_cc_true(F, IR[5:3]) )
                        end // case: 3

                      MCycle[3] :
                        begin
                          Write = 1'b1;
                          IncDec_16 = 4'b1111;
                          Set_Addr_To = aSP;
                          Set_BusB_To = 4'b1100;
                        end

                      MCycle[4] :
                        begin
                          Write = 1'b1;
                          Call = 1'b1;
                        end

                      default :;
                    endcase
                  end // if (IR[5] == 1'b0 || Mode != 3 )
              end // case:
8'b11000100,8'b11001100,8'b11010100,8'b11011100,8'b11100100,8'b11101100,8'b11110100,8'b11111100

            8'b11001001  :
              begin
                // RET
                MCycles = 3'b011;
                case (1'b1) // MCycle
                  MCycle[0] :
                    begin
                      TStates = 3'b101;
                      Set_Addr_To = aSP;
                    end

                  MCycle[1] :
                    begin
                      IncDec_16 = 4'b0111;
                      Set_Addr_To = aSP;
                      LDZ = 1'b1;
                    end

                  MCycle[2] :
```

```
                    begin
                      Jump = 1'b1;
                      IncDec_16 = 4'b0111;
                    end

                  default :;
                endcase // case(MCycle)
              end // case: 8'b11001001


8'b11000000,8'b11001000,8'b11010000,8'b11011000,8'b11100000,8'b11101000,8'b11110000,8'b11111000
:
              begin
                if (IR[5] == 1'b1 && Mode == 3 )
                  begin
                    case (IR[4:3])
                      2'b00  :
                        begin
                          // LD ($FF00+nn),A
                          MCycles = 3'b011;
                          case (1'b1) // MCycle
                            MCycle[1] :
                              begin
                                Inc_PC = 1'b1;
                                Set_Addr_To = aIOA;
                                Set_BusB_To   = 4'b0111;
                              end

                            MCycle[2] :
                              Write = 1'b1;
                            default :;
                          endcase // case(MCycle)
                        end // case: 2'b00

                      2'b01  :
                        begin
                          // ADD SP,n
                          MCycles = 3'b011;
                          case (1'b1) // MCycle
                            MCycle[1] :
                              begin
                                ALU_Op = 4'b0000;
                                Inc_PC = 1'b1;
                                Read_To_Reg = 1'b1;
                                Save_ALU = 1'b1;
                                Set_BusA_To = 4'b1000;
                                Set_BusB_To = 4'b0110;
                              end

                            MCycle[2] :
                              begin
                                NoRead = 1'b1;
                                Read_To_Reg = 1'b1;
                                Save_ALU = 1'b1;
                                ALU_Op = 4'b0001;
                                Set_BusA_To = 4'b1001;
                                Set_BusB_To = 4'b1110;        // Incorrect unsigned
!!!!!!!!!!!!!!!!!!!!
                              end

                            default :;
                          endcase // case(MCycle)
                        end // case: 2'b01

                      2'b10  :
                        begin
                          // LD A,($FF00+nn)
                          MCycles = 3'b011;
                          case (1'b1) // MCycle
                            MCycle[1] :
                              begin
```

```
                Inc_PC = 1'b1;
                Set_Addr_To = aIOA;
              end

            MCycle[2] :
              Read_To_Acc = 1'b1;
            default :;
          endcase // case(MCycle)
        end // case: 2'b10

      2'b11  :
        begin
          // LD HL,SP+n       -- Not correct !!!!!!!!!!!!!!!!!!
          MCycles = 3'b101;
          case (1'b1) // MCycle
            MCycle[1] :
              begin
                Inc_PC = 1'b1;
                LDZ = 1'b1;
              end

            MCycle[2] :
              begin
                Set_Addr_To = aZI;
                Inc_PC = 1'b1;
                LDW = 1'b1;
              end

            MCycle[3] :
              begin
                Set_BusA_To[2:0] = 3'b101; // L
                Read_To_Reg = 1'b1;
                Inc_WZ = 1'b1;
                Set_Addr_To = aZI;
              end

            MCycle[4] :
              begin
                Set_BusA_To[2:0] = 3'b100; // H
                Read_To_Reg = 1'b1;
              end

            default :;
          endcase // case(MCycle)
        end // case: 2'b11

    endcase // case(IR[4:3])

  end
else
  begin
    // RET cc
    MCycles = 3'b011;
    case (1'b1) // MCycle
      MCycle[0] :
        begin
          if (is_cc_true(F, IR[5:3]) )
            begin
              Set_Addr_To = aSP;
            end
          else
            begin
              MCycles = 3'b001;
            end
          TStates = 3'b101;
        end // case: 1

      MCycle[1] :
        begin
          IncDec_16 = 4'b0111;
          Set_Addr_To = aSP;
```

```
                        LDZ = 1'b1;
                      end
                    MCycle[2] :
                      begin
                        Jump = 1'b1;
                        IncDec_16 = 4'b0111;
                      end
                    default :;
                  endcase
                end // else: !if(IR[5] == 1'b1 && Mode == 3 )
              end // case:
8'b11000000,8'b11001000,8'b11010000,8'b11011000,8'b11100000,8'b11101000,8'b11110000,8'b11111000


8'b11000111,8'b11001111,8'b11010111,8'b11011111,8'b11100111,8'b11101111,8'b11110111,8'b11111111
:
              begin
                // RST p
                MCycles = 3'b011;
                case (1'b1) // MCycle
                  MCycle[0] :
                    begin
                      TStates = 3'b101;
                      IncDec_16 = 4'b1111;
                      Set_Addr_To = aSP;
                      Set_BusB_To = 4'b1101;
                    end

                  MCycle[1] :
                    begin
                      Write = 1'b1;
                      IncDec_16 = 4'b1111;
                      Set_Addr_To = aSP;
                      Set_BusB_To = 4'b1100;
                    end

                  MCycle[2] :
                    begin
                      Write = 1'b1;
                      RstP = 1'b1;
                    end

                  default :;
                endcase // case(MCycle)
              end // case:
8'b11000111,8'b11001111,8'b11010111,8'b11011111,8'b11100111,8'b11101111,8'b11110111,8'b11111111

            // INPUT AND OUTPUT GROUP
            8'b11011011  :
              begin
                if (Mode != 3 )
                  begin
                    // IN A,(n)
                    MCycles = 3'b011;
                    case (1'b1) // MCycle
                      MCycle[1] :
                        begin
                          Inc_PC = 1'b1;
                          Set_Addr_To = aIOA;
                        end

                      MCycle[2] :
                        begin
                          Read_To_Acc = 1'b1;
                          IORQ = 1'b1;
                        end

                      default :;
                    endcase
                  end // if (Mode != 3 )
              end // case: 8'b11011011
```

```
      8'b11010011  :
        begin
          if (Mode != 3 )
            begin
              // OUT (n),A
              MCycles = 3'b011;
              case (1'b1) // MCycle
                MCycle[1] :
                  begin
                    Inc_PC = 1'b1;
                    Set_Addr_To = aIOA;
                    Set_BusB_To = 4'b0111;
                  end

                MCycle[2] :
                  begin
                    Write = 1'b1;
                    IORQ = 1'b1;
                  end

                default :;
              endcase
            end // if (Mode != 3 )
        end // case: 8'b11010011


      //-------------------------------------------------------------------------
      //-------------------------------------------------------------------------
      // MULTIBYTE INSTRUCTIONS
      //-------------------------------------------------------------------------
      //-------------------------------------------------------------------------

      8'b11001011  :
        begin
          if (Mode != 2 )
            begin
              Prefix = 2'b01;
            end
        end

      8'b11101101  :
        begin
          if (Mode < 2 )
            begin
              Prefix = 2'b10;
            end
        end

      8'b11011101,8'b11111101  :
        begin
          if (Mode < 2 )
            begin
              Prefix = 2'b11;
            end
        end

    endcase // case(IR)
  end // case: 2'b00


2'b01  :
  begin


    //-------------------------------------------------------------------------
    //
    //  CB prefixed instructions
    //
    //-------------------------------------------------------------------------
```

```
            Set_BusA_To[2:0] = IR[2:0];
            Set_BusB_To[2:0] = IR[2:0];

            casex (IR)

8'b00000000,8'b00000001,8'b00000010,8'b00000011,8'b00000100,8'b00000101,8'b00000111,

8'b00010000,8'b00010001,8'b00010010,8'b00010011,8'b00010100,8'b00010101,8'b00010111,

8'b00001000,8'b00001001,8'b00001010,8'b00001011,8'b00001100,8'b00001101,8'b00001111,

8'b00011000,8'b00011001,8'b00011010,8'b00011011,8'b00011100,8'b00011101,8'b00011111,

8'b00100000,8'b00100001,8'b00100010,8'b00100011,8'b00100100,8'b00100101,8'b00100111,

8'b00101000,8'b00101001,8'b00101010,8'b00101011,8'b00101100,8'b00101101,8'b00101111,

8'b00110000,8'b00110001,8'b00110010,8'b00110011,8'b00110100,8'b00110101,8'b00110111,
              8'b00111000,8'b00111001,8'b00111010,8'b00111011,8'b00111100,8'b00111101,8'b00111111
:
                begin
                  // RLC r
                  // RL r
                  // RRC r
                  // RR r
                  // SLA r
                  // SRA r
                  // SRL r
                  // SLL r (Undocumented) / SWAP r
                  if (MCycle[0] ) begin
                    ALU_Op = 4'b1000;
                    Read_To_Reg = 1'b1;
                    Save_ALU = 1'b1;
                  end
                end // case:
8'b00000000,8'b00000001,8'b00000010,8'b00000011,8'b00000100,8'b00000101,8'b00000111,...

              8'b00xxx110  :
                begin
                  // RLC (HL)
                  // RL (HL)
                  // RRC (HL)
                  // RR (HL)
                  // SRA (HL)
                  // SRL (HL)
                  // SLA (HL)
                  // SLL (HL) (Undocumented) / SWAP (HL)
                  MCycles = 3'b011;
                  case (1'b1) // MCycle
                    MCycle[0], MCycle[6] :
                      Set_Addr_To = aXY;
                    MCycle[1] :
                      begin
                        ALU_Op = 4'b1000;
                        Read_To_Reg = 1'b1;
                        Save_ALU = 1'b1;
                        Set_Addr_To = aXY;
                        TStates = 3'b100;
                      end

                    MCycle[2] :
                      Write = 1'b1;
                    default :;
                  endcase // case(MCycle)
                end // case:
8'b00000110,8'b00010110,8'b00001110,8'b00011110,8'b00101110,8'b00111110,8'b00100110,8'b00110110


8'b01000000,8'b01000001,8'b01000010,8'b01000011,8'b01000100,8'b01000101,8'b01000111,

8'b01001000,8'b01001001,8'b01001010,8'b01001011,8'b01001100,8'b01001101,8'b01001111,
```

```
8'b01010000,8'b01010001,8'b01010010,8'b01010011,8'b01010100,8'b01010101,8'b01010111,

8'b01011000,8'b01011001,8'b01011010,8'b01011011,8'b01011100,8'b01011101,8'b01011111,

8'b01100000,8'b01100001,8'b01100010,8'b01100011,8'b01100100,8'b01100101,8'b01100111,

8'b01101000,8'b01101001,8'b01101010,8'b01101011,8'b01101100,8'b01101101,8'b01101111,

8'b01110000,8'b01110001,8'b01110010,8'b01110011,8'b01110100,8'b01110101,8'b01110111,

8'b01111000,8'b01111001,8'b01111010,8'b01111011,8'b01111100,8'b01111101,8'b01111111 :
                    begin
                      // BIT b,r
                      if (MCycle[0] )
                        begin
                          Set_BusB_To[2:0] = IR[2:0];
                          ALU_Op = 4'b1001;
                        end
                    end // case:
8'b01000000,8'b01000001,8'b01000010,8'b01000011,8'b01000100,8'b01000101,8'b01000111,...


8'b01000110,8'b01001110,8'b01010110,8'b01011110,8'b01100110,8'b01101110,8'b01110110,8'b01111110
:
                begin
                  // BIT b,(HL)
                  MCycles = 3'b010;
                  case (1'b1) // MCycle
                    MCycle[0], MCycle[6] :
                      Set_Addr_To = aXY;
                    MCycle[1] :
                      begin
                        ALU_Op = 4'b1001;
                        TStates = 3'b100;
                      end

                    default :;
                  endcase // case(MCycle)
                end // case:
8'b01000110,8'b01001110,8'b01010110,8'b01011110,8'b01100110,8'b01101110,8'b01110110,8'b01111110


8'b11000000,8'b11000001,8'b11000010,8'b11000011,8'b11000100,8'b11000101,8'b11000111,

8'b11001000,8'b11001001,8'b11001010,8'b11001011,8'b11001100,8'b11001101,8'b11001111,

8'b11010000,8'b11010001,8'b11010010,8'b11010011,8'b11010100,8'b11010101,8'b11010111,

8'b11011000,8'b11011001,8'b11011010,8'b11011011,8'b11011100,8'b11011101,8'b11011111,

8'b11100000,8'b11100001,8'b11100010,8'b11100011,8'b11100100,8'b11100101,8'b11100111,

8'b11101000,8'b11101001,8'b11101010,8'b11101011,8'b11101100,8'b11101101,8'b11101111,

8'b11110000,8'b11110001,8'b11110010,8'b11110011,8'b11110100,8'b11110101,8'b11110111,

8'b11111000,8'b11111001,8'b11111010,8'b11111011,8'b11111100,8'b11111101,8'b11111111 :
                    begin
                      // SET b,r
                      if (MCycle[0] )
                        begin
                          ALU_Op = 4'b1010;
                          Read_To_Reg = 1'b1;
                          Save_ALU = 1'b1;
                        end
                    end // case:
8'b11000000,8'b11000001,8'b11000010,8'b11000011,8'b11000100,8'b11000101,8'b11000111,...
```

```
8'b11000110,8'b11001110,8'b11010110,8'b11011110,8'b11100110,8'b11101110,8'b11110110,8'b11111110
:
                begin
                  // SET b,(HL)
                  MCycles = 3'b011;
                  case (1'b1) // MCycle
                    MCycle[0], MCycle[6] :
                      Set_Addr_To = aXY;
                    MCycle[1] :
                      begin
                        ALU_Op = 4'b1010;
                        Read_To_Reg = 1'b1;
                        Save_ALU = 1'b1;
                        Set_Addr_To = aXY;
                        TStates = 3'b100;
                      end
                    MCycle[2] :
                      Write = 1'b1;
                    default :;
                  endcase // case(MCycle)
                end // case:
8'b11000110,8'b11001110,8'b11010110,8'b11011110,8'b11100110,8'b11101110,8'b11110110,8'b11111110


8'b10000000,8'b10000001,8'b10000010,8'b10000011,8'b10000100,8'b10000101,8'b10000111,

8'b10001000,8'b10001001,8'b10001010,8'b10001011,8'b10001100,8'b10001101,8'b10001111,

8'b10010000,8'b10010001,8'b10010010,8'b10010011,8'b10010100,8'b10010101,8'b10010111,

8'b10011000,8'b10011001,8'b10011010,8'b10011011,8'b10011100,8'b10011101,8'b10011111,

8'b10100000,8'b10100001,8'b10100010,8'b10100011,8'b10100100,8'b10100101,8'b10100111,

8'b10101000,8'b10101001,8'b10101010,8'b10101011,8'b10101100,8'b10101101,8'b10101111,

8'b10110000,8'b10110001,8'b10110010,8'b10110011,8'b10110100,8'b10110101,8'b10110111,

8'b10111000,8'b10111001,8'b10111010,8'b10111011,8'b10111100,8'b10111101,8'b10111111 :
                    begin
                      // RES b,r
                      if (MCycle[0] )
                        begin
                          ALU_Op = 4'b1011;
                          Read_To_Reg = 1'b1;
                          Save_ALU = 1'b1;
                        end
                    end // case:
8'b10000000,8'b10000001,8'b10000010,8'b10000011,8'b10000100,8'b10000101,8'b10000111,...


8'b10000110,8'b10001110,8'b10010110,8'b10011110,8'b10100110,8'b10101110,8'b10110110,8'b10111110
:
                begin
                  // RES b,(HL)
                  MCycles = 3'b011;
                  case (1'b1) // MCycle
                    MCycle[0], MCycle[6] :
                      Set_Addr_To = aXY;
                    MCycle[1] :
                      begin
                        ALU_Op = 4'b1011;
                        Read_To_Reg = 1'b1;
                        Save_ALU = 1'b1;
                        Set_Addr_To = aXY;
                        TStates = 3'b100;
                      end

                    MCycle[2] :
                      Write = 1'b1;
```

```
                  default :;
                endcase // case(MCycle)
              end // case:
8'b10000110,8'b10001110,8'b10010110,8'b10011110,8'b10100110,8'b10101110,8'b10110110,8'b10111110

            endcase // case(IR)
          end // case: 2'b01


        default :
          begin : default_ed_block

            //-------------------------------------------------------------------------
            //
            //  ED prefixed instructions
            //
            //-------------------------------------------------------------------------

            casex (IR)
              /*
               * Undocumented NOP instructions commented out to reduce size of mcode
               *

8'b00000000,8'b00000001,8'b00000010,8'b00000011,8'b00000100,8'b00000101,8'b00000110,8'b00000111

,8'b00001000,8'b00001001,8'b00001010,8'b00001011,8'b00001100,8'b00001101,8'b00001110,8'b00001111

,8'b00010000,8'b00010001,8'b00010010,8'b00010011,8'b00010100,8'b00010101,8'b00010110,8'b00010111

,8'b00011000,8'b00011001,8'b00011010,8'b00011011,8'b00011100,8'b00011101,8'b00011110,8'b00011111

,8'b00100000,8'b00100001,8'b00100010,8'b00100011,8'b00100100,8'b00100101,8'b00100110,8'b00100111

,8'b00101000,8'b00101001,8'b00101010,8'b00101011,8'b00101100,8'b00101101,8'b00101110,8'b00101111

,8'b00110000,8'b00110001,8'b00110010,8'b00110011,8'b00110100,8'b00110101,8'b00110110,8'b00110111

,8'b00111000,8'b00111001,8'b00111010,8'b00111011,8'b00111100,8'b00111101,8'b00111110,8'b00111111


,8'b10000000,8'b10000001,8'b10000010,8'b10000011,8'b10000100,8'b10000101,8'b10000110,8'b10000111

,8'b10001000,8'b10001001,8'b10001010,8'b10001011,8'b10001100,8'b10001101,8'b10001110,8'b10001111

,8'b10010000,8'b10010001,8'b10010010,8'b10010011,8'b10010100,8'b10010101,8'b10010110,8'b10010111

,8'b10011000,8'b10011001,8'b10011010,8'b10011011,8'b10011100,8'b10011101,8'b10011110,8'b10011111
                                      ,
8'b10100100,8'b10100101,8'b10100110,8'b10100111
                                     ,
8'b10101100,8'b10101101,8'b10101110,8'b10101111
                                    ,
8'b10110100,8'b10110101,8'b10110110,8'b10110111
                                   ,
8'b10111100,8'b10111101,8'b10111110,8'b10111111

,8'b11000000,8'b11000001,8'b11000010,8'b11000011,8'b11000100,8'b11000101,8'b11000110,8'b11000111

,8'b11001000,8'b11001001,8'b11001010,8'b11001011,8'b11001100,8'b11001101,8'b11001110,8'b11001111

,8'b11010000,8'b11010001,8'b11010010,8'b11010011,8'b11010100,8'b11010101,8'b11010110,8'b11010111

,8'b11011000,8'b11011001,8'b11011010,8'b11011011,8'b11011100,8'b11011101,8'b11011110,8'b11011111

,8'b11100000,8'b11100001,8'b11100010,8'b11100011,8'b11100100,8'b11100101,8'b11100110,8'b11100111

,8'b11101000,8'b11101001,8'b11101010,8'b11101011,8'b11101100,8'b11101101,8'b11101110,8'b11101111

,8'b11110000,8'b11110001,8'b11110010,8'b11110011,8'b11110100,8'b11110101,8'b11110110,8'b11110111
```

```
,8'b11111000,8'b11111001,8'b11111010,8'b11111011,8'b11111100,8'b11111101,8'b11111110,8'b11111111
:
                                                                    ; // NOP, undocumented

              8'b01111110,8'b01111111  :
                // NOP, undocumented
                ;
               */

              // 8 BIT LOAD GROUP
              8'b01010111  :
                begin
                  // LD A,I
                  Special_LD = 3'b100;
                  TStates = 3'b101;
                end

              8'b01011111  :
                begin
                  // LD A,R
                  Special_LD = 3'b101;
                  TStates = 3'b101;
                end

              8'b01000111  :
                begin
                  // LD I,A
                  Special_LD = 3'b110;
                  TStates = 3'b101;
                end

              8'b01001111  :
                begin
                  // LD R,A
                  Special_LD = 3'b111;
                  TStates = 3'b101;
                end

              // 16 BIT LOAD GROUP
              8'b01001011,8'b01011011,8'b01101011,8'b01111011  :
                begin
                  // LD dd,(nn)
                  MCycles = 3'b101;
                  case (1'b1) // MCycle
                    MCycle[1] :
                      begin
                        Inc_PC = 1'b1;
                        LDZ = 1'b1;
                      end

                    MCycle[2] :
                      begin
                        Set_Addr_To = aZI;
                        Inc_PC = 1'b1;
                        LDW = 1'b1;
                      end

                    MCycle[3] :
                      begin
                        Read_To_Reg = 1'b1;
                        if (IR[5:4] == 2'b11 )
                          begin
                            Set_BusA_To = 4'b1000;
                          end
                        else
                          begin
                            Set_BusA_To[2:1] = IR[5:4];
                            Set_BusA_To[0] = 1'b1;
                          end
                        Inc_WZ = 1'b1;
```

```
          Set_Addr_To = aZI;
        end // case: 4

      MCycle[4] :
        begin
          Read_To_Reg = 1'b1;
          if (IR[5:4] == 2'b11 )
            begin
              Set_BusA_To = 4'b1001;
            end
          else
            begin
              Set_BusA_To[2:1] = IR[5:4];
              Set_BusA_To[0] = 1'b0;
            end
        end // case: 5

      default :;
    endcase // case(MCycle)
  end // case: 8'b01001011,8'b01011011,8'b01101011,8'b01111011


8'b01000011,8'b01010011,8'b01100011,8'b01110011  :
  begin
    // LD (nn),dd
    MCycles = 3'b101;
    case (1'b1) // MCycle
      MCycle[1] :
        begin
          Inc_PC = 1'b1;
          LDZ = 1'b1;
        end

      MCycle[2] :
        begin
          Set_Addr_To = aZI;
          Inc_PC = 1'b1;
          LDW = 1'b1;
          if (IR[5:4] == 2'b11 )
            begin
              Set_BusB_To = 4'b1000;
            end
          else
            begin
              Set_BusB_To[2:1] = IR[5:4];
              Set_BusB_To[0] = 1'b1;
              Set_BusB_To[3] = 1'b0;
            end
        end // case: 3

      MCycle[3] :
        begin
          Inc_WZ = 1'b1;
          Set_Addr_To = aZI;
          Write = 1'b1;
          if (IR[5:4] == 2'b11 )
            begin
              Set_BusB_To = 4'b1001;
            end
          else
            begin
              Set_BusB_To[2:1] = IR[5:4];
              Set_BusB_To[0] = 1'b0;
              Set_BusB_To[3] = 1'b0;
            end
        end // case: 4

      MCycle[4] :
        begin
          Write = 1'b1;
        end
```

```
      default :;
    endcase // case(MCycle)
  end // case: 8'b01000011,8'b01010011,8'b01100011,8'b01110011

8'b10100000 , 8'b10101000 , 8'b10110000 , 8'b10111000  :
  begin
    // LDI, LDD, LDIR, LDDR
    MCycles = 3'b100;
    case (1'b1) // MCycle
      MCycle[0] :
        begin
          Set_Addr_To = aXY;
          IncDec_16 = 4'b1100; // BC
        end

      MCycle[1] :
        begin
          Set_BusB_To = 4'b0110;
          Set_BusA_To[2:0] = 3'b111;
          ALU_Op = 4'b0000;
          Set_Addr_To = aDE;
          if (IR[3] == 1'b0 )
            begin
              IncDec_16 = 4'b0110; // IX
            end
          else
            begin
              IncDec_16 = 4'b1110;
            end
        end // case: 2

      MCycle[2] :
        begin
          I_BT = 1'b1;
          TStates = 3'b101;
          Write = 1'b1;
          if (IR[3] == 1'b0 )
            begin
              IncDec_16 = 4'b0101; // DE
            end
          else
            begin
              IncDec_16 = 4'b1101;
            end
        end // case: 3

      MCycle[3] :
        begin
          NoRead = 1'b1;
          TStates = 3'b101;
        end

      default :;
    endcase // case(MCycle)
  end // case: 8'b10100000 , 8'b10101000 , 8'b10110000 , 8'b10111000

8'b10100001 , 8'b10101001 , 8'b10110001 , 8'b10111001  :
  begin
    // CPI, CPD, CPIR, CPDR
    MCycles = 3'b100;
    case (1'b1) // MCycle
      MCycle[0] :
        begin
          Set_Addr_To = aXY;
          IncDec_16 = 4'b1100; // BC
        end

      MCycle[1] :
        begin
          Set_BusB_To = 4'b0110;
```

```
                        Set_BusA_To[2:0] = 3'b111;
                        ALU_Op = 4'b0111;
                        Save_ALU = 1'b1;
                        PreserveC = 1'b1;
                        if (IR[3] == 1'b0 )
                          begin
                            IncDec_16 = 4'b0110;
                          end
                        else
                          begin
                            IncDec_16 = 4'b1110;
                          end
                      end // case: 2

                    MCycle[2] :
                      begin
                        NoRead = 1'b1;
                        I_BC = 1'b1;
                        TStates = 3'b101;
                      end

                    MCycle[3] :
                      begin
                        NoRead = 1'b1;
                        TStates = 3'b101;
                      end

                    default :;
                  endcase // case(MCycle)
                end // case: 8'b10100001 , 8'b10101001 , 8'b10110001 , 8'b10111001


8'b01000100,8'b01001100,8'b01010100,8'b01011100,8'b01100100,8'b01101100,8'b01110100,8'b01111100
:
                begin
                  // NEG
                  ALU_Op = 4'b0010;
                  Set_BusB_To = 4'b0111;
                  Set_BusA_To = 4'b1010;
                  Read_To_Acc = 1'b1;
                  Save_ALU = 1'b1;
                end

              8'b01000110,8'b01001110,8'b01100110,8'b01101110  :
                begin
                  // IM 0
                  IMode = 2'b00;
                end

              8'b01010110,8'b01110110  :
                // IM 1
                IMode = 2'b01;

              8'b01011110,8'b01110111  :
                // IM 2
                IMode = 2'b10;

              // 16 bit arithmetic
              8'b01001010,8'b01011010,8'b01101010,8'b01111010  :
                begin
                  // ADC HL,ss
                  MCycles = 3'b011;
                  case (1'b1) // MCycle
                    MCycle[1] :
                      begin
                        NoRead = 1'b1;
                        ALU_Op = 4'b0001;
                        Read_To_Reg = 1'b1;
                        Save_ALU = 1'b1;
                        Set_BusA_To[2:0] = 3'b101;
                        case (IR[5:4])
```

```
            0,1,2  :
              begin
                Set_BusB_To[2:1] = IR[5:4];
                Set_BusB_To[0] = 1'b1;
              end
            default :
              Set_BusB_To = 4'b1000;
          endcase
          TStates = 3'b100;
        end // case: 2

      MCycle[2] :
        begin
          NoRead = 1'b1;
          Read_To_Reg = 1'b1;
          Save_ALU = 1'b1;
          ALU_Op = 4'b0001;
          Set_BusA_To[2:0] = 3'b100;
          case (IR[5:4])
            0,1,2  :
              begin
                Set_BusB_To[2:1] = IR[5:4];
                Set_BusB_To[0] = 1'b0;
              end
            default :
              Set_BusB_To = 4'b1001;
          endcase // case(IR[5:4])
        end // case: 3

      default :;
    endcase // case(MCycle)
  end // case: 8'b01001010,8'b01011010,8'b01101010,8'b01111010

8'b01000010,8'b01010010,8'b01100010,8'b01110010  :
  begin
    // SBC HL,ss
    MCycles = 3'b011;
    case (1'b1) // MCycle
      MCycle[1] :
        begin
          NoRead = 1'b1;
          ALU_Op = 4'b0011;
          Read_To_Reg = 1'b1;
          Save_ALU = 1'b1;
          Set_BusA_To[2:0] = 3'b101;
          case (IR[5:4])
            0,1,2  :
              begin
                Set_BusB_To[2:1] = IR[5:4];
                Set_BusB_To[0] = 1'b1;
              end
            default :
              Set_BusB_To = 4'b1000;
          endcase
          TStates = 3'b100;
        end // case: 2

      MCycle[2] :
        begin
          NoRead = 1'b1;
          ALU_Op = 4'b0011;
          Read_To_Reg = 1'b1;
          Save_ALU = 1'b1;
          Set_BusA_To[2:0] = 3'b100;
          case (IR[5:4])
            0,1,2  :
              Set_BusB_To[2:1] = IR[5:4];
            default :
              Set_BusB_To = 4'b1001;
          endcase
        end // case: 3
```

```
          default :;

        endcase // case(MCycle)
      end // case: 8'b01000010,8'b01010010,8'b01100010,8'b01110010

    8'b01101111  :
      begin
        // RLD
        MCycles = 3'b100;
        case (1'b1) // MCycle
          MCycle[1] :
            begin
              NoRead = 1'b1;
              Set_Addr_To = aXY;
            end

          MCycle[2] :
            begin
              Read_To_Reg = 1'b1;
              Set_BusB_To[2:0] = 3'b110;
              Set_BusA_To[2:0] = 3'b111;
              ALU_Op = 4'b1101;
              TStates = 3'b100;
              Set_Addr_To = aXY;
              Save_ALU = 1'b1;
            end

          MCycle[3] :
            begin
              I_RLD = 1'b1;
              Write = 1'b1;
            end

          default :;
        endcase // case(MCycle)
      end // case: 8'b01101111

    8'b01100111  :
      begin
        // RRD
        MCycles = 3'b100;
        case (1'b1) // MCycle
          MCycle[1] :
            Set_Addr_To = aXY;
          MCycle[2] :
            begin
              Read_To_Reg = 1'b1;
              Set_BusB_To[2:0] = 3'b110;
              Set_BusA_To[2:0] = 3'b111;
              ALU_Op = 4'b1110;
              TStates = 3'b100;
              Set_Addr_To = aXY;
              Save_ALU = 1'b1;
            end

          MCycle[3] :
            begin
              I_RRD = 1'b1;
              Write = 1'b1;
            end

          default :;
        endcase // case(MCycle)
      end // case: 8'b01100111


8'b01000101,8'b01001101,8'b01010101,8'b01011101,8'b01100101,8'b01101101,8'b01110101,8'b01111101
:
      begin
        // RETI, RETN
```

```
                  MCycles = 3'b011;
                  case (1'b1) // MCycle
                    MCycle[0] :
                      Set_Addr_To = aSP;

                    MCycle[1] :
                      begin
                        IncDec_16 = 4'b0111;
                        Set_Addr_To = aSP;
                        LDZ = 1'b1;
                      end

                    MCycle[2] :
                      begin
                        Jump = 1'b1;
                        IncDec_16 = 4'b0111;
                        I_RETN = 1'b1;
                      end

                    default :;
                  endcase // case(MCycle)
                end // case:
8'b01000101,8'b01001101,8'b01010101,8'b01011101,8'b01100101,8'b01101101,8'b01110101,8'b01111101


8'b01000000,8'b01001000,8'b01010000,8'b01011000,8'b01100000,8'b01101000,8'b01110000,8'b01111000
:
                begin
                  // IN r,(C)
                  MCycles = 3'b010;
                  case (1'b1) // MCycle
                    MCycle[0] :
                      Set_Addr_To = aBC;

                    MCycle[1] :
                      begin
                        IORQ = 1'b1;
                        if (IR[5:3] != 3'b110 )
                          begin
                            Read_To_Reg = 1'b1;
                            Set_BusA_To[2:0] = IR[5:3];
                          end
                        I_INRC = 1'b1;
                      end

                    default :;
                  endcase // case(MCycle)
                end // case:
8'b01000000,8'b01001000,8'b01010000,8'b01011000,8'b01100000,8'b01101000,8'b01110000,8'b01111000


8'b01000001,8'b01001001,8'b01010001,8'b01011001,8'b01100001,8'b01101001,8'b01110001,8'b01111001
:
                begin
                  // OUT (C),r
                  // OUT (C),0
                  MCycles = 3'b010;
                  case (1'b1) // MCycle
                    MCycle[0] :
                      begin
                        Set_Addr_To = aBC;
                        Set_BusB_To[2:0]        = IR[5:3];
                        if (IR[5:3] == 3'b110 )
                          begin
                            Set_BusB_To[3] = 1'b1;
                          end
                      end

                    MCycle[1] :
                      begin
                        Write = 1'b1;
```

```
                    IORQ = 1'b1;
                  end

                default :;
              endcase // case(MCycle)
            end // case:
8'b01000001,8'b01001001,8'b01010001,8'b01011001,8'b01100001,8'b01101001,8'b01110001,8'b01111001

          8'b10100010 , 8'b10101010 , 8'b10110010 , 8'b10111010  :
            begin
              // INI, IND, INIR, INDR
              MCycles = 3'b100;
              case (1'b1) // MCycle
                MCycle[0] :
                  begin
                    Set_Addr_To = aBC;
                    Set_BusB_To = 4'b1010;
                    Set_BusA_To = 4'b0000;
                    Read_To_Reg = 1'b1;
                    Save_ALU = 1'b1;
                    ALU_Op = 4'b0010;
                  end

                MCycle[1] :
                  begin
                    IORQ = 1'b1;
                    Set_BusB_To = 4'b0110;
                    Set_Addr_To = aXY;
                  end

                MCycle[2] :
                  begin
                    if (IR[3] == 1'b0 )
                      begin
                       IncDec_16 = 4'b0110;
                      end
                    else
                      begin
                       IncDec_16 = 4'b1110;
                      end
                    TStates = 3'b100;
                    Write = 1'b1;
                    I_BTR = 1'b1;
                  end // case: 3

                MCycle[3] :
                  begin
                    NoRead = 1'b1;
                    TStates = 3'b101;
                  end

                default :;
              endcase // case(MCycle)
            end // case: 8'b10100010 , 8'b10101010 , 8'b10110010 , 8'b10111010

          8'b10100011 , 8'b10101011 , 8'b10110011 , 8'b10111011  :
            begin
              // OUTI, OUTD, OTIR, OTDR
              MCycles = 3'b100;
              case (1'b1) // MCycle
                MCycle[0] :
                  begin
                    TStates = 3'b101;
                    Set_Addr_To = aXY;
                    Set_BusB_To = 4'b1010;
                    Set_BusA_To = 4'b0000;
                    Read_To_Reg = 1'b1;
                    Save_ALU = 1'b1;
                    ALU_Op = 4'b0010;
                  end
```

```
              MCycle[1] :
                begin
                  Set_BusB_To = 4'b0110;
                  Set_Addr_To = aBC;
                  if (IR[3] == 1'b0 )
                    begin
                      IncDec_16 = 4'b0110;
                    end
                  else
                    begin
                      IncDec_16 = 4'b1110;
                    end
                end

              MCycle[2] :
                begin
                  if (IR[3] == 1'b0 )
                    begin
                      IncDec_16 = 4'b0010;
                    end
                  else
                    begin
                      IncDec_16 = 4'b1010;
                    end
                  IORQ = 1'b1;
                  Write = 1'b1;
                  I_BTR = 1'b1;
                end // case: 3

              MCycle[3] :
                begin
                  NoRead = 1'b1;
                  TStates = 3'b101;
                end

              default :;
            endcase // case(MCycle)
          end // case: 8'b10100011 , 8'b10101011 , 8'b10110011 , 8'b10111011

      endcase // case(IR)
    end // block: default_ed_block
endcase // case(ISet)

if (Mode == 1 )
  begin
    if (MCycle[0] )
      begin
        //TStates = 3'b100;
      end
    else
      begin
        TStates = 3'b011;
      end
  end

if (Mode == 3 )
  begin
    if (MCycle[0] )
      begin
        //TStates = 3'b100;
      end
    else
      begin
        TStates = 3'b100;
      end
  end

if (Mode < 2 )
  begin
    if (MCycle[5] )
      begin
```

```
              Inc_PC = 1'b1;
              if (Mode == 1 )
                begin
                  Set_Addr_To = aXY;
                  TStates = 3'b100;
                  Set_BusB_To[2:0] = SSS;
                  Set_BusB_To[3] = 1'b0;
                end
              if (IR == 8'b00110110 || IR == 8'b11001011 )
                begin
                  Set_Addr_To = aNone;
                end
            end
          if (MCycle[6] )
            begin
              if (Mode == 0 )
                begin
                  TStates = 3'b101;
                end
              if (ISet != 2'b01 )
                begin
                  Set_Addr_To = aXY;
                end
              Set_BusB_To[2:0] = SSS;
              Set_BusB_To[3] = 1'b0;
              if (IR == 8'b00110110 || ISet == 2'b01 )
                begin
                  // LD (HL),n
                  Inc_PC = 1'b1;
                end
              else
                begin
                  NoRead = 1'b1;
                end
            end
        end // if (Mode < 2 )

    end // always @ (IR, ISet, MCycle, F, NMICycle, IntCycle)
endmodule // T80_MCode
```

tv80_alu.v

```
//
// TV80 8-Bit Microprocessor Core
// Based on the VHDL T80 core by Daniel Wallner (jesus@opencores.org)
//
// Copyright (c) 2004 Guy Hutchison (ghutchis@opencores.org)
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included
// in all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
// EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
// MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
// IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
// CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
// TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
// SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

module tv80_alu (/*AUTOARG*/
  // Outputs
  Q, F_Out,
  // Inputs
```

```
Arith16, Z16, ALU_Op, IR, ISet, BusA, BusB, F_In
);

parameter            Mode   = 3;
parameter            Flag_C = 0;
parameter            Flag_N = 1;
parameter            Flag_P = 2;
parameter            Flag_X = 3;
parameter            Flag_H = 4;
parameter            Flag_Y = 5;
parameter            Flag_Z = 6;
parameter            Flag_S = 7;

input                Arith16;
input                Z16;
input [3:0]           ALU_Op ;
input [5:0]           IR;
input [1:0]           ISet;
input [7:0]           BusA;
input [7:0]           BusB;
input [7:0]           F_In;
output [7:0]          Q;
output [7:0]          F_Out;
reg [7:0]             Q;
reg [7:0]             F_Out;

function [4:0] AddSub4;
  input [3:0] A;
  input [3:0] B;
  input Sub;
  input Carry_In;
  begin
    AddSub4 = { 1'b0, A } + { 1'b0, (Sub)?~B:B } + Carry_In;
  end
endfunction // AddSub4

function [3:0] AddSub3;
  input [2:0] A;
  input [2:0] B;
  input Sub;
  input Carry_In;
  begin
    AddSub3 = { 1'b0, A } + { 1'b0, (Sub)?~B:B } + Carry_In;
  end
endfunction // AddSub4

function [1:0] AddSub1;
  input A;
  input B;
  input Sub;
  input Carry_In;
  begin
    AddSub1 = { 1'b0, A } + { 1'b0, (Sub)?~B:B } + Carry_In;
  end
endfunction // AddSub4

// AddSub variables (temporary signals)
reg UseCarry;
reg Carry7_v;
reg OverFlow_v;
reg HalfCarry_v;
reg Carry_v;
reg [7:0] Q_v;

reg [7:0] BitMask;


always @(/*AUTOSENSE*/ALU_Op or BusA or BusB or F_In or IR)
  begin
    case (IR[5:3])
      3'b000 : BitMask = 8'b00000001;
```

```
        3'b001 : BitMask = 8'b00000010;
        3'b010 : BitMask = 8'b00000100;
        3'b011 : BitMask = 8'b00001000;
        3'b100 : BitMask = 8'b00010000;
        3'b101 : BitMask = 8'b00100000;
        3'b110 : BitMask = 8'b01000000;
        default: BitMask = 8'b10000000;
      endcase // case(IR[5:3])

      UseCarry = ~ ALU_Op[2] && ALU_Op[0];
      { HalfCarry_v, Q_v[3:0] } = AddSub4(BusA[3:0], BusB[3:0], ALU_Op[1], ALU_Op[1] ^ (UseCarry
&& F_In[Flag_C]) );
      { Carry7_v, Q_v[6:4]  } = AddSub3(BusA[6:4], BusB[6:4], ALU_Op[1], HalfCarry_v);
      { Carry_v, Q_v[7] } = AddSub1(BusA[7], BusB[7], ALU_Op[1], Carry7_v);
      OverFlow_v = Carry_v ^ Carry7_v;
    end // always @ *

  reg [7:0] Q_t;
  reg [8:0] DAA_Q;

  always @ (/*AUTOSENSE*/ALU_Op or Arith16 or BitMask or BusA or BusB
            or Carry_v or F_In or HalfCarry_v or IR or ISet
            or OverFlow_v or Q_v or Z16)
    begin
      Q_t = 8'hxx;
      DAA_Q = {9{1'bx}};

      F_Out = F_In;
      case (ALU_Op)
        4'b0000, 4'b0001,  4'b0010, 4'b0011, 4'b0100, 4'b0101, 4'b0110, 4'b0111 :
          begin
            F_Out[Flag_N] = 1'b0;
            F_Out[Flag_C] = 1'b0;

            case (ALU_Op[2:0])

              3'b000, 3'b001 : // ADD, ADC
                begin
                  Q_t = Q_v;
                  F_Out[Flag_C] = Carry_v;
                  F_Out[Flag_H] = HalfCarry_v;
                  F_Out[Flag_P] = OverFlow_v;
                end

              3'b010, 3'b011, 3'b111 : // SUB, SBC, CP
                begin
                  Q_t = Q_v;
                  F_Out[Flag_N] = 1'b1;
                  F_Out[Flag_C] = ~ Carry_v;
                  F_Out[Flag_H] = ~ HalfCarry_v;
                  F_Out[Flag_P] = OverFlow_v;
                end

              3'b100 : // AND
                begin
                  Q_t[7:0] = BusA & BusB;
                  F_Out[Flag_H] = 1'b1;
                end

              3'b101 : // XOR
                begin
                  Q_t[7:0] = BusA ^ BusB;
                  F_Out[Flag_H] = 1'b0;
                end

              default : // OR 3'b110
                begin
                  Q_t[7:0] = BusA | BusB;
                  F_Out[Flag_H] = 1'b0;
                end
```

```
    endcase // case(ALU_OP[2:0])

    if (ALU_Op[2:0] == 3'b111 )
       begin // CP
        F_Out[Flag_X] = BusB[3];
        F_Out[Flag_Y] = BusB[5];
      end
     else
       begin
        F_Out[Flag_X] = Q_t[3];
        F_Out[Flag_Y] = Q_t[5];
      end

    if (Q_t[7:0] == 8'b00000000 )
       begin
        F_Out[Flag_Z] = 1'b1;
        if (Z16 == 1'b1 )
           begin
            F_Out[Flag_Z] = F_In[Flag_Z];     // 16 bit ADC,SBC
          end
      end
     else
       begin
        F_Out[Flag_Z] = 1'b0;
      end // else: !if(Q_t[7:0] == 8'b00000000 )

    F_Out[Flag_S] = Q_t[7];
    case (ALU_Op[2:0])
      3'b000, 3'b001, 3'b010, 3'b011, 3'b111 : // ADD, ADC, SUB, SBC, CP
         ;

      default :
        F_Out[Flag_P] = ~(^Q_t);
    endcase // case(ALU_Op[2:0])

    if (Arith16 == 1'b1 )
       begin
        F_Out[Flag_S] = F_In[Flag_S];
        F_Out[Flag_Z] = F_In[Flag_Z];
        F_Out[Flag_P] = F_In[Flag_P];
      end
   end // case: 4'b0000, 4'b0001,  4'b0010, 4'b0011, 4'b0100, 4'b0101, 4'b0110, 4'b0111

4'b1100 :
   begin
    // DAA
    F_Out[Flag_H] = F_In[Flag_H];
    F_Out[Flag_C] = F_In[Flag_C];
    DAA_Q[7:0] = BusA;
    DAA_Q[8] = 1'b0;
    if (F_In[Flag_N] == 1'b0 )
       begin
        // After addition
        // Alow > 9 || H == 1
        if (DAA_Q[3:0] > 9 || F_In[Flag_H] == 1'b1 )
           begin
            if ((DAA_Q[3:0] > 9) )
               begin
                F_Out[Flag_H] = 1'b1;
              end
             else
               begin
                F_Out[Flag_H] = 1'b0;
              end
            DAA_Q = DAA_Q + 6;
          end // if (DAA_Q[3:0] > 9 || F_In[Flag_H] == 1'b1 )

        // new Ahigh > 9 || C == 1
        if (DAA_Q[8:4] > 9 || F_In[Flag_C] == 1'b1 )
           begin
            DAA_Q = DAA_Q + 96; // 0x60
```

```
          end
        end
      else
        begin
         // After subtraction
         if (DAA_Q[3:0] > 9 || F_In[Flag_H] == 1'b1 )
            begin
             if (DAA_Q[3:0] > 5 )
                begin
                 F_Out[Flag_H] = 1'b0;
               end
             DAA_Q[7:0] = DAA_Q[7:0] - 6;
           end
         if (BusA > 153 || F_In[Flag_C] == 1'b1 )
            begin
             DAA_Q = DAA_Q - 352; // 0x160
           end
        end // else: !if(F_In[Flag_N] == 1'b0 )

      F_Out[Flag_X] = DAA_Q[3];
      F_Out[Flag_Y] = DAA_Q[5];
      F_Out[Flag_C] = F_In[Flag_C] || DAA_Q[8];
      Q_t = DAA_Q[7:0];

      if (DAA_Q[7:0] == 8'b00000000 )
         begin
          F_Out[Flag_Z] = 1'b1;
        end
       else
         begin
          F_Out[Flag_Z] = 1'b0;
        end

      F_Out[Flag_S] = DAA_Q[7];
      F_Out[Flag_P] = ~ (^DAA_Q);
    end // case: 4'b1100

  4'b1101, 4'b1110 :
    begin
      // RLD, RRD
      Q_t[7:4] = BusA[7:4];
      if (ALU_Op[0] == 1'b1 )
         begin
          Q_t[3:0] = BusB[7:4];
        end
       else
         begin
          Q_t[3:0] = BusB[3:0];
        end
      F_Out[Flag_H] = 1'b0;
      F_Out[Flag_N] = 1'b0;
      F_Out[Flag_X] = Q_t[3];
      F_Out[Flag_Y] = Q_t[5];
      if (Q_t[7:0] == 8'b00000000 )
         begin
          F_Out[Flag_Z] = 1'b1;
        end
       else
         begin
          F_Out[Flag_Z] = 1'b0;
        end
      F_Out[Flag_S] = Q_t[7];
       F_Out[Flag_P] = ~(^Q_t);
    end // case: when 4'b1101, 4'b1110

  4'b1001 :
    begin
      // BIT
      Q_t[7:0] = BusB & BitMask;
      F_Out[Flag_S] = Q_t[7];
      if (Q_t[7:0] == 8'b00000000 )
```

```
       begin
        F_Out[Flag_Z] = 1'b1;
        F_Out[Flag_P] = 1'b1;
       end
     else
       begin
        F_Out[Flag_Z] = 1'b0;
        F_Out[Flag_P] = 1'b0;
       end
    F_Out[Flag_H] = 1'b1;
    F_Out[Flag_N] = 1'b0;
    F_Out[Flag_X] = 1'b0;
    F_Out[Flag_Y] = 1'b0;
    if (IR[2:0] != 3'b110 )
       begin
        F_Out[Flag_X] = BusB[3];
        F_Out[Flag_Y] = BusB[5];
       end
   end // case: when 4'b1001

4'b1010 :
  // SET
  Q_t[7:0] = BusB | BitMask;

4'b1011 :
  // RES
  Q_t[7:0] = BusB & ~ BitMask;

4'b1000 :
   begin
    // ROT
    case (IR[5:3])
      3'b000 : // RLC
         begin
          Q_t[7:1] = BusA[6:0];
          Q_t[0] = BusA[7];
          F_Out[Flag_C] = BusA[7];
         end

      3'b010 : // RL
         begin
          Q_t[7:1] = BusA[6:0];
          Q_t[0] = F_In[Flag_C];
          F_Out[Flag_C] = BusA[7];
         end

      3'b001 : // RRC
         begin
          Q_t[6:0] = BusA[7:1];
          Q_t[7] = BusA[0];
          F_Out[Flag_C] = BusA[0];
         end

      3'b011 : // RR
         begin
          Q_t[6:0] = BusA[7:1];
          Q_t[7] = F_In[Flag_C];
          F_Out[Flag_C] = BusA[0];
         end

      3'b100 : // SLA
         begin
          Q_t[7:1] = BusA[6:0];
          Q_t[0] = 1'b0;
          F_Out[Flag_C] = BusA[7];
         end

      3'b110 : // SLL (Undocumented) / SWAP
         begin
          if (Mode == 3 )
             begin
```

```
                  Q_t[7:4] = BusA[3:0];
                  Q_t[3:0] = BusA[7:4];
                  F_Out[Flag_C] = 1'b0;
                end
               else
                 begin
                  Q_t[7:1] = BusA[6:0];
                  Q_t[0] = 1'b1;
                  F_Out[Flag_C] = BusA[7];
                end // else: !if(Mode == 3 )
            end // case: 3'b110

          3'b101 : // SRA
            begin
             Q_t[6:0] = BusA[7:1];
             Q_t[7] = BusA[7];
             F_Out[Flag_C] = BusA[0];
            end

          default : // SRL
            begin
             Q_t[6:0] = BusA[7:1];
             Q_t[7] = 1'b0;
             F_Out[Flag_C] = BusA[0];
            end
        endcase // case(IR[5:3])

        F_Out[Flag_H] = 1'b0;
        F_Out[Flag_N] = 1'b0;
        F_Out[Flag_X] = Q_t[3];
        F_Out[Flag_Y] = Q_t[5];
        F_Out[Flag_S] = Q_t[7];
        if (Q_t[7:0] == 8'b00000000 )
          begin
           F_Out[Flag_Z] = 1'b1;
          end
        else
          begin
           F_Out[Flag_Z] = 1'b0;
          end
        F_Out[Flag_P] = ~(^Q_t);

        if (ISet == 2'b00 )
          begin
           F_Out[Flag_P] = F_In[Flag_P];
           F_Out[Flag_S] = F_In[Flag_S];
           F_Out[Flag_Z] = F_In[Flag_Z];
          end
      end // case: 4'b1000


     default :
       ;

   endcase // case(ALU_Op)

   Q = Q_t;
 end // always @ (Arith16, ALU_OP, F_In, BusA, BusB, IR, Q_v, Carry_v, HalfCarry_v,
OverFlow_v, BitMask, ISet, Z16)

endmodule // T80_ALU
```

tv80_reg.v

```
//
// TV80 8-Bit Microprocessor Core
// Based on the VHDL T80 core by Daniel Wallner (jesus@opencores.org)
//
// Copyright (c) 2004 Guy Hutchison (ghutchis@opencores.org)
//
```

```
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included
// in all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
// EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
// MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
// IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
// CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
// TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
// SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

module tv80_reg (/*AUTOARG*/
  // Outputs
  DOBH, DOAL, DOCL, DOBL, DOCH, DOAH, BC, DE, HL,
  // Inputs
  AddrC, AddrA, AddrB, DIH, DIL, clk, CEN, WEH, WEL
  );
    input  [2:0] AddrC;
    output [7:0] DOBH;
    input  [2:0] AddrA;
    input  [2:0] AddrB;
    input  [7:0] DIH;
    output [7:0] DOAL;
    output [7:0] DOCL;
    input  [7:0] DIL;
    output [7:0] DOBL;
    output [7:0] DOCH;
    output [7:0] DOAH;
    input  clk, CEN, WEH, WEL;

         output [15:0] BC;
         output [15:0] DE;
         output [15:0] HL;

  reg [7:0] RegsH [0:7];
  reg [7:0] RegsL [0:7];

  always @(posedge clk)
    begin
      if (CEN)
        begin
          if (WEH) RegsH[AddrA] <= DIH;
          if (WEL) RegsL[AddrA] <= DIL;
        end
    end

  assign DOAH = RegsH[AddrA];
  assign DOAL = RegsL[AddrA];
  assign DOBH = RegsH[AddrB];
  assign DOBL = RegsL[AddrB];
  assign DOCH = RegsH[AddrC];
  assign DOCL = RegsL[AddrC];

  // break out ram bits for waveform debug
  wire [7:0] H = RegsH[2];
  wire [7:0] L = RegsL[2];

  assign BC = { RegsH[0], RegsL[0] };
  assign DE = { RegsH[1], RegsL[1] };
  assign HL = { RegsH[2], RegsL[2] };

// synopsys dc_script_begin
// set_attribute current_design "revision" "$Id: tv80_reg.v,v 1.1 2004-05-16 17:39:57 ghutchis
Exp $" -type string -quiet
```

```
// synopsys dc_script_end
endmodule
```