The Design and Implementation of the Nintendo Entertainment System

Jonathan Downey
Lauri Kauppila
Brian Myhre

6.111
Introductory Digital Systems Laboratory
Professor Chris Terman

December 9, 2004

Abstract

Two decades ago the Nintendo Entertainment System entered the US market and continues to profoundly influence and define American culture and consumer electronics. For the final project in Introductory Digital Systems Laboratory, our team decided to design and construct the console to its original specifications. The system is composed of two major subsystems, the Central Processing Unit (CPU) and Picture Processing Unit (PPU), and we divided up tasks along the lines of these major functional components. The project began with extensive research into the technical specifications and design. From there we implemented the system on three FPGAs, one for the CPU, one for the Picture Processing, and one for the VGA Output Generation, using the Verilog hardware description language. In the end, the various components were not fully integrated into a complete console, but we achieved all our major system design goals and are very close to a working Nintendo Entertainment System.

Table of Contents

Table of Contents (cont…)

List of Tables

List of Figures

Project Motivation

When our team set out to select a project for Introductory Digital Systems Laboratory, we first created a clearly defined list of priorities and motivations to guide this process. We decided to find a project with distinct subsystem, so work could be divided in an efficient and logical manner among the members of our team. This would also introduce a greater variety and diversity of interesting aspects for us to complete.

We were also looking for a project with complex behavior and architecture that would challenge our electrical engineering skills and provide ample opportunity to exercise and demonstrate the concepts presented during 6.111 lectures. An implementation that demanded the versatility, performance capabilities, and substantial amount of logic available to us with the field programmable gate array (FPGA) was of high interest as well. We were not interested in something that could be easily accomplished with a microcontroller or wired combinations of discrete logic units.

Another motivation in project selection was to achieve high entertainment value. This allows for greater appreciation of our accomplishment by non-technical people and other not familiar with the real challenges of complex digital system design.

Our selection was an excellent match for these goals and aligned very well with our motivations. We decided to design and build a hardware emulation of the original Nintendo Entertainment System to its native specifications.

Background

In 1983 the immediate precursor to the Nintendo Entertainment System was released in Japan under the name Famicom, which was an abbreviation for “Family Computer.” It sold over 47,000 units within the first six months and its first games included Nintendo classics such as Donkey Kong and Mario Brothers.

http://en.wikipedia.org


http://en.wikipedia.org

The video US video game market witness a major crash in 1983-1984, and although Nintendo was eagerly working to bring its system to American markets, interest among distributors and retailers was very low. The company decided to redesign the case to look more like a computer and renamed it “Nintendo Entertainment System.” Then, with a promise to retailers that it would buy back any unsold systems, the NES was released in 1985. The console, packaged with the games Duck Hunt and Super Mario Brothers, went for $199.

The Nintendo was in production for a full decade and represented an incredible success for the company. Nintendo sold over 62 million systems and 500 million games. Today it is the most widely emulated system, with other thirty different software versions.

Project Overview

The Nintendo system allows for fantastic games and colorful graphics with very limited resources. Smart architecture was developed to make sure that all resources are utilized to their fullest extent, allowing for a small and cost-effective product. Our goal was to design and build a system that could play the original Nintendo games when provided user input through the original Nintendo controllers.

The Nintendo's functionality is based on two main components: a 6502-family processor (CPU) and the picture processing unit (PPU). These two components interface with small amounts of memory to produce a game. The CPU interfaces with controllers and produces the audio output. The PPU processes graphics data and outputs either a PAL or NTSC standard video signal.

The CPU is a custom 8-bit 2A03, which includes the processor core, onboard audio generation, and the controller interface. The core is a 6502 running at 1.79 MHz. The audio is created from two square waves, one triangle wave, one digital input, and one noise channel. The control pads deliver data to the system from a shift register.

The PPU was custom designed by Nintendo to perform the video construction and generation. It manipulates sprite and background tiles to produce images from pattern data stored inside the game cartridge and internal Video RAM. The color palette has 64 values, and the output is 256 by 240 pixels.

Overall Nintendo System Architecture

In general, the CPU will interface with the program ROM of a game cartridge to construct a game in an abstract sense (variables, game rules, etc.), and then informs on a high level what the PPU should display on the screen. The PPU, based on CPU commands, does lots of fast processing (including rapid memory fetches) to produce 256x240 pixel color output to a screen.

To be efficient, the system uses multiple small regions of memory in parallel. The following is a list of all the memories that exist:

name	size	description	location	access
System Memory	2kb	Holds temporary game data that is accessed by the CPU	CPU address space	CPU
VRAM	2kb	holds high-level graphics data: two nametables and two attribute tables [see PPU section]	PPU address space	PPU
Program ROM	32+ kb	Game data. Includes sound data, game color data, and game execution data.	CPU address space	CPU
Character ROM	8+ kb	Holds the patterns for game graphics. Does NOT include color information.	PPU address space	PPU
Sprite RAM	256 bytes	Holds up to 64 sprites to be displayed in any given frame	PPU (internal)	PPU/CPU
Temporary Sprite RAM	24 bytes	Temporary space that holds information for up to 8 sprites to be displayed on the next scanline.	PPU (internal)	PPU
Color palette RAM	32 bytes	Contains information about how to color sprites and background.	PPU (internal)	PPU/CPU

Table 1 – Memory of the Nintendo Entertainment System

The full schematic of the system is found in appendix A. It shows the exact interface between different parts of the system, and helps to emphasize the compactness of the final product. For example, it is worth nothing that the PPU "AD" lines are used for bidirectional data flow to VRAM, bidirectional data flow to character ROM, address lines to VRAM, and address lines to program ROM. The CPU memory space is equally complex, where we find that the same data lines are used for system memory and program ROM, and the same address lines are used for program ROM, system memory, and for accessing PPU registers. Careful timing is needed to make sure that data contention doesn't happen. What's more, this architecture emphasizes how busy the system is during the processing of a game: there is hardly ever a time when data lines are not being used.

Information was compiled from several sources:

- An engineering schematic of Famicom, a close version of the Nintendo sold in Japan
- a reverse-engineered document produced by Electonix Corp.
- reverese-engineered cartridge pinouts, found on nesdev.pariodius.com, a website for Nintendo engineering enthusiasts

Our final schematic considered the merits of each information source, considered extra information that was deduced by our team, and compared conflicting information between reverse-engineered documents. Our final schematic is believed to be completely accurate.

The Central Processing Unit

The Central Processing Unit, or CPU, is located inside the custom 2A03 chip on the Nintendo. This CPU core is a 6502 microprocessor and is responsible for directly or indirectly controlling every other device in the Nintendo. The CPU interfaces with the system RAM, cartridge program ROM, and several peripheral devices such as the controller interface, Audio Processing Unit, and Picture Processing Unit. All of these devices interact together as the microcomputer system.

The Historic 6502

The 6502 is a historic 8-bit microprocessor originally designed by MOS Technology in 1975. The 6500 family of processors was born after several designers of the Motorola 6800 left Motorola unhappy with the recent release of the 6800. After joining MOS Technology (which later became the Commodore Semiconductor Group), these designers set out to develop a new CPU that would be able to out-perform the Motorola 6800. When introduced, the 6502 not only outperformed the 6800 but was the least expensive yet fully-featured CPU on the market. The 6502 only cost $25 while the 6800 was selling for six times that. The 6502 and Zilog Z80 are considered responsible for sparking off a series of computer development projects that would eventually result in the home computer revolution of the 1980s.

The 6502 was a landmark in microprocessor design. It was able to have very few registers because of its extremely fast access to RAM and pipelined bus technology. The processor's internal logic also ran at the same speed as its peripherals which allowed for a lower system cost.

Along with being used in the Nintendo Entertainment System and Japanese Famicom System, the 6502 was also used in the Apple I and Apple II families, the Atari, and the Commodore 64. The 6502 is still produced today for embedded systems. The legacy of the 6502 is that its efficient design is said to have inspired the development of the ARM RISC processors which are now used in many handheld devices.

System Hardware

Bus Architecture

The 6502 communicates with RAM, ROM, and its peripherals using two primary busses. The address bus is a 16-bit uni-directional bus that is always generated by the CPU. This bus controls which device the CPU is communicating with and what address the CPU is trying to access. In all, the CPU is capable of addressing up to 64 KB of memory or memory-mapped peripherals. A read-write line accompanies the address bus to specify whether the CPU is performing a read or write operation. Because the convention is for this line to be low during a write operation, this line also functions as the write enable line to RAM.

Along with the address bus, the 6502 also has an 8-bit bi-directional data bus. This data bus is used to specify the byte of data that the CPU is either reading or writing. Because this bus is bidirectional, each peripheral may write to it at different times as specified by the address bus. The proper timing for these two busses is shown in Figure 1.

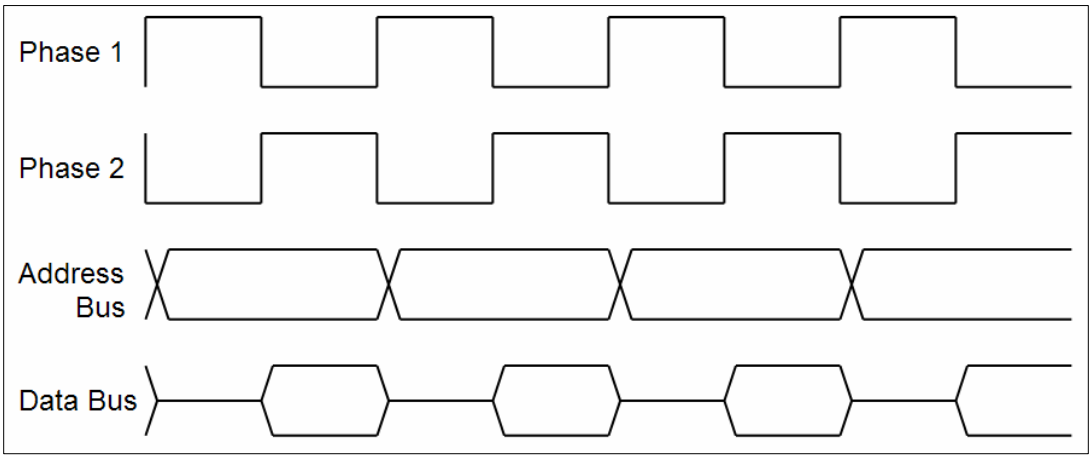


Figure 1 – 6502 Bus Access Timing Diagram

Because the address bus and read-write line are always written to by the CPU, the address bus and read-write line change their values on the rising edge of every phase one clock. The data bus however must be tri-stated (floating) during the phase one clock and is only driven during the phase two clock. This allows the peripheral devices time to decode the address before deciding whether or not to input or output data. When the phase two clock is high, one device should be writing to the data

Which device is reading given data or writing data to the data bus is defined by the memory map. The memory map for the CPU in the Nintendo is given in Table 2. For example, if the address is less than 0800h, the CPU is accessing the system RAM.

Start	End	Size	Description
0x0000	0x07ff	0x0800	System RAM
0x2000	0x2007	0x0008	PPU Registers
0x4000	0x4016	0x0017	Internal Registers for Audio and Controllers
0x5000	0x5fff	0x1000	Expansion Modules
0x8000	0xffff	0x8000	Cartridge Program ROM

Table 2 – CPU Memory Map

Instruction Set

The 6502 is an accumulator plus index register machine. It executes 56 machine code instructions where each instruction can take anywhere from two to seven clock cycles to execute. The 6502 is able to load a byte of memory into its accumulator or index registers. It can then perform arithmetic operations on that byte, transfer it from one register to another, and write it back to memory. In addition, the 6502 has several operations which read, modify and then write a byte in memory, all in one instruction. Along with arithmetic operations, the 6502 is able to control its program flow by changing flags and altering its program counter by branching to different instructions. With the addition of the stack, the 6502 is able to push bytes of data onto the stack and retrieve them later. This allows the CPU to jump to subroutines and allows for interrupts and context saving. A detailed look at the Instruction Set is included in Appendix C.

Addressing Modes

In order to retrieve data for an operation, the 6502 has thirteen addressing modes. These addressing modes are listed briefly below: (A detailed look at Addressing Modes is also in Appendix C)

- Accumulator – The data in the accumulator is used.
- Immediate - The byte in memory immediately following the instruction is used.
- Zero Page – The Nth byte in the first page of RAM is used where N is the byte in memory immediately following the instruction.
- Zero Page, X Index – The (N+X)th byte in the first page of RAM is used where N is the byte in memory immediately following the instruction and X is the contents of the X index register.
- Zero Page, Y Index – Same as above but with the Y index register
- Absolute – The two bytes in memory following the instruction specify the absolute address of the byte of data to be used.
- Absolute, X Index - The two bytes in memory following the instruction specify the base address. The contents of the X index register are then added to the base address to obtain the address of the byte of data to be used.
- Absolute, Y Index – Same as above but with the Y index register
- Implied – Data is either not needed or the location of the data is implied by the instruction.
- Relative – The sum of the program counter and the byte in memory immediately following the instruction is used.
- (Indirect, X) – A combination of Indirect Addressing and Indexed Addressing
- (Indirect) , Y - A combination of Indirect Addressing and Indexed Addressing
- Absolute Indirect - The two bytes in memory following the instruction specify the absolute address of the two bytes that contain the absolute address of the byte of data to be used.

Interrupts

The “break” instruction and interrupt operations are all very similar in execution. After the fetch of a break instruction, the CPU writes the program counter and status register to the stack thereby saving the current state of the CPU so that it can restore that state at a later time. Then the CPU sets the break flag and does an indirect jump to the location specified by the contents of the interrupt vector.

The 6502 processor responds to three types of interrupts. When the reset line is brought low (as it is when the system starts), a reset interrupt is generated. This causes the processor to begin executing a modified “break” instruction. The modification is that the read-write line is held high. If this were not the case, the CPU would write to random registers or bytes of data during the context save. This is because the internal contents of the CPU are unknown before initialization and is the entire reason that a reset must occur. After the “fake” context save, the CPU does an indirect jump to the reset vector which is at a different location than the interrupt vector. This indirect jump is done by reading in the two bytes of data at the reset vector and loading the program counter with their combined value. This causes the CPU to begin executing

code at a new location where code should be located to handle the initialization sequence. The break bit is cleared during a reset interrupt.

The second type of interrupt (and second in priority) is the non-maskable interrupt (NMI). This interrupt occurs on a negative transition of the NMI line to the CPU. When this NMI interrupt is triggered, the CPU forces a break instruction and proceeds by saving the CPU's context. After the context save, the CPU performs the indirect jump as with the reset interrupt but to a location specified by the NMI vector. The break bit is cleared during an NMI.

The final type of interrupt is the general external interrupt (IRQ). This interrupt occurs whenever the IRQ line is low during an instruction fetch and the general interrupt is enabled. Instead of fetching the instruction, the CPU forces a break instruction executes it instead. The only difference is that the break bit is not set just as in the NMI and reset interrupts. Unlike the NMI, the general IRQ can be enabled and disabled in software.

After executing the proper code, the CPU is able to return to the point it was at before executing the interrupt. It does this by restoring the context that was saved. This is called a "return from interrupt" instruction.

Implementation

Clock Divider

Nintendo's version of the 6502 has an internal clock divider that allows the 6502 to use the same clock as the Picture Processing Unit but run at a different frequency. By dividing the 21.47727 MHz main clock by twelve, the 6502 runs at 1.79 MHz, well within the 2 MHz limit of the technology. This 1.79 MHZ clock is the phase one clock and is abbreviated clk1 in most of our documentation. The phase two clock (clk2) is also generated from the clock divider. This clock is the same frequency as the phase one clock but is exactly out of phase with it. The phase one clock is used by the 6502 core and the other devices internal to the 2A03 chip as their main clock. The phase two clock is used by peripheral devices and the data bus driver to ensure that the data bus is tri-stated during the phase one clock.

Interrupt Controller

This module allows for all three types of external interrupts. Based on the RESET, IRQ, and NMI lines during an instruction fetch, the interrupt controller signals the control-signal generator to either execute the fetched instruction or to force a break instruction. The interrupt controller also specifies a vector mode which is either normal, interrupt, non-maskable, or reset. This vector mode allows the control-signal generator to cause an indirect jump to the proper location in memory.

Timing Control

The timing control module generates the cycle number that the processor is currently executing. All instruction fetches happen during cycle zero. From there the cycle number is incremented on each positive edge of the phase one clock. Once the instruction is complete (as specified by the control-signal generator), the cycle number reverts back to cycle zero and another instruction is fetched.

Instruction Register

The instruction register is loaded with a new instruction during every cycle zero. It is either loaded with a byte from the data bus, or forced to the break instruction if an interrupt has occurred. While the CPU is executing a cycle other than zero, the instruction register holds its contents.

Instruction Decode

The instruction decode register uses the machine instruction in the instruction register and breaks it into an opcode and an addressing mode. Since operations can have several addressing modes, several machine instructions can result in the same opcode but with different addressing modes.

Control-Signal Generator

The control signal generator specifies all of the control-signals to each part of the CPU on every clock cycle. Given a particular clock cycle number, opcode, addressing mode and vector mode, the control-signal generator will specify exactly what should happen in the CPU. For example, if the accumulator should be loaded with a new value on this clock cycle, the control-signal *A_LOAD* will be asserted high. Most arithmetic operations actually happen in cycle zero. Appendix A displays most of the control signals for cycle zero arithmetic operations.

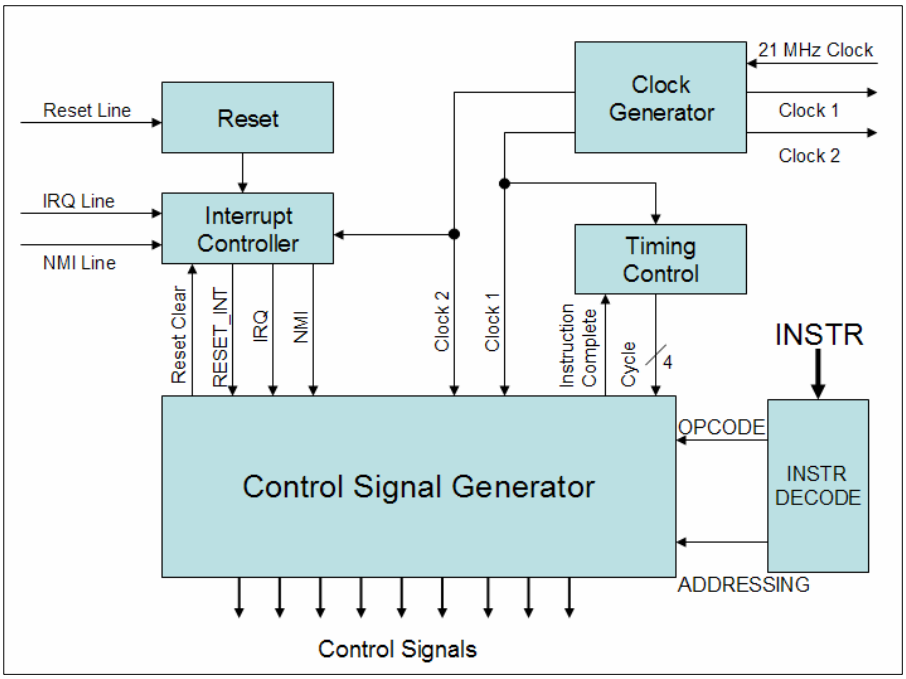


Figure 2 – CPU Front End Internal Architecture

Arithmetic Logic Unit and Operand Selectors

The arithmetic logic unit (ALU) is the module that implements all of the arithmetic operations specified by instructions and needed for particular addressing modes. The ALU has an 8-bit adder with subtraction. It also has a barrel shifter and several bitwise functions. The operation of the ALU is specified by three control-signals. They are: *ALU_FN*, *OPERAND_A*, and *OPERAND_B*. The *ALU_FN* control signal can specify the following operations:

ALU_FN =

FN_A =	Pass operand A
FN_B =	Pass operand B
FN_ADD =	Add operand A and B with carry
FN_SUB =	Subtract operand B from A including borrow
FN_AND =	AND operands A and B
FN_OR =	OR operands A and B
FN_XOR =	Exclusive OR operands A and B
FN_SL =	Shift operand A left and move a zero into bit 0
FN_SR =	Shift operand A right and move a zero into bit 7
FN_RL =	Rotate op. A left by moving C into bit 0 and bit 7 into C
FN_RR =	Rotate op. A right by moving C into bit 7 and bit 0 into C
FN_ADD_NC =	Add operand A and B without the carry bit
FN_SUB_NB =	Subtract operand B from A without the borrow bit
FN_STATUS =	Pass the value of the Status Register

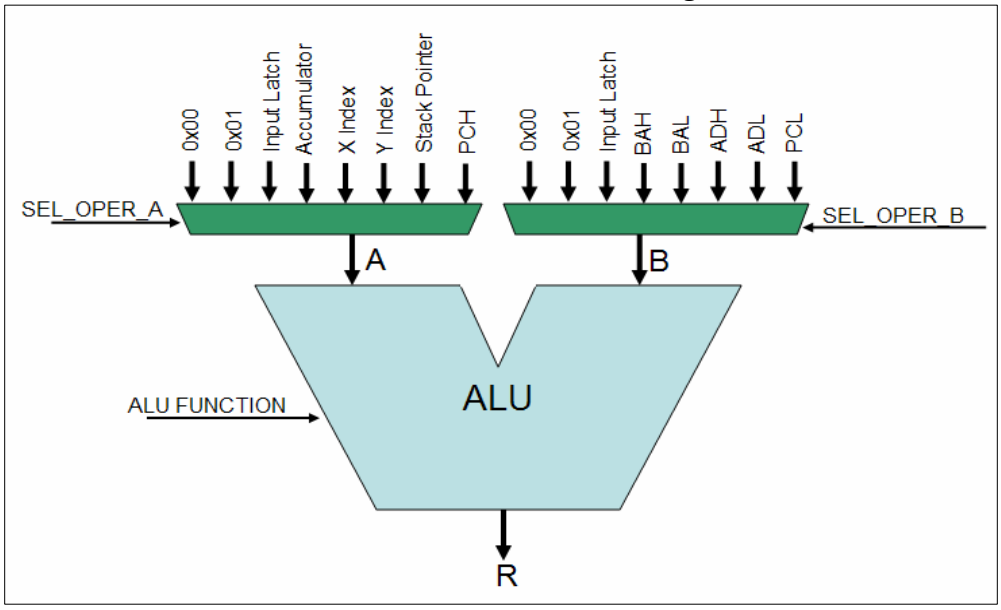


Figure 3 – ALU Architecture

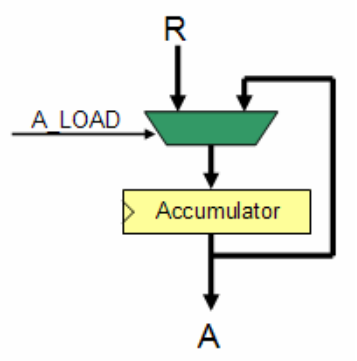
These operations can be done on many combinations of operands. These operands are specified by the *OPERAND_A* and *OPERAND_B* control-signals as follows:

SEL_OPER_A =	
OPERAND_ZERO	The value 0x00
OPERAND_ONE	The value 0x01
OPERAND_MEM	The contents of the input latch
OPERAND_ACCUM	The accumulator
OPERAND_X	The X Index
OPERAND_Y	The Y Index
OPERAND_S	The Stack Pointer
OPERAND_PCH	The upper byte of the Program Counter

SEL_OPER_B =	
OPERAND_ZERO	The value 0x00

OPERAND_ONE	The value 0x01
OPERAND_MEM	The contents of the input latch
OPERAND_ BAH	The upper byte of the Base Address
OPERAND_ BAL	The lower byte of the Base Address
OPERAND_ ADH	The upper byte of the Base Address
OPERAND_ ADL	The lower byte of the Base Address
OPERAND_PCL	The lower byte of the Program Counter

The Accumulator

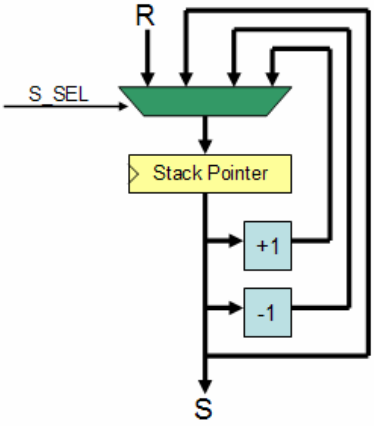


The accumulator is one of the three general purpose register in the 6502 processor. Most all arithmetic operations either use the accumulator as an operand or as the place to store the result. Many operations used the accumulator for both purposes. The accumulator is loaded with a new value when *A_LOAD* is high. Otherwise, the accumulator will hold its value.

Indexing Registers: X and Y

The X and Y index registers are the other two general purpose registers in the 6502. They differ from the accumulator in that an arithmetic operation can not be directly stored in them. The accumulator however can be transferred to either of these registers. Unlike the accumulator, both the X and Y index registers can be incremented or decremented using a one-byte instruction with an implied addressing mode. The control-signals *X_SEL* and *Y_SEL* control whether they hold their values or load a new value from the ALU.

The Stack Pointer



In order to enable efficient interrupts and subroutine calls, the 6502 processor must have a stack. The stack pointer is an 8-bit register that points to the top of the stack in page one of RAM. This register has the special ability to increment and decrement without using the ALU. When pushing a byte onto the stack, the stack pointer decrements. When pulling a byte off the stack, the stack pointer increments. The operation of the stack pointer is controlled by the *S_SEL* control-signal.

Data Output and Input Registers

The data output register is loaded with the ALU result when *LATCH_OUTPUT* is high. This allows the CPU to drive this value onto the data bus in a following clock cycle. To receive data from the data bus, the input register is loaded with the contents of the data bus. This happens when the control-signal *LATCH_INPUT* is high. This value is then used in most arithmetic and storage operations in following clock cycles.

Address Bus Selector

In order to read and write data on the data bus, a valid address must first be supplied on the address bus. This address can be equal to many internal values within the CPU. Depending on the current cycle and addressing mode, the address bus may be selected to any of the following

using AB_SEL. For example, during a stack read or write, the higher address byte will be equal to 0x01 and the lower byte will equal the Stack Pointer. This is represented by: {01,S}. The hardware implementation for this is two very large muxes.

AB_SEL =

AB_PCX:	{PCH,PCL}
AB_ADX:	{ADH,ADL}
AB_BAX:	{BAH,BAL}
AB_AD_ZERO:	{00,ADL}
AB_BA_ZERO:	{00.BAL}
AB_STACK:	{01,S}
AB_NMI_LOW:	{FF,FA}
AB_NMI_HIGH:	{FF,FB}
AB_RESET_LOW:	{FF,FC}
AB_RESET_HIGH:	{FF,FD}
AB_IRQ_LOW:	{FF,FE}
AB_IRQ_HIGH:	{FF,FF}

Addressing Registers: ADH, ADL, BAH and BAL

The 6502 has four 8-bit addressing registers that it uses to control many of the addressing modes. Two of these, specifically ADH and ADL, combine to form the absolute addressing register, and the other two, BAH and BAL, combine to form the base addressing register. Different addressing techniques load data into each of these registers to accomplish different methods of addressing a byte of data. For example, during a load instruction with absolute indexed addressing, the BAH and BAL registers will be loaded with the absolute base address of the byte to be loaded. Then one of the index registers will be added to this base address to obtain the actual address of the byte to be loaded. In addition to these registers, the CPU documentation also suggests that there are registers named IAH and IAL that are for indirect addressing. I chose not to implement them (as I am sure the actual processors also did not) because all of the addressing techniques can be implemented using only ADH, BAH, ADL and BAL. BAH and BAL simply substituted for IAH and IAL when needed. The operation of these registers is done using the *ADX_SEL* and *BAX_SEL* control-signals.

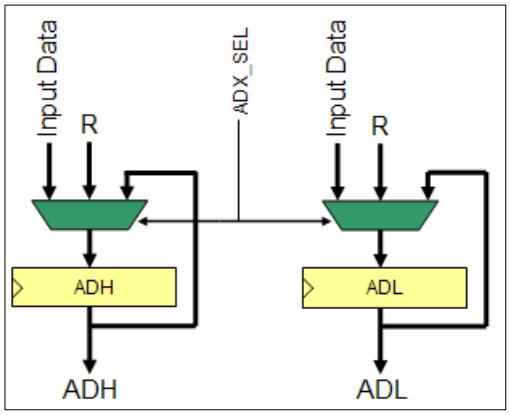


Figure 4 – CPU Addressing Registers

ADX_SEL or BAX_SEL =

ADR_HOLD:	Both Registers Hold Contents
ADR_LATCH_L:	Lower byte gets value of data bus
ADR_LATCH_H:	Upper byte gets value of data bus
ADR_LOAD_L:	Lower Byte gets value of R
ADR_LOAD_H:	Upper Byte gets value of R
ADR_ABS_INDEX	Lower Byte gets R and Upper Byte gets value of data bus

Program Counter

The program counter is a 16-bit register that keeps track of the address of the next instruction to execute. While most instructions increase the value of this program counter to the address of the next instruction, other operations may change the contents of this register to cause a branch or a jump. This is the way that program flow is controlled. The program counter is controlled by the control-signal *PC_SEL* and can either hold its value, increment its value or be loaded with one of several values documented below.

PC_SEL =	
PC_HOLD:	PC holds value
PC_INC:	PC increments value
PC_LATCH_L:	Lower byte gets value of data bus
PC_LATCH_H:	Upper byte gets value of data bus
PC_LOAD_L:	Lower byte gets value from ALU result
PC_LOAD_H:	Lower byte gets value from ALU result
PC_LATCH_ADX	Load both PCH and PCL from ADH and ADL
PC_LATCH_ADXP	Load both PCH and PCL with {ADH,ADL} + 1

Status Register and Control Flags

To aid in the control of the microprocessor, the 6502 has a status register that contains some very important flags. Although these individual flags are all members of the status register, they are most appropriately individual registers each one bit in size. The Status register can only be written to as a whole when pulling the value of the status register off of the stack.

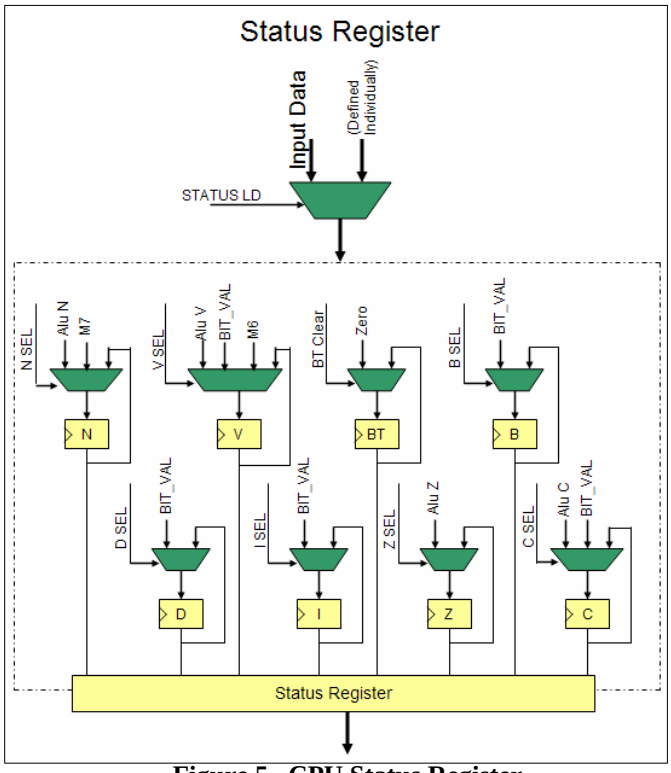


Figure 5 –CPU Status Register

The first flag (the most significant) in the status register is the N flag. This flag is always set when the last arithmetic operation resulted in a negative number (i.e. the most significant bit is a one). This flag can also be changed by load instructions.

The second flag in the status register is the overflow flag or V flag. This flag is set when the last operation resulted in an arithmetic overflow. The exact workings of this are slightly complicated, but it is generally set when the result of an operation does not make sense. For example: the overflow flag is set when a positive plus a positive results in a negative number. This flag can be cleared in software.

The next flag in the status register is a custom flag that was added by our team. This flag is called the Boot flag (BT). On reset, this flag is automatically set and remains set until a custom

instruction, specifically EBT or 0xff, is run. This instruction clears the BT bit and causes a software reset. The purpose of this bit is to allow for custom boot code to run on the 6502 before it begins to load the cartridge. With this ability, the extensions of the system would be almost limitless. During boot mode, the cartridge program ROM is disabled and instructions are instead fetched from a boot ROM inside of the FPGA.

The fourth flag in the status register is the Break flag (B). This flag is set when a break command is executed and cleared when an interrupt occurs. This bit is needed because the break instruction causes the program to jump to the interrupt vector as if an interrupt had occurred. By checking the break flag, the software can check to see if the source of the interrupt was in software or hardware.

The Decimal mode flag (D) follows the break flag. This flag is used to enable or disable decimal mode in some 6502s. This ability is not implemented in the Nintendo's 6502.

Next, the Interrupt Disable flag (I) is used to disable interrupts caused by the IRQ line. These interrupts are ignored if the I flag is high. This flag can be both set and cleared in software.

The Zero flag (Z) is set whenever the most recent arithmetic operation resulted in zero. This includes loading of some registers and transfers.

Finally, the last flag in the status register is the Carry flag (C). This flag is used by many operations. It is used it rotations. It is the 9^{th} of the result of an add operation. The not of the carry flag is also the borrow bit and signifies whether or not a borrow has occurred in a subtraction operation.

Added Features to the CPU

Boot Mode

The boot mode modification was documented briefly in coverage of the status register. This modification allows for the CPU to execute code without a cartridge and before the cartridge is loaded. It is meant for three things. First, the boot mode allowed for easy and quick testing of program code for the 6502. Second, the boot mode is intended to allow for a custom screen to appear on the screen when the NES system is turned on. This will happen briefly before the game is loaded. Finally, the boot mode will allow for extensions to the NES system such as access to a hard drive to load games from.

Custom Output Registers

Two custom registers have been added to the CPU at addresses 0x4018 and 0x4019. These are write only registers. Register 0x4018 is an eight bit register and 0x4019 is a three bit register. Together these registers directly control eleven extra processor output pins. This allows for debugging and the game that is implemented in the following section.

Custom Timer Register

Debugging the CPU without the rest of the Nintendo system has been very important. To keep track of timing without the NMI line from the PPU, a custom timer has been implemented at register 0x4020. When the least significant bit of this register is zero, the timer is disabled. Otherwise the timer generates an IRQ interrupt at certain rate and holds the IRQ line low until the register is reset. The rate of this interrupt is (127 / x) * 60 Hz where x is the contents of the seven most significant bits in the register. This custom register is used to generate timing in the following section.

Demonstration: The 6502 in Action

Demonstrating the functionality of the CPU aside from the PPU was very important. This would allow for much easier debugging of the PPU if we could show that the CPU worked correctly by itself. In order to do this, we wrote a game in 6502 assembly to demonstrate the complete functionality of the CPU.

Hardware Setup: Scanning LEDs on the Output Port

Because the PPU would not be involved in this testing, an alternate output had to be provided from the CPU. This was done using the custom output port. Nine red and nine green leds were connected to the pins of the processor in a 3x3x2 matrix configuration. Three wires were used at row select lines, three wires were used to light the three red leds in a row, and three wires were used to light the three green leds in a given row. This setup allowed for a total control of eighteen leds using only nine control lines.

In order to control the leds and make them all appear properly lit, the array of leds had to be scanned through faster than the human eye could see. To do this, every 5.6 ms one row of the matrix had to be selected by bringing its line low. The other two row-select lines were brought high. This allowed for the six outputs on the other control lines to properly light though six leds for 5.6 ms. After that, a different set of six leds would be lit. This was done repeatedly so make all eighteen leds appear to be either on or off when in fact they were all flashing.

Gameplay: Super Steal'em Tic-Tac-Toe

Super Steal'em Tic-Tac-Toe is a variation of the popular tic-tac-toe game. We created this variation to demonstrate the 6502. The timer was used to cause an interrupt every 5.6ms and update the led matrix. The NES controllers were read from at an update rate of 60 Hz. This input information was used to update variables in the RAM and control the game play.

Each player (the game uses two controllers) takes turns selecting a space to place their mark. Each player's mark is represented by a color. To select a mark, they each move a cursor which is a blinking led of their color. Once a mark is placed, it may be stolen by the opponent. The objective of the game is familiar: Place three marks in a row to win the game. The game play is new because you opponent can foil your plans. The additional rules to the game are that you may not steal your opponent's most recent move, and you may not make more than three steals per game.

The source code for this game is included in Appendix G. Have fun reading the 6502 assembly.

Nintendo Picture Processing Unit

The Nintendo Picture Processing Unit (PPU) processes all graphical data and outputs it to the video display. It utilizes high level information controlled by the CPU, such as sprite location and background tile indexes, to process any given frame. The PPU is clocked three times faster than the CPU, which underscores its function as a redundant but busy processing module.

PPU architecture

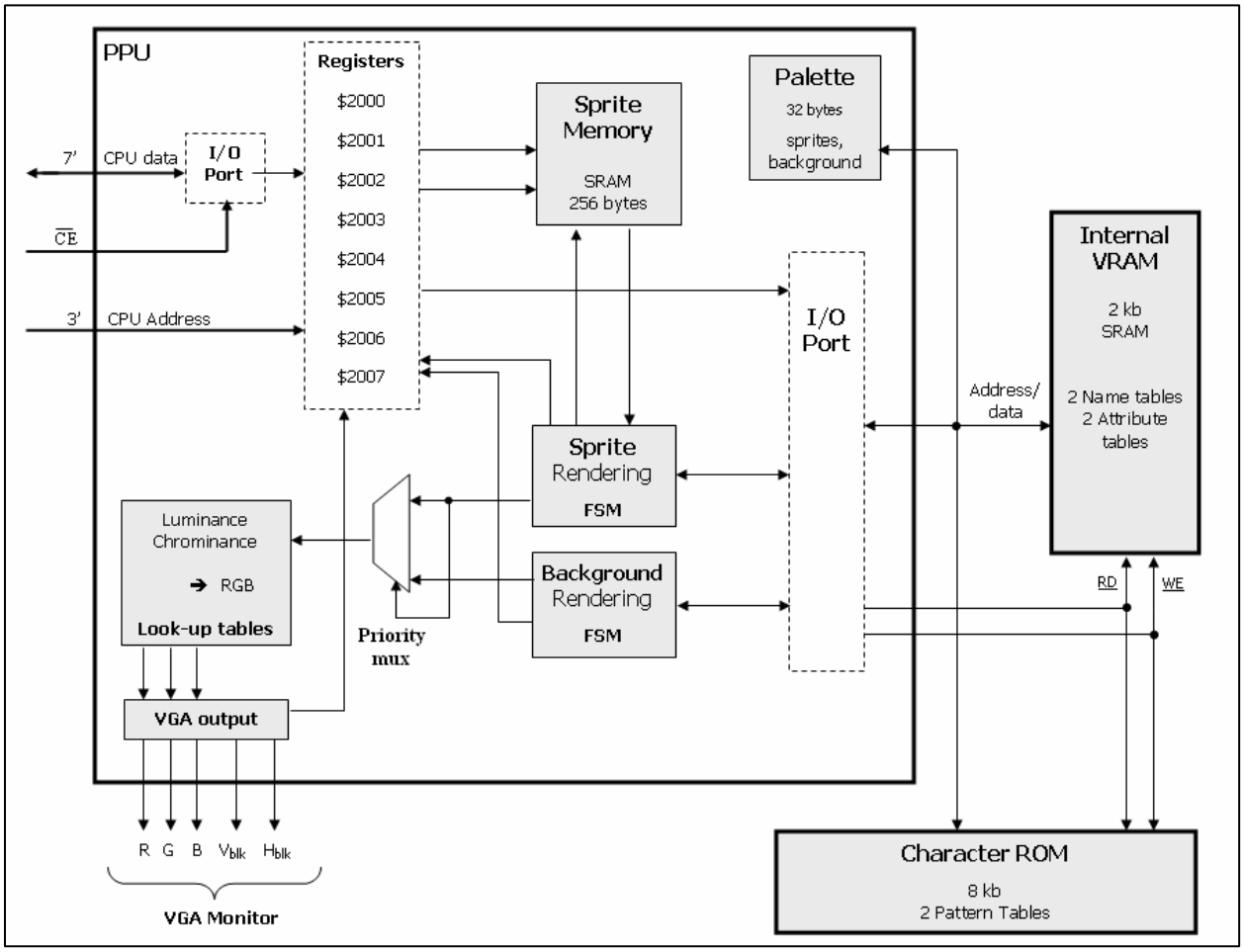


Figure 6 – PPU layout and module connections

The PPU has several main sections, which can be explained in turn. Figure 6 shows the layout and connection of the different sections.

Registers

The PPU's functions are controlled through eight main registers that are accessed by the CPU's address and data lines. Some of these registers hold information that PPU elements use, such as whether sprites should be displayed. Other registers are used to inform the CPU about what has been rendered on the screen. Finally, several registers are used for memory writes, so that the CPU can give the PPU its needed high-level graphics information.

Below is a short list of each register's functionality, courtesy of Jeremy Chadwick. This documentation was found on nesdev.pariodius.com, a website for Nintendo engineering enthusiasts.

```
+---------+-------------------------------------------------------+
| Address | Description                                           |
+---------+-------------------------------------------------------+
|  $2000  | PPU Control Register #1 (W)                           |
|         |   %vMsbpiNN                                           |
|         |                v = Execute NMI on VBlank              |
|         |                       1 = Enabled                     |
|         |                M = PPU Selection (unused)             |
|         |                       0 = Master                      |
|         |                       1 = Slave                       |
|         |                s = Sprite Size                        |
|         |                       0 = 8x8                         |
|         |                       1 = 8x16                        |
|         |                b = Background Pattern Table Address   |
|         |                       0 = $0000 (VRAM)                |
|         |                       1 = $1000 (VRAM)                |
|         |                p = Sprite Pattern Table Address       |
|         |                       0 = $0000 (VRAM)                |
|         |                       1 = $1000 (VRAM)                |
|         |                i = PPU Address Increment              |
|         |                       0 = Increment by 1              |
|         |                       1 = Increment by 32             |
|         |              NN = Name Table Address                  |
|         |                     00 = $2000 (VRAM)                 |
|         |                     01 = $2400 (VRAM)                 |
|         |                     10 = $2800 (VRAM)                 |
|         |                     11 = $2C00 (VRAM)                 |
|         |                                                       |
|         | NOTE: Bit #6 (M) has no use, as there is only one (1) |
|         |       PPU installed in all forms of the NES and Famicom. |
+---------+-------------------------------------------------------+
|  $2001  | PPU Control Register #2 (W)                           |
|         |   %fffpcsit                                           |
|         |             fff = Full Background Colour              |
|         |                     000 = Black                       |
|         |                     001 = Red                         |
|         |                     010 = Blue                        |
|         |                     100 = Green                       |
|         |                p = Sprite Visibility                  |
|         |                       1 = Display                     |
|         |                c = Background Visibility              |
|         |                       1 = Display                     |
|         |                s = Sprite Clipping                    |
|         |                       0 = Sprites not displayed in left |
|         |                           8-pixel column              |
|         |                       1 = No clipping                 |
|         |                i = Background Clipping                |
|         |                       0 = Background not displayed in |
|         |                           left 8-pixel column         |
|         |                       1 = No clipping                 |
|         |                t = Display Type                       |
```

		0 = Colour display
		1 = Mono-type (B&W) display
$2002	PPU Status Register (R)	
	%vhsw----	
		v = VBlank Occurance
		1 = In VBlank
		h = Sprite #0 Occurance
		1 = VBlank has hit Sprite #0
		s = Scanline Sprite Count
		0 = Less than 8 sprites on the current scanline
		1 = More than 8 sprites on the current scanline
		w = VRAM Write Flag
		1 = Writes to VRAM are ignored
$2003	SPR-RAM Address Register (W)	
	Specifies the 8-bit address in SPR-RAM to access via $2004.	
$2004	SPR-RAM I/O Register (RW)	
	Used to R/W 8-bit data to/from SPR-RAM.	
$2005	Background Scroll Register (W2)	
	Used to pan/scroll the pre-rendered screen (excluding sprites) horizontally and vertically. The data is written in the following manner:	
	BYTE 1: Horizontal BYTE 2: Vertical	
$2006	VRAM Address Register (W2)	
	Specifies the 16-bit address in VRAM to access via $2007. The data is written in the following manner:	
	BYTE 1: Upper 8-bit portion of effective address BYTE 2: Lower 8-bit portion of effective address	
$2007	VRAM I/O Register (RW)	
	Used to R/W 8-bit data to/from VRAM.	

Figure 7 – PPU Register Functionality

Graphics Rendering

The PPU has two main modules for graphics rendering, the background rendering and sprite rendering modules. These two modules work in parallel to produce their respective graphics, which are later prioritized on a per-pixel basis to produce the final image. The chosen pixel is given color information by the priority multiplexer, and this data is sent to the monitor.

A high-level block diagram of the multiplexer is shown in Figure 8.

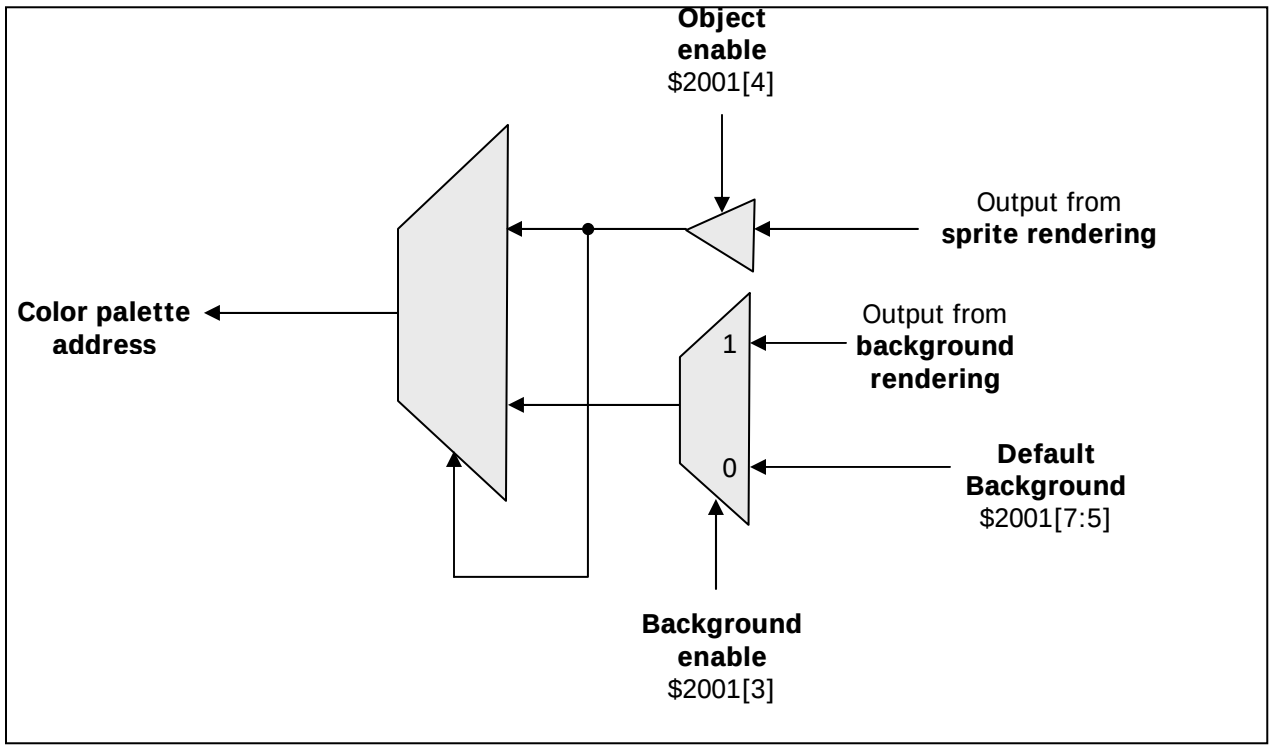


Figure 8 – A high-level block diagram of the priority multiplexer

Further information about the background and sprite rendering modules is given later in a and separate section.

I/O control

The PPU memory space consists of multiple different memory chips, and the PPU needs to ensure that they are accessed without conflicts. To reduce the number of pins needed, Nintendo designed the PPU to have a shared bidirectional data bus with the 2kb VRAM and the 8+ kb character ROM in the cartridge. What's more, these data lines are used for the lower address lines for both chips, so it is crucial that memory access timing is well defined and precise.

To ensure that timing for the data lines was accurate, the I/O control was clocked at the full 21.47 MHz, instead of the PPU's pixel rendering rate of 5.37 MHz. The same was done with the CPU register access, to ensure that registers that needed to read/write to PPU memory space could do so with the correct timing.

The most important control signals in the I/O control are threefold:

- ALE: The ALE (address latch enable) determines if the data on the data lines should be latched to the address input of the two memories.
- /RD, /WE: These are standard signals that determine whether a module is reading or writing from memory. Since multiple modules need to read from memory, multiple modules have control of this signal.

- PA13: The highest-order address line is used to select between memory chips. The signal is inverted before it reaches the /CS (chip select) input of the internal VRAM, and is sent directly to the /CS of cartridge character ROM. This ensures that only one of the chips can control the data lines at any one point in time.

Internal Memory

The PPU contains several regions of internal memory. Having internal memory helps the PPU process information faster, since its multi-purpose external data bus cannot handle everything. The two main internal memories are the color palette memory and the sprite memory. The sprite memory contains information about 64 sprites, the limitation for how many sprites can be in any given frame. The color palette memory is accessed during pixel output, to determine how the pattern information from sprite and background modules should be colored.

VGA Output Module

Once color data emerges from the priority multiplexer, it must be converted from Nintendo colors to VGA. There are 64 colors in the palette, and each can be represented as a combination of red, green, and blue levels.

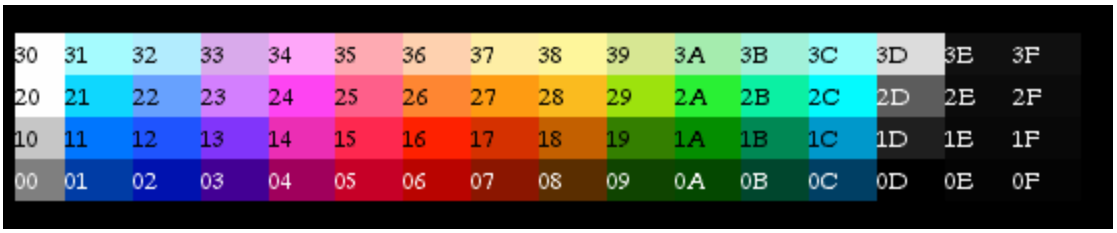


Figure 9 – Nintendo Color Palette

Three lookup tables, one for each color, are established to generate this output. They contain 6-bit red, green, and blue levels, which are sent to 18 pins of the FPGA.

A resistor digital to analog converter network converts these outputs into three signals between 0 and 0.7 volts, fed into the R,G,B pins of the monitor connector. The other critical signals are hsync and vsync, which establish the proper monitor refresh rate. Since the Nintendo did not originally produce VGA output, its clock is not optimized for the correct timing. As a result, we had to experiment with different timing setting and methods to achieve correct operation.

Another challenge was accommodating the 480 lines native to VGA displays. Fortunately, this is exactly twice the vertical resolution of the Nintendo, so a two line output buffer was employed for this task. To achieve the horizontal resolution of 256 pixels, with the pre-defined 21.47727 MHz clock on the Nintendo, a border was placed on both sides of the data before final output.

Sprite Rendering

Overview

Sprites provide the Nintendo with the most flexible graphics capabilities. Each sprite consists of a small custom-colored pattern, placed in an exact location on the screen, and with attributes like vertical/horizontal flip and foreground/background priority. Because of these features and flexibility, sprites are used for almost all game animations and characters.

The PPU has enough internal memory for 64 sprites to be displayed in any given frame. This high-level memory is constructed by the CPU during vertical blank, and is then used by the PPU during pixel output to produce the necessary graphics. Each sprite needs four bytes of data. The schematic below shows the layout of the sprite memory, courtesy of Marat Fayzullin (documentation found on nesdev.parodius.com)

```
Sprite Attribute RAM:
| Sprite#0 | Sprite#1 | ... | Sprite#62 | Sprite#63 |
     |          |
     +---- 4 bytes: 0: Y position of the left-top corner - 1
                    1: Sprite pattern number
                    2: Color and attributes:
                       bits 1,0: two upper bits of color
                       bits 2,3,4: Unknown (???)
                       bit 5: if 1, display sprite behind background
                       bit 6: if 1, flip sprite horizontally
                       bit 7: if 1, flip sprite vertically
                    3: X position of the left-top corner
```

The Nintendo has one important limitation with its sprites. Due to time constraints with memory access, only 8 sprites can be shown per scanline. This requires game developers to either limit their sprite number or use background tiles for some of their animations. More about this limitation is explained later. A high-level block diagram of the entire sprite renderer is shown in Figure 11.

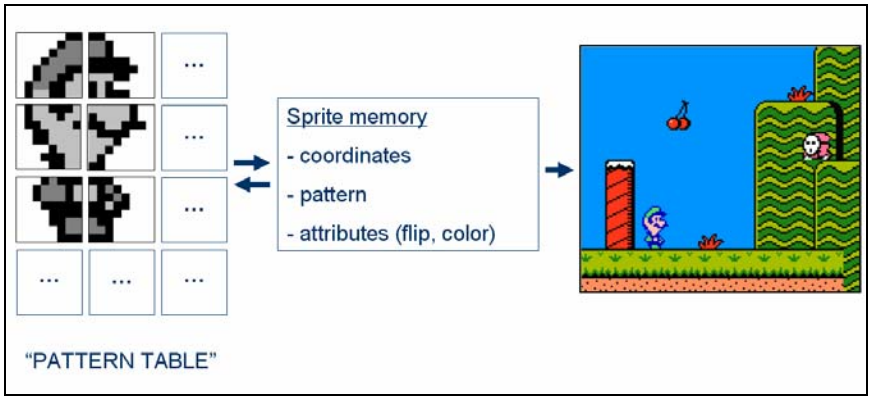


Figure 10 – Sprite Rendering Visualization

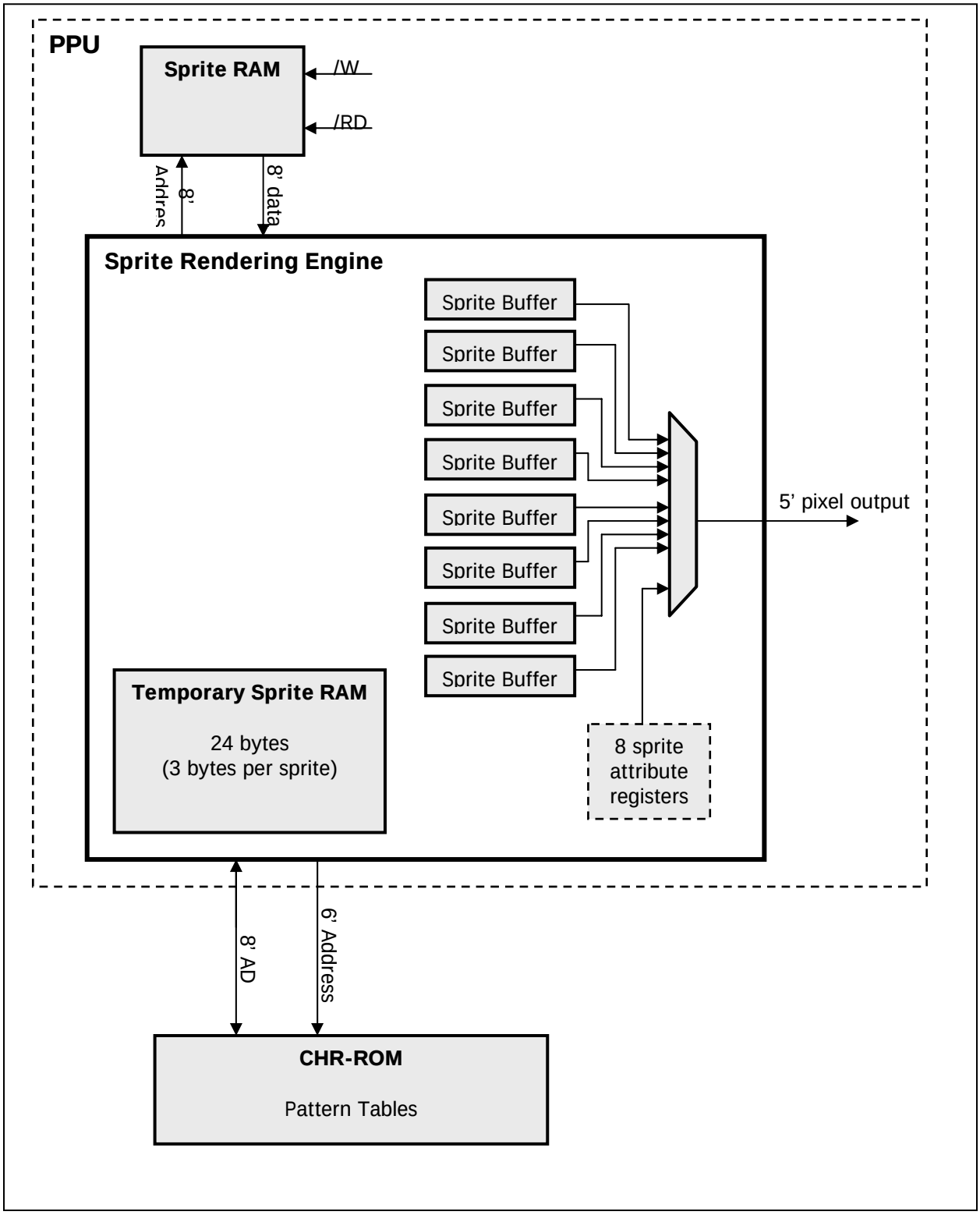


Figure 11 – High-level block diagram of the sprite renderer

Rendering stages

Sprite rendering happens in three discrete stages: an “in-range evaluation” phase, a “memory fetch” phase, and a “real-time output” phase. There are three stages because memory access is limited, and because some processes need to be handled in parallel.

The timing of these stages is as follows:

scanline pixel index	sprite rendering phase
0..128	in-range evaluation
257…320	memory fetch
0..256	real-time output

Table 3 – Timing of Rendering Stages

Range Evaluation

The range evaluation stage happens during pixel rendering. The PPU scans through the y-coordinates of each element in its internal sprite RAM, compares that value to the current scanline number, and hence determines whether this sprite should be displayed on the *next* scanline.

If a sprite is found to be in range for the next scanline, the PPU will take note of it. In our implementation, we simply stored the sprite’s data to a temporary sprite RAM that was accessed later, but it is just as easy to store the sprite’s index number in the 64-element memory. Figure 12 shows a detailed state transition diagram for this stage.

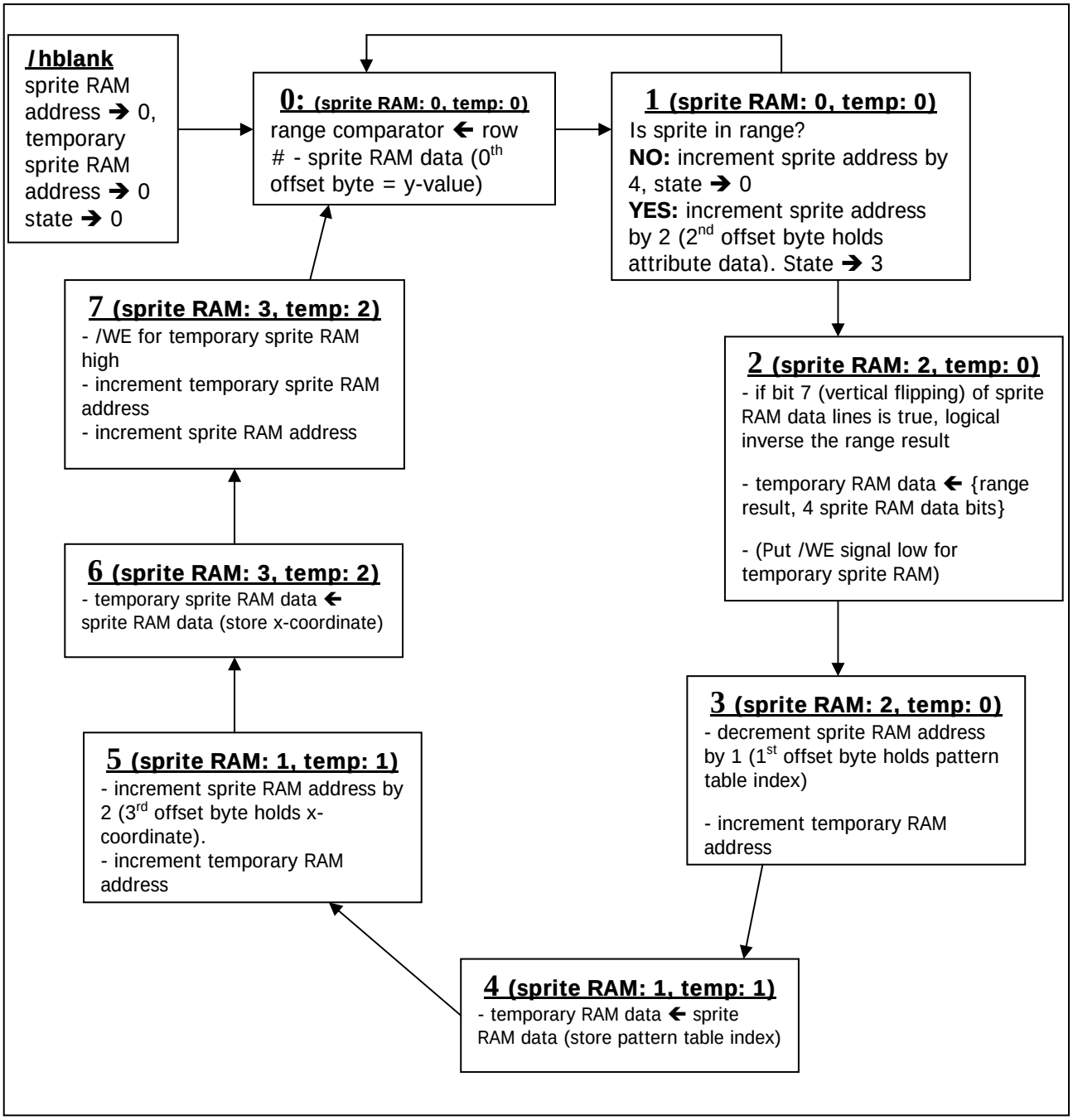


Figure 1 – Detailed state transition diagram for Sprite Rendering

Memory Fetch

During the first part of horizontal blanking, the background renderer no longer needs to access external memory. Thus, the sprite renderer has a small amount of time during which it needs to fetch the pattern data for each sprite that was found to be in range.

The time limitation is so severe that the Nintendo can only fetch information for eight sprites during this time. This means that if a scanline has more than eight sprites, some of them will simply not be displayed. To allow game developers to handle the instances when more than eight sprites exist on a scanline, the PPU raises a flag in the "in-range evaluation" phase if more than 8 sprites are found. It then ignores the rest of the sprites. This flag is stored in register 2002, which can be accessed by the CPU during vertical blanking. Most games cause the CPU to switch the location of the sprites in memory, so that some sprites are visible in every other frame while others are cut out. This creates the graphics blinking that is seen in many complicated Nintendo games. The output is slightly messy, but at least all the characters can be seen.

As pattern information is received from memory, the PPU constructs "sprite buffers" for the "real-time output" phase. Each buffer will contain pattern data for the sprite, color attributes, priority information, and an exact horizontal coordinate. Constructing the sprite buffers in a separate region of memory is necessary so that the real-time output phase does not have memory conflicts with the "in-range evaluation" phase, both of which must fetch sprite data in parallel.

Figure 13 shows a detailed state transition diagram for this stage.

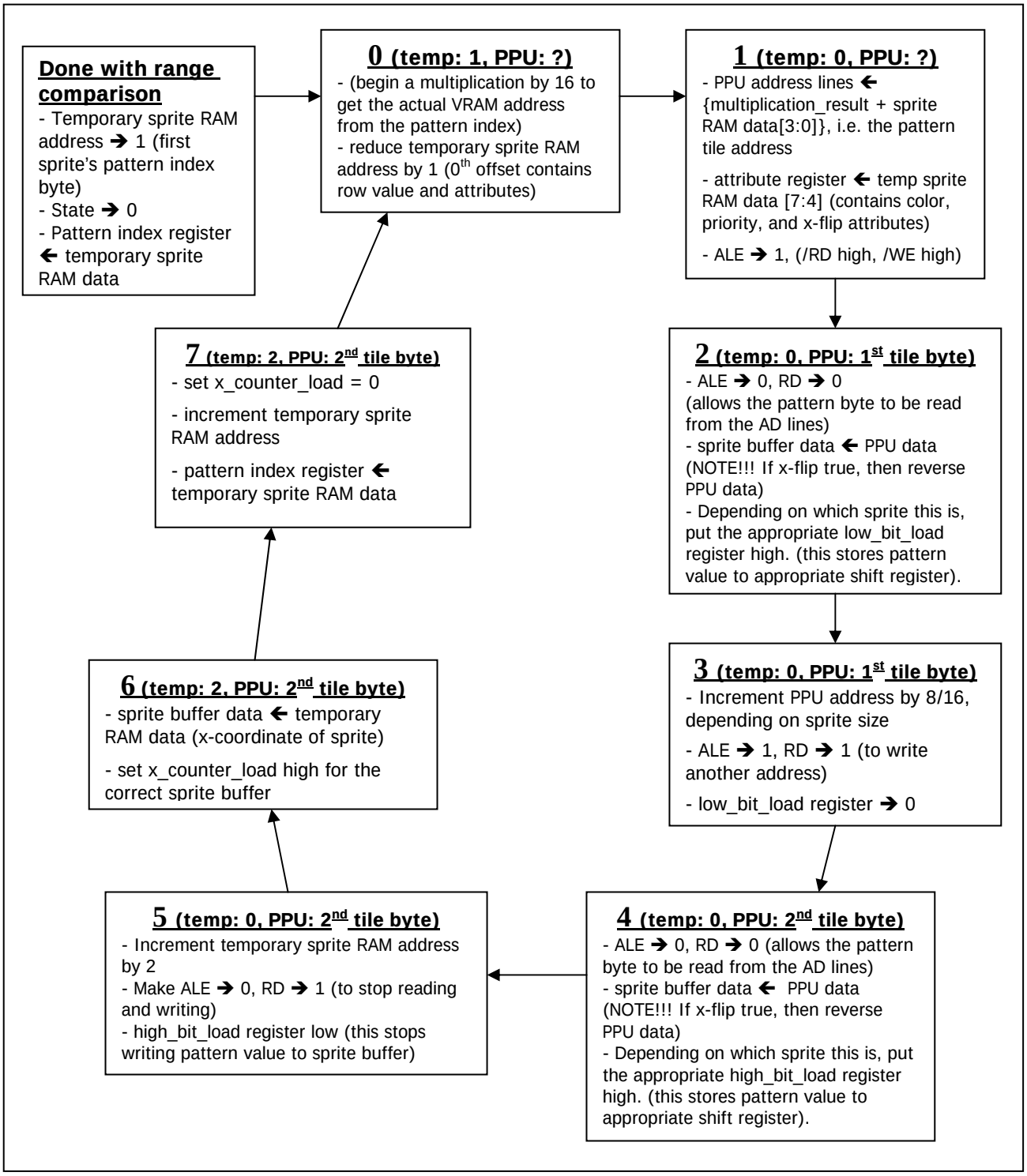


Figure 2 – Detailed state transition diagram of the memory fetch stage

Real-time Output

The real-time output phase uses the "sprite buffer" modules to output pixels when needed. Inside a sprite buffer (there are eight copies of it) are two shift registers, an attribute register, and a horizontal coordinate counter. The coordinate counter keeps track of whether the sprite should be displayed, and if it should, it will clock the two shift registers so that they output data.

Signals from all eight sprite buffers are combined with specific priority, and one pixel signal is then sent out from the sprite renderer. The priority is determined by a simple rule: the earlier sprites in memory have a preference, and if any sprite is invisible for a given pixel, pass priority to the next sprite in line. Thus sprite "0" (the first sprite in memory) will always be displayed, and the other sprites can only show underneath it some of its pixels are invisible.

A block diagram of a sprite buffer is given in Figure 14.

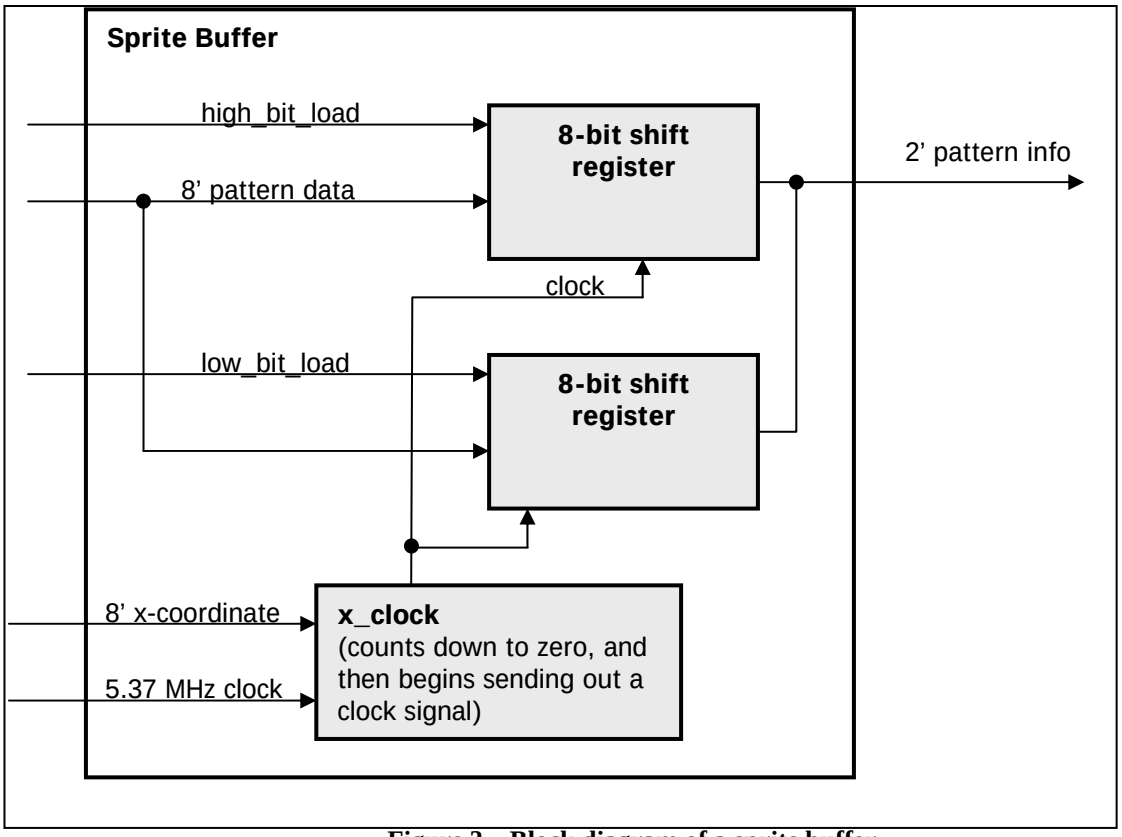


Figure 3 – Block diagram of a sprite buffer

Background Rendering

Overview

Inside the Picture Processing Unit (PPU), a major subsection is the background rendering module, which is also referred to as “playfield rendering” by some Nintendo documentation. This device constructs an image from the 8 pixel by 8 pixel tiles stored on the character ROM inside the cartridge and passes output to the priority multiplexer for combination with sprite information

The background is rendered in near-real time before display on the screen. The entire screen is rewritten at approximately 60 times per second, and each of these frames is divided into scanlines, representing the amount of time it takes for the display to write a line of pixels on the screen. The fist twenty of these scanlines are called the Vertical Blank period, when no data is being output and the vertical synchronization signal is sent to the screen.

During the twenty-first scanline, immediately before feeding actual image data, the background renderer fetches the first two eight pixel tiles present on the first scanline, capturing one of the eight rows for each tile that corresponds to this position. As soon as this data starts displaying on the screen, the background renderer begins to retrieve the third eight pixel section of the first row, and this process requires the same amount of time as it takes to display eight pixels. The result is that the background renderer is usually starts fetching background data exactly sixteen pixels before it appears on the screen.

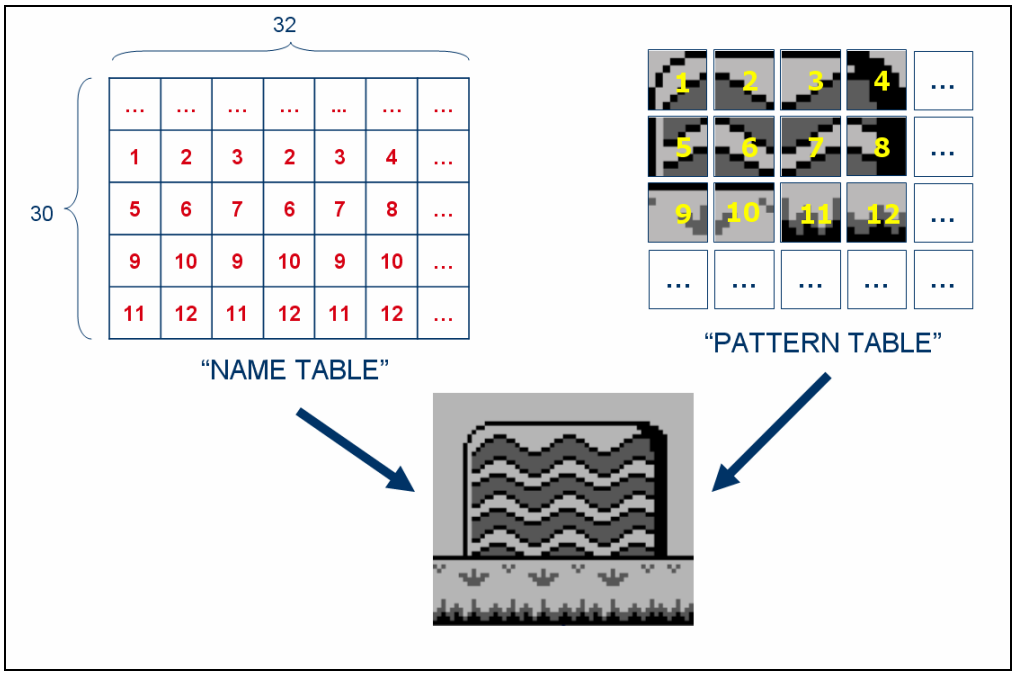


Figure 4 – Background Rendering Visualization

Memory Fetch Cycle

Construction of a complete background tile requires four fetches inside the memory space available to the PPU circuitry, and each of these fetches takes two cycles of a 5.36 MHz clock, which is the same clock incrementing the display of pixels on the screen. At the beginning of the first clock cycle, the address of the desired data is latched in a register that is connected to the address lines of the various memory storage units. The address latches are enabled, which stores this address in a set of registers external to the PPU and allows data to arrive on the same set of wires. The output enable line to the cartridge and VRAM also external to the PPU is disabled. On the next rising clock edge, the address latch enable is deactivated, setting the value of these registers. The output enable line is simultaneously activated, allowing data to propagate through the memory storage elements and appear on the address/data bus connected with the PPU.

On the following rising clock edge, which is the conclusion of the first fetch cycle and the beginning of the second fetch cycle, the data value is latched into a temporary register, according to which piece of information it represents. At the same time, the address is for the next fetch, the address latch is disabled, and the output enable line is deactivated. In some cases the resulting data determines the address of subsequent fetch operations, and in other cases it maps directly to the output produced by the background renderer. The four particular fetch results are described below.

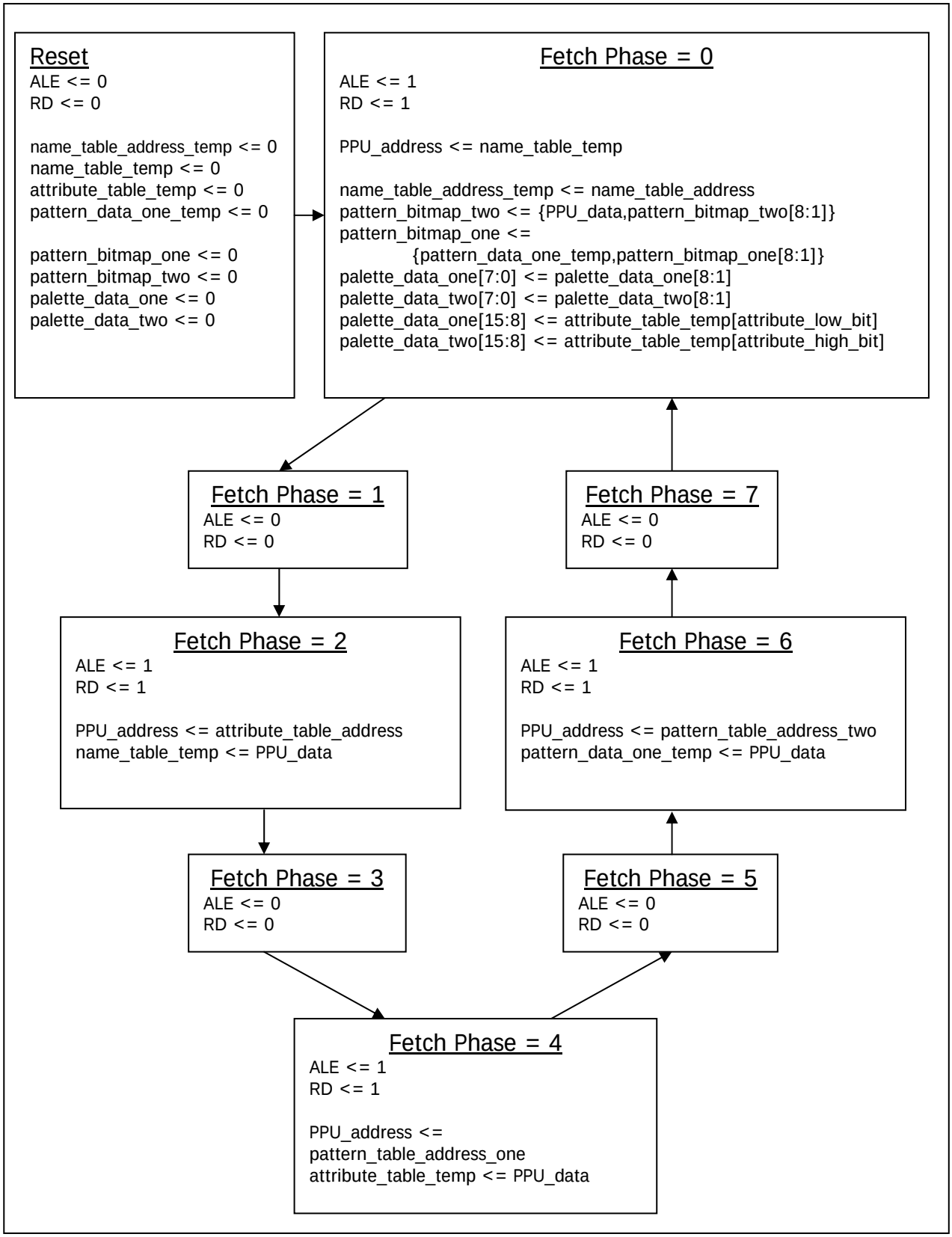


Figure 1 – Background Rendering State Transition Diagram

Cycle One: The Name Table

The Name Table is the first memory storage element accessed by the picture processing unit. It contains references to locations within the character ROM-based pattern tables, which hold partial information for all the eight pixel by eight pixel tiles available to the given Nintendo game. The address of the name table data is determined as follows:

```
assign name_table = y_scroll_nametable_select ?
                        (x_scroll_nametable_select ? 4'b1011 : 4'b1010) :
                        (x_scroll_nametable_select ? 4'b1001 : 4'b1000);

assign name_table_address = {name_table, y_scroll[7:3], x_scroll[7:3]};
```

The first assign statement selects between the four name tables within the memory space by determining the upper four bits of the fourteen bit memory address. Each name table is uniquely identified by a single bit x and y coordinate, allowing for four name tables. Within a given name table, the x and y scroll values determine the exact placement of the particular tile. The upper five bits of each scroll value reference the particular tile, and the lower three scroll bits reference a particular one of the eight lines within the tile, which is not needed until leader in the rendering process. The tiles are positioned in the name table sequentially moving across and then down the rows. Therefore, the upper five bits of the x scroll value are the lowest five bits of the name table address, referencing one of the thirty-two pattern tiles on that line. The next five bits of the address are the upper five bits of the y scroll value, counting from 00000 to 11101, so they address one of the thirty possible lines within the selected name table.

Cycle Two: The Attribute Table

After the name table data is fetched, the attribute table information for the tile is accessed. A single attribute byte applies to sixteen different tiles, so the entirety of each of the four name tables is 64 bytes, filling the name space as just two scanlines of name table data. The following is a diagram from the “NES Documentation” by yoshi@parodius.com that illustrates how name a single attribute byte corresponds to sixteen tiles in the name table.

```
+------------+------------+
|  Square 0  |  Square 1  |  #0-F represent 8x8 tiles
|   #0  #1   |   #4  #5   |
|   #2  #3   |   #6  #7   |  Square [x] represents
+------------+------------+   four (4) 8x8 tiles
|  Square 2  |  Square 3  |  (i.e. a 16x16 pixel
|   #8  #9   |   #C  #D   |        grid)
|   #A  #B   |   #E  #F   |
+------------+------------+
```

Figure 17– Attribute Table Diagram

Notice that the tile labeled “#2” is actually 32 bytes above the tile labeled “#0”, and the tile labeled “#4” is actually the adjacent byte in the name table to the tile labeled “#1”. As a result of this grouping scheme, the attribute byte address is calculated as follows:

```
assign attribute_table = y_scroll_nametable_select ?
                (x_scroll_nametable_select ? 8'b10111111 : 8'b10101111) :
                (x_scroll_nametable_select ? 8'b10011111 : 8'b10001111);

assign attribute_table_address = {attribute_table, y_scroll[7:5], x_scroll[7:5]};
```

As with name tables, there are four attribute tables, and the two binary name table select values pick one of these and set the upper eight bits of the fourteen bit address. Since the attribute byte “tiles” are effectively 32 pixels by 32 pixels, only the upper three bits of the x and y scroll values are necessary to select the particular byte.

The Nintendo then implements a creative scheme for associating the attribute byte bits with the sixteen tiles. Only two bits are needed to set the attributes of a given tile, so an attribute byte can contain 4 unique bit combinations. This diagram, also from the “NES Documentation” by yoshi@parodius.com, illustrates how an attribute table byte is mapped to the sixteen tiles with which it is associated.

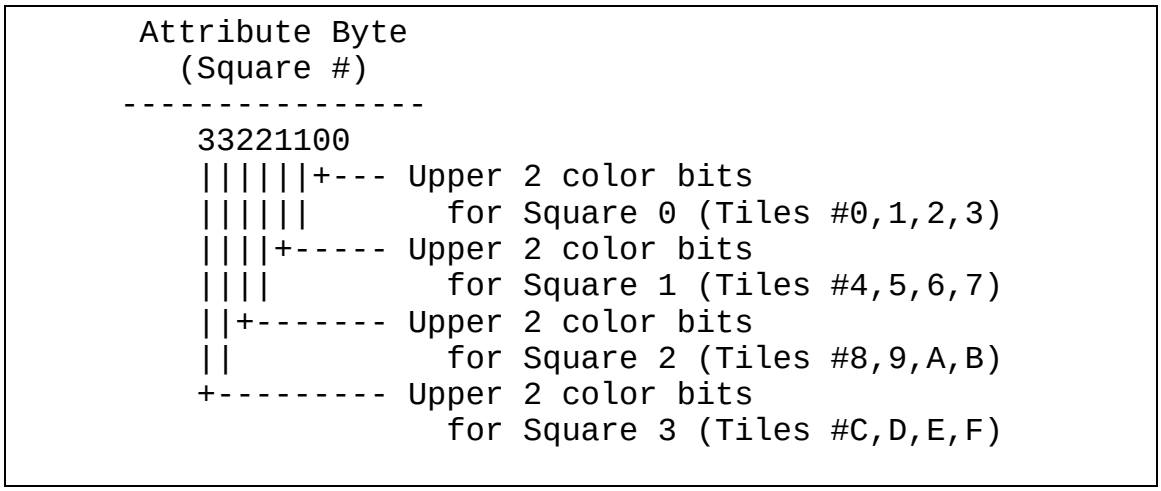


Figure 18 – Attribute Byte Diagram

Cycle Three: The Pattern Table Lower Bit

The third memory fetch retrieves address zero bits from the pattern tables, since the pattern tables stores the pattern bits for a given pixel in separate bytes. The eight bits of a single byte map directly to the eight pixels of a single line of a pattern tile. This means that a single tile occupies sixteen bytes in the pattern tables on the character ROM, and the address is specified as follows:

```
assign pattern_table_address_one =
    {1'b0, playfield_pattern_table_select, name_table_temp, 1'b0, y_scroll[2:0]};
```

The pattern table occupies the lower half of PPU memory space, so the most significant address bit is always zero. There are two pattern tables within this memory space, one in addresses $0000 to $0FFF and the second in addresses $1000 to $1FFF. A register value passed to the background rending module by the main PPU controller selects between these two pattern tables. This is the second most significant bit, referenced as “playfield_pattern_table_select” in the assignment. The next eight bits are the data retrieved during from the name table during the first memory fetch cycle. This corresponds to the starting address of the tile data within the pattern table. Since this is a fetch of the lower pattern bit, the next address bit is zero, corresponding to the first eight tile bytes. Finally, the Nintendo selects the actual byte corresponding to a line of the eight line tile. The lowest three bits of the y scroll value provide this information to finish off the address declaration.

Cycle Four: The Pattern Table Upper Bit

The second pattern table bit is located eight bytes away from this first one. It is fetched in the same manner, and it is the fourth bit needed (along with the two attribute bits and the other pattern table bit) to select one of the sixteen colors in the current palette.

```
assign pattern_table_address_two =
    {1'b0, playfield_pattern_table_select, name_table_temp, 1'b1, y_scroll[2:0]};
```

Notice that the only difference between this address and the one for fetch three is the fourth bit is one instead of zero. Below is a graphical illustration, again from the “NES Documentation” by yoshi@parodius.com, showing how sixteen pattern table bytes correspond to two bits of information per pixel for the 64 pixels of a given tile.

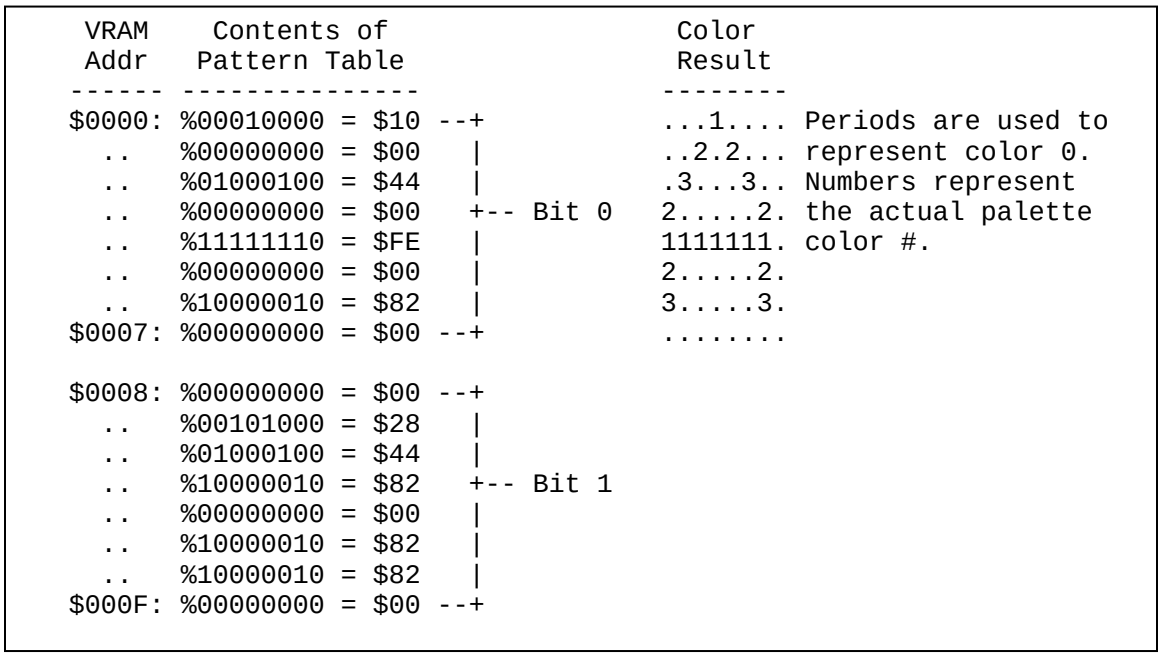


Figure 19 – Pattern Table Diagram

Real-Time Output

At the conclusion of this four fetch sequence, the four resulting bytes are loaded into the upper half of four sixteen bit registers. This is what allows the background generator to work ahead by two tiles, as described above. Each pixel clock cycle, the registers are shifted one bit to the right, which greatly simplified the assignment of final background renderer output. The lowest three bits of the x scroll value determine the alignment of the eight pixel tiles with regard to the edge of the screen, so it determines which of the sixteen register bits is selected for background output. However, since the register components are shifted along with the pixel clock, the same register bit is selected until the x scroll value changes.

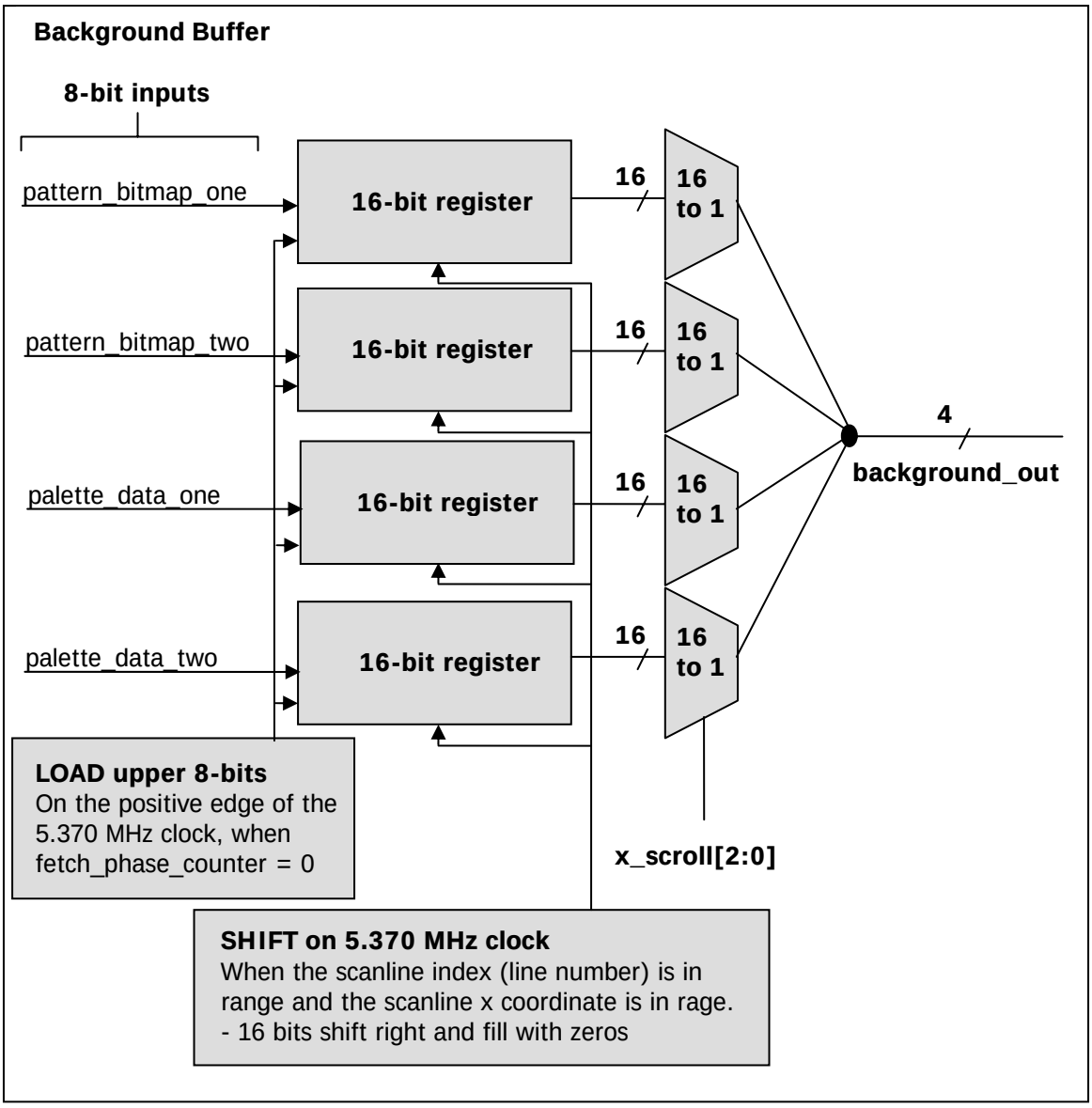


Figure 20 – Background Buffer Block Diagram

Testing and Debugging

Outside of the initial research, this proved to be the most involved and time consuming aspect of our project. Debugging was particularly hard because of incomplete and unreliable information about the system. This was particularly true for the picture processing unit, since it was developed in-house and Nintendo never released details or technical specifications. Even the most complete reverse engineering documents we had access to often provided conflicting information. For example, we could not confirm the function of all the registers inside the PPU.

Testing was performed by a combination of Verilog simulation on the computer, examining actual output on the logic analyzer, and observing the performance of test cases on the monitor and other user interface devices.

Full simulation of the Verilog was critical to establish any chance of the system performing as expected. In developing the CPU, we rigorously tested every sub-module before integrating it into the larger system to identify potential problems and programming mistakes. Due to careful and meticulous diagramming of the system before programming, there were few CPU problems identified by the simulation.

Simulating the PPU unit was a substantially time-consuming activity. This was first because of the complex timing dependencies associated with memory fetch, image construction, and VGA generation. The scanlines are filled almost entirely with memory-access intensive operations.

The second reason for time-consuming Verilog simulation was the very long time periods associated with PPU operations. Displaying a single scan line on the screen requires 31.77us, and a SINGLE frame is 16.69ms. These are huge amounts of time for the simulator, and it would easily take over 10 minutes to simulate a couple frames. This greatly complicated the simulation of vblank, and it made simulating state changes between frames very impractical. Nonetheless, the simulator did prove to be an immensely valuable resource. It allowed early detection of data changes before latching operations, nonfunctioning output lines, and inverted signals.

After the Verilog passed computer simulation verification, we attached the logic analyzer to actual outputs on the lab kit. This was a particularly important and critical testing phase for the CPU. It allowed for verification of values on the data and address buses during memory operations. The analyzer also confirmed correct interfacing with external devices, such as the controllers and the externally supplied interrupts.

During PPU testing, the logic analyzer was helpful in locating glitches in the signal lines and thus pointing out the need for registered inputs and outputs to the circuitry. However, it was still impossible to completely eliminate this problem, which was troubling, especially in light of other unanswered questions.

Writing screen output generation test cases and visually confirming the results was another component of the testing process. The CPU was visually confirmed by playing a tic-tac-toe game custom designed for it.

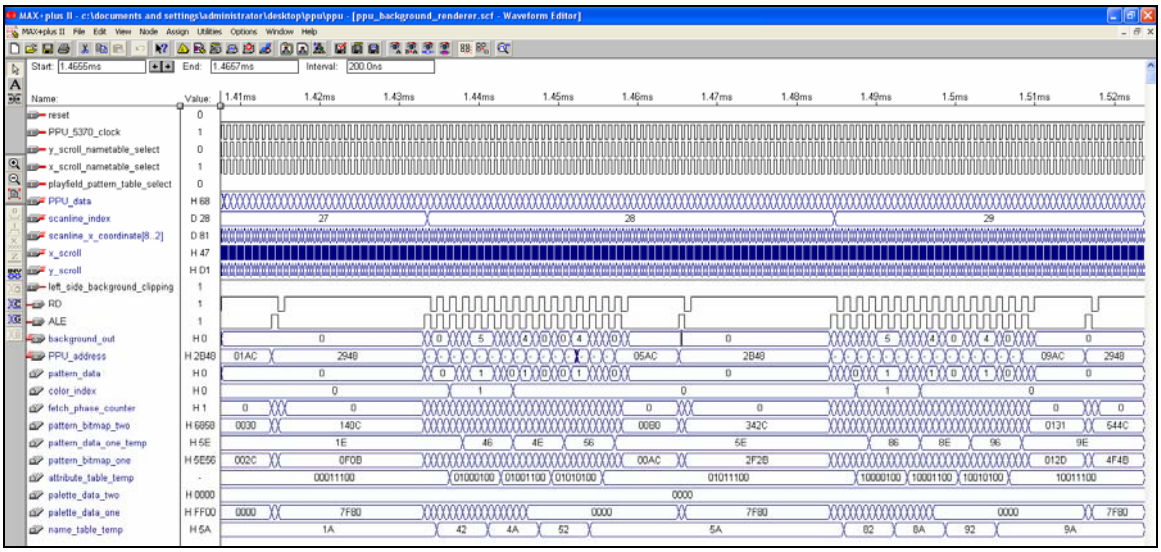


Figure 21 – Example of Simulation of the Background Renderer

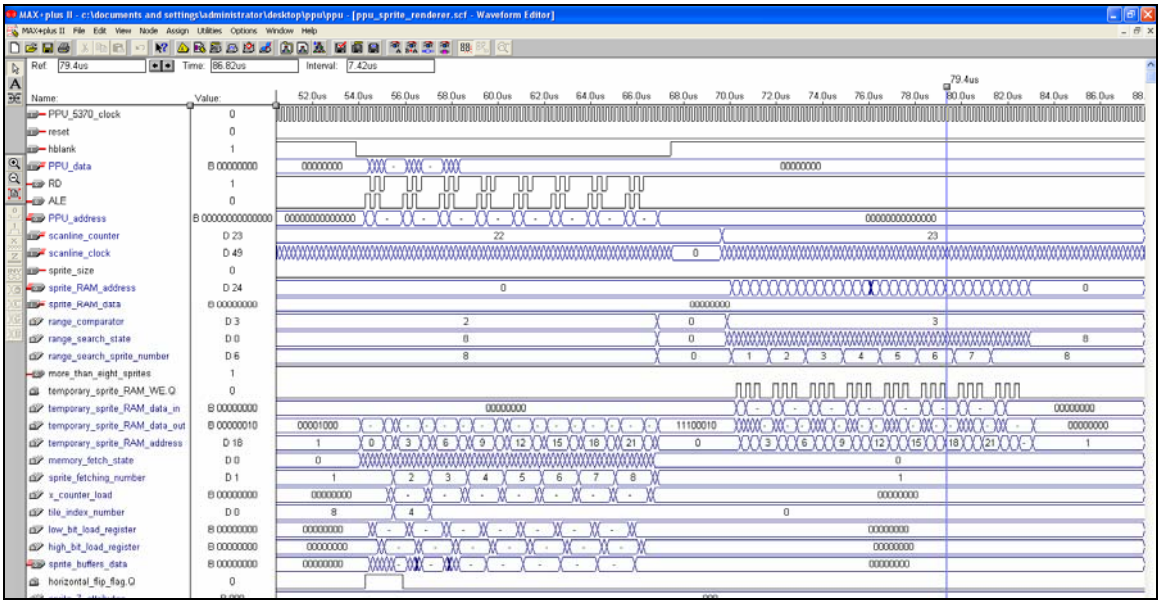


Figure 22 – Example of Simulation of the Sprite Renderer

Conclusion

A hardware emulation of the classic, two decade old Nintendo Entertainment System proved to be a fascinating, some frustrating, and always very educational process. The initial research phase, which exposed us to the enormous number of NES enthusiasts and very active communities of software emulator designers, was a substantial undertaking in itself. Sorting through unverified, and in many cases unauthorized, schematics of the system, understanding the extensive 6502 technical and programming manuals, and distilling the information into valuable clues quickly showed us the time commitment demanded by this undertaking. Once system implementation was underway, hardware failures, wiring challenges, damaged analyzer pods, and other unplanned obstacles stretched us in new ways.

Despite the obstacles, this project was an incredible success. The CPU was entirely implemented and tested on the logic analyzer with the full complex instruction set. An intricate two player tic-tac-toe game, with user input from two original Nintendo controllers, further demonstrated and brilliantly illustrated the processor's operation as it output to a nine by nine array of red and green LEDs.

The PPU was successful in displaying moving sprites on the VGA screen, complete with vertical and horizontal flipping. Any solid background color could be generated, and a full screen of background sprites was well underway. The color lookup tables and VGA generation circuitry performed with incredible reliability and highly accurate representation of the entire 64 color Nintendo palette. Creative and innovate techniques were used to account for the differences in VGA and NTSC resolution while preserving the output image appearance.

We look forward to continued work on this system, and we appreciate everyone who shared the Fall 2004 6.111 experience with us.

Sincerely,
Team Nintendo

Appendix A: Control Signals for Clock Cycle Zero

	ALU_FN	SEL_OPER_A	SEL_OPER_A	A_LOAD	X_SEL	Y_SEL	S_SEL	STATUS_LD	N_SEL	V_SEL	BT_CLEAR	B_SEL	D_SEL	I_SEL	Z_SEL	C_SEL	BIT_VAL
ADC	2	3	2	1	0	0	0	0	1	1	0	0	0	0	1	1	0
AND	4	3	2	1	0	0	0	0	1	0	0	0	0	0	1	0	0
ASL(A)	7	3	0	1	0	0	0	0	1	0	0	0	0	0	1	1	0
ASL																	
BCC																	
BCS																	
BEQ																	
BIT	4	3	2	0	0	0	0	0	2	2	0	0	0	0	1	0	0
BMI																	
BNE																	
BPL																	
BRK																	
BVC																	
BVS																	
CLC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
CLD	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
CLI	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
CLV	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0
CMP	12	3	2	0	0	0	0	0	1	0	0	0	0	0	1	1	0
CPX	12	4	2	0	0	0	0	0	1	0	0	0	0	0	1	1	0
CPY	12	5	2	0	0	0	0	0	1	0	0	0	0	0	1	1	0
DEC																	
DEX	12	4	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0
DEY	12	5	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
EOR	6	3	2	1	0	0	0	0	1	0	0	0	0	0	1	0	0
INC																	
INX	11	4	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0
INY	11	5	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
JMP																	
JSR																	
LDA	0	2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0
LDX	0	2	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0
LDY	0	2	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0
LSR(A)	8	3	0	1	0	0	0	0	1	0	0	0	0	0	1	1	0
LSR																	
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ORA	5	3	2	1	0	0	0	0	1	0	0	0	0	0	1	0	0
PHA																	

PHP																	
PLA	0	2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0
PLP	0	2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
ROL(A)	9	3	0	1	0	0	0	0	1	0	0	0	0	0	1	1	0
ROL																	
ROR(A)	10	3	0	1	0	0	0	0	1	0	0	0	0	0	1	1	0
ROR																	
RTI																	
RTS																	
SBC	3	3	2	1	0	0	0	0	1	1	0	0	0	0	1	1	0
SEC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1
SED	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
SEI	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
STA																	
STX																	
STY																	
TAX	0	3	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0
TAY	0	3	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0
TYA	0	5	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0
TSX	0	6	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0
TXA	0	4	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0
TXS	0	4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
EBT	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Appendix B: Control Signals

Addressing Registers: {ADH,ADL} or {BAH,BAL}

ADX_SEL or BAX_SEL =

ADR_HOLD:	Both Registers Hold Contents
ADR_LATCH_L:	LSB gets value of data bus
ADR_LATCH_H:	MSB gets value of data bus
ADR_LOAD_L:	LSB gets value of R
ADR_LOAD_H:	MSB gets value of R
ADR_ABS_INDEX	LSB gets R and MSB gets data bus

PC Counter:

PC_SEL =

PC_HOLD:	PC holds value
PC_INC:	PC increments value
PC_LATCH_L:	Lower byte gets value of data bus
PC_LATCH_H:	Upper byte gets value of data bus
PC_LOAD_L:	Lower byte gets value from ALU result
PC_LOAD_H:	Lower byte gets value from ALU result
PC_LATCH_ADX	Load both PCH and PCL from ADH and ADL
PC_LATCH_ADXP	Load both PCH and PCL with {ADH,ADL} + 1

Address Bus:

AB_SEL =

AB_PCX;	{PCH,PCL}
AB_ADX;	{ADH,ADL}
AB_BAX;	{BAH,BAL}
AB_AD_ZERO;	{00,ADL}
AB_BA_ZERO;	{00.BAL}
AB_STACK;	{01,S}
AB_NMI_LOW:	{FF,FA}
AB_NMI_HIGH:	{FF,FB}
AB_RESET_LOW:	{FF,FC}
AB_RESET_HIGH:	{FF,FD}
AB_IRQ_LOW:	{FF,FE}
AB_IRQ_HIGH:	{FF,FF}

Instruction Latch

INSTR_SEL =

INSTR_SEL_HOLD	Instruction register holds its value
INSTR_SEL_LOAD	Loads its value from the data bus
INSTR_SEL_FORCE_BRK	Loads the value 0x00 into the instr. Latch

Latch Output

LATCH_OUTPUT = 1	The output register loads a new value from the result of the ALU

LATCH_OUTPUT = 0 The output register holds its current value

Latch Input

LATCH_INPUT = 1 The input register loads the value of the data bus
LATCH_INPUT = 0 The input register holds its value

Key Registers: S, X, Y

S_SEL =

REG_HOLD: Holds Contents
REG_LOAD: Load register from R
REG_INC: Contents Increment
REG_DEC: Contents Decrement

Y_SEL or X_SEL =

REG_HOLD: Holds Contents
REG_LOAD: Load register from R

Accumulator:

A_LOAD = 0 Accumulator Holds Contents
A_LOAD = 1 Accumulator Loads new Value from R

ALU:

ALU_FN =

FN_A = Pass operand A
FN_B = Pass operand B
FN_ADD = Add operand A and B with carry
FN_SUB = Subtract operand B from A including borrow
FN_AND = AND operands A and B
FN_OR = OR operands A and B
FN_XOR = Exclusive OR operands A and B
FN_SL = Shift operand A left and move a zero into bit 0
FN_SR = Shift operand A right and move a zero into bit 7
FN_RL = Rotate op. A left by moving C into bit 0 and bit 7 into C
FN_RR = Rotate op. A right by moving C into bit 07 and bit 0 into C
FN_ADD_NC = Add operand A and B without the carry bit
FN_SUB_NB = Subtract operand B from A without the borrow bit
FN_STATUS = Pass the value of the Status Register

SEL_OPER_A, *SEL_OPER_B* =

OPERAND_ZERO The value 0x00
OPERAND_ONE The value 0x01
OPERAND_MEM The contents of the input latch

SEL_OPER_A =

OPERAND_ACCUM The accumulator
OPERAND_X The X Index
OPERAND_Y The Y Index
OPERAND_S The Stack Pointer

OPERAND_PCH	The upper byte of the Program Counter
SEL_OPER_B =	
OPERAND_ BAH	The upper byte of the Base Address
OPERAND_ BAL	The lower byte of the Base Address
OPERAND_ ADH	The upper byte of the Base Address
OPERAND_ ADL	The lower byte of the Base Address
OPERAND_PCL	The lower byte of the Program Counter

Appendix C: 6502 Instruction Clock Cycles

Operation Operands	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Arithmetic OP OP(A) -> A [Accumulator]	Fetch (PC) PC++	(PC) PC Hold	Fetch Next (PC) Execute OP					
Arithmetic OP OP(Mem) -> A [Immediate]	Fetch (PC) PC++	Latch (PC) PC++	Fetch Next (PC) Execute OP					
Arithmetic OP OP(Mem) -> A [Zero Page]	Fetch (PC) PC++	(PC)->ADL PC++	Latch(00,ADL) PC Hold	Fetch Next (PC) Execute OP				
Arithmetic OP OP(Mem) -> A [Zero, Index]	Fetch (PC) PC++	(PC)->BAL PC++	(00,BAL) PC Hold BAL+X ->BAL	Latch(00,BAL) PC Hold	Fetch Next (PC) Execute OP			
Arithmetic OP OP(Mem) -> A [Absolute]	Fetch (PC) PC++	(PC)->ADL PC++	(PC)->ADH PC++	Latch(ADH,ADL) PC Hold	Fetch Next Execute OP			
Arithmetic OP OP(Mem) -> A [Absolute, Index]	Fetch (PC) PC++	(PC)->BAL PC++	(PC)->BAH PC++ BAL+X ->BAL	Latch(BAH,BAL) BAH+C -> BAH InstrDone = !C	Latch(BAH,BAL) PC Hold	Fetch Next (PC) Execute OP		
Arithmetic OP OP(Mem) -> A [Indirect, X]	Fetch (PC) PC++	(PC)->BAL PC++	(00,BAL) PC Hold BAL+X->BAL	(00,BAL) -> ADL BAL+1 -> BAL	(00,BAL)-> ADH Hold ADL	Latch(ADH,ADL)	Fetch Next (PC) Execute OP	
Arithmetic OP OP(Mem) -> A [Indirect, Y]	Fetch (PC) PC++	(PC)->ADL PC++	(00,ADL)->BAL ADL+1->ADL	(00,ADL)->BAH BAL+Y->BAL	Latch(BAH,BAL) BAH+C->BAH InstrDone = !C	Latch(BAH,BAL)	Fetch Next (PC) Execute OP	
Jump JMP [Absolute-JMP]	Fetch (PC) PC++	(PC) -> ADL PC++	(PC)->ADH PC++	Fetch Next from (ADH,ADL) ADH->PCH ADL->PCL				
Jump JMP [Indirect-JMP]	Fetch (PC) PC++	(PC)-> BAL PC++	(PC)-> BAH PC++	(BAH,BAL)->ADL BAL+1->BAL	(BAH,BAL)->ADH	Fetch Next from (ADH,ADL) ADH->PCH ADL->PCL		
Branch BEQ… ect. [Relative]	Fetch (PC) PC++	Latch (PC) PC ++ InstrDone= !Branch	(PC) PCL+Mem->PCL InstrDone if No Page Cross	(PC) PCH+/-1->PCH InstrDone = 1	Fetch Next (PC)			

Set, Clear, NOP, Transfer [Implied]	Fetch (PC) PC++	(PC) PC Hold	Fetch Next (PC) Execute					
Push PHA, PHP A or Status [Special]	Fetch (PC) PC++	(PC) A,P -> Output PC Hold	Write (S) PC Hold Decrement S	Fetch Next (PC)				
Pull A or Status [Special]	Fetch (PC) PC++	(PC) PC Hold	Read (S) PC Hold Increment S	Latch (S) PC Hold	Fetch Next (PC) M -> A,P			
Break BRK [Special]	Fetch (PC) PC++	(PC) PCH->Output PC++	Write (S) PCL->Output Dec S	Write (S) P->Output Dec S	Write (S) Dec S	(VectorL)->PCL	(VectorH)->PCH Set B	Fetch Next from new PC
RESET, NMI or IRQ Interrupt [Special]	Force BRK Instruction PC HOLD	(PC) PCH->Output Hold PC	Write (S) PCL->Output Dec S	Write (S) P->Output Dec S	Write (S) Dec S	(VectorL)->PCL Set I	(VectorH)->PCH Clear B	Fetch Next from new PC
Return from INT RTI [Special]	Fetch (PC) PC++	(PC) PC Hold	Read (S) PC Hold Increment S	(S) -> P PC Hold Increment S	(S) -> PCL Increment S	(S) -> PCH	Fetch Next from new PC	
Call SR JSR [Absolute-JSR]	Fetch (PC) PC++	(PC)->ADL PC++	(S) PCH->Latch Out	Write (S) Decrement S PCL -> Latch Out	Write (S) Decrement S	(PC)->ADH	Fetch Next from (ADH,ADL) ADH->PCH ADL->PCL	
Return from Sub RTS [Special]	Fetch (PC) PC++	(PC) PC Hold	Read (S) PC Hold Increment S	(S) -> PCL PC Hold Increment S	(S) -> PCH PC Hold	(PC) PC ++	Fetch Next (PC)	
Arithmetic OP Read-Modify-Write OP(Mem)->Mem [Zero Page]	Fetch (PC) PC++	(PC)->ADL PC++	Latch(00,ADL) PC Hold	Execute OP Latch Data Output	Write (00,ADL)	Fetch Next (PC)		
Arithmetic OP Read-Modify-Write OP(Mem)->Mem [Zero Page, X]	Fetch (PC) PC++	(PC)->BAL PC++	(00,BAL) PC Hold BAL+X ->BAL	Latch(00,BAL) PC Hold	Execute OP Latch Data Output	Write (00,BAL)	Fetch Next (PC)	
Arithmetic OP Read-Modify-Write OP(Mem)->Mem [Absolute]	Fetch (PC) PC++	(PC)->ADL PC++	(PC+2)->ADH PC++	Latch(ADH,ADL) PC Hold	Execute OP Latch Data Output	Write (ADH,ADL)	Fetch Next (PC)	
Arithmetic OP Read-Modify-Write OP(Mem)->Mem [Absolute, X]	Fetch (PC) PC++	(PC)->BAL PC++	(PC+2)->BAH PC++ BAL+X ->BAL	Latch(BAH,BAL) BAH+C -> BAH InstrDone = 0	Latch(BAH,BAL) PC Hold InstrDone = 0	Execute OP Latch Output	Write (BAH,BAL)	Fetch Next (PC)

Appendix D: CPU Verilog

addr_mod_decode.v

```
module addr_mode_decode(instr, ADDR_MODE);
    input [7:0] instr;
    output [3:0] ADDR_MODE;

    // Addressing Modes
    parameter ADDR_ACCUM = 0;
    parameter ADDR_IMM =  1;
    parameter ADDR_ZERO =  2;
    parameter ADDR_ZERO_X =   3;
    parameter ADDR_ZERO_Y = 4;
    parameter ADDR_ABS =  5;
    parameter ADDR_ABS_X = 6;
    parameter ADDR_ABS_Y = 7;
    parameter ADDR_IMP =  8;
    parameter ADDR_REL =  9;
    parameter ADDR_IND_X = 10;
    parameter ADDR_IND_Y = 11;
    parameter ADDR_INDIR = 12;
    parameter ADDR_SPECIAL = 13;

    reg [3:0] ADDR_MODE;
    always@(instr) begin
        if
            ((instr==8'h0a)||
            (instr==8'h4a)||
            (instr==8'h2a)||
            (instr==8'h6a)
            )   ADDR_MODE = ADDR_ACCUM;
        else if ((instr==8'h09)||
            (instr==8'h29)||
            (instr==8'h49)||
            (instr==8'h69)||
            (instr==8'ha0)||
            (instr==8'ha2)||
            (instr==8'ha9)||
            (instr==8'hc0)||
            (instr==8'hc9)||
            (instr==8'he0)||
            (instr==8'he9)
            )   ADDR_MODE = ADDR_IMM;
        else if
            ((instr==8'h05)||
            (instr==8'h06)||
            (instr==8'h24)||
            (instr==8'h25)||
            (instr==8'h26)||
            (instr==8'h45)||
            (instr==8'h46)||
            (instr==8'h65)||
            (instr==8'h66)||
            (instr==8'h84)||
            (instr==8'h85)||
            (instr==8'h86)||
            (instr==8'ha4)||
            (instr==8'ha5)||
            (instr==8'ha6)||
            (instr==8'hc4)||
            (instr==8'hc5)||
            (instr==8'hc6)||
            (instr==8'he4)||
            (instr==8'he5)||
            (instr==8'he6)
            )   ADDR_MODE = ADDR_ZERO;
        else if
```

```
    ((instr==8'h15)||
    (instr==8'h16)||
    (instr==8'h35)||
    (instr==8'h36)||
    (instr==8'h55)||
    (instr==8'h56)||
    (instr==8'h75)||
    (instr==8'h76)||
    (instr==8'h94)||
    (instr==8'h95)||
    (instr==8'hb4)||
    (instr==8'hb5)||
    (instr==8'hd5)||
    (instr==8'hd6)||
    (instr==8'hf5)||
    (instr==8'hf6)
    )   ADDR_MODE = ADDR_ZERO_X;
else if
    ((instr==8'h96)||
    (instr==8'hb6)
    )   ADDR_MODE = ADDR_ZERO_Y;
else if
    ((instr==8'h0d)||
    (instr==8'h0e)||
    (instr==8'h20)||
    (instr==8'h2c)||
    (instr==8'h2d)||
    (instr==8'h2e)||
    (instr==8'h4c)||
    (instr==8'h4d)||
    (instr==8'h4e)||
    (instr==8'h6d)||
    (instr==8'h6e)||
    (instr==8'h8c)||
    (instr==8'h8d)||
    (instr==8'h8e)||
    (instr==8'hac)||
    (instr==8'had)||
    (instr==8'hae)||
    (instr==8'hcc)||
    (instr==8'hcd)||
    (instr==8'hce)||
    (instr==8'hec)||
    (instr==8'hed)||
    (instr==8'hee)
    )   ADDR_MODE = ADDR_ABS;
else if
    ((instr==8'h1d)||
    (instr==8'h1e)||
    (instr==8'h3d)||
    (instr==8'h3e)||
    (instr==8'h5d)||
    (instr==8'h5e)||
    (instr==8'h7d)||
    (instr==8'h7e)||
    (instr==8'h9d)||
    (instr==8'hbc)||
    (instr==8'hbd)||
    (instr==8'hdd)||
    (instr==8'hde)||
    (instr==8'hfd)||
    (instr==8'hfe)
    )   ADDR_MODE = ADDR_ABS_X;
else if
    ((instr==8'h19)||
    (instr==8'h39)||
    (instr==8'h59)||
    (instr==8'h79)||
    (instr==8'h99)||
    (instr==8'hb9)||
    (instr==8'hd9)||
```

```
            (instr==8'hf9)||
            (instr==8'hbe)                              )   ADDR_MODE = ADDR_ABS_Y;
        else if
            ((instr==8'h10)||
            (instr==8'h30)||
            (instr==8'h50)||
            (instr==8'h70)||
            (instr==8'h90)||
            (instr==8'hb0)||
            (instr==8'hd0)||
            (instr==8'hf0)
            )   ADDR_MODE = ADDR_REL;
        else if
            ((instr==8'h01)||
            (instr==8'h21)||
            (instr==8'h41)||
            (instr==8'h61)||
            (instr==8'h81)||
            (instr==8'ha1)||
            (instr==8'hc1)||
            (instr==8'he1)
            )   ADDR_MODE = ADDR_IND_X;
        else if
            ((instr==8'h11)||
            (instr==8'h31)||
            (instr==8'h51)||
            (instr==8'h71)||
            (instr==8'h91)||
            (instr==8'hb1)||
            (instr==8'hd1)||
            (instr==8'hf1)
            )   ADDR_MODE = ADDR_IND_Y;
        else if (instr==8'h6c) ADDR_MODE = ADDR_INDIR;
        else if
            ((instr==8'h48)||
            (instr==8'h08)||
            (instr==8'h68)||
            (instr==8'h28)||
            (instr==8'h40)||
            (instr==8'h60)||
            (instr==8'h00)
            )   ADDR_MODE = ADDR_SPECIAL;


        else    ADDR_MODE = ADDR_IMP;
    end

endmodule
```

ALU.v

```
/* ======================================================================
|   NES - 6502 - ALU
|-----------------------------------------------------------------------
|   Description:
|       - Two Operands: A and B
|       - 11 Basic Operations on the Operands
|
|-----------------------------------------------------------------------
|   Created by Team Nintendo:  team-nintendo@mit.edu
\======================================================================*/

module alu(alu_fn, A, B, Cin, R, Cout, Zout, Vout, Nout);
    input [3:0] alu_fn;
    input [7:0] A;
    input [7:0] B;
    input Cin;
```

```
    output [7:0] R;
    output Cout;
    output Zout;
    output Vout;
    output Nout;

    // Functions
    parameter FN_A =   0;
    parameter FN_B =   1;
    parameter FN_ADD = 2;
    parameter FN_SUB = 3;
    parameter FN_AND = 4;
    parameter FN_OR =  5;
    parameter FN_XOR =  6;
    parameter FN_SL =  7;
    parameter FN_SR =  8;
    parameter FN_RL =  9;
    parameter FN_RR =  10;
    parameter FN_ADD_NC = 11;
    parameter FN_SUB_NB = 12;

    wire [7:0] result_sig;
    wire cout_sig;
    wire overflow_sig;
    wire Cin_val;

    assign Cin_val = (alu_fn==FN_ADD_NC) ? 1'b0 :
                     (alu_fn==FN_SUB_NB) ? 1'b1 :
                     Cin;

    add_sub add_sub_inst (
    .add_sub ( ~((alu_fn == FN_SUB) || (alu_fn == FN_SUB_NB)) ),
    .dataa ( A ),
    .datab ( B ),
    .cin ( Cin_val ),
    .result ( result_sig ),
    .cout ( cout_sig ),
    .overflow ( Vout )
    );

    assign R = (alu_fn == FN_A) ? A :
               (alu_fn == FN_B) ? B :
               ((alu_fn == FN_ADD) || (alu_fn == FN_SUB) || (alu_fn==FN_ADD_NC) ||
(alu_fn==FN_SUB_NB)) ?  result_sig:
               (alu_fn == FN_AND) ? (A & B) :
               (alu_fn == FN_OR) ? (A | B) :
               (alu_fn == FN_XOR) ? (A ^ B) :
               (alu_fn == FN_SL) ? {A[6:0], 1'b0} :
               (alu_fn == FN_SR) ? {1'b0, A[7:1]} :
               (alu_fn == FN_RL) ? {A[6:0], Cin} :
               (alu_fn == FN_RR) ? {Cin, A[7:1]} :
               8'b00000000;

    assign Cout =  ((alu_fn == FN_ADD) || (alu_fn == FN_SUB) || (alu_fn==FN_ADD_NC) ||
(alu_fn==FN_SUB_NB)) ? cout_sig :
                   ((alu_fn == FN_RL) || (alu_fn == FN_SL)) ? A[7] :
                   ((alu_fn == FN_RR) || (alu_fn == FN_SR)) ? A[0] :
                   1'b0;

    assign Nout = R[7];

    assign Zout = (R == 8'd0) ? 1'b1 : 1'b0;

endmodule
```

Clock_Generator.v

```
/* ======================================================================
```

```
|   NES - 6502 - Clock Generator
|       Generates the phase one and phase two clocks by dividing by 5.5 (10 MHz)
|       to get 1.81 MHz
|
|   Cycle   0123456789AB0123456789AB0123456789AB0123456789AB0123456789AB
|   Input   ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  1/2  duty
|   clk1    ------_______------_______------_______------_______------_____  6/12 duty
|   clk2    _______------_______------_______------_______------_______-----  6/12 duty
|
|   Created by Team Nintendo:  team-nintendo@mit.edu
|
\=========================================================================*/

module Clock_Generator(MCLR, Clock_Input, clk1, clk2);
    input  MCLR, Clock_Input;
    output clk1, clk2;

    reg [3:0] cycle;
    reg clk1;
    reg clk2;

    always @(posedge Clock_Input)
    begin
        if (MCLR)
        begin
            case (cycle)
            4'd5:
                begin
                clk1 <= 0;
                clk2 <= 0;
                end
            4'd11:
                begin
                clk1 <= 1;
                clk2 <= 0;
                end
            default:
                begin
                clk1 <= clk1;
                clk2 <= 0;
                end
            endcase
            if (cycle!=4'd11)
                cycle <= cycle + 1;
            else
                cycle <= 0;
        end
        else begin
            case (cycle)
            4'd5:
                begin
                clk1 <= 0;
                clk2 <= 1;
                end
            4'd11:
                begin
                clk1 <= 1;
                clk2 <= 0;
                end
            default:
                begin
                clk1 <= clk1;
                clk2 <= clk2;
                end
            endcase
            if (cycle!=4'd11)
                cycle <= cycle + 1;
            else
                cycle <= 0;
        end
    end
```

```
endmodule
```

CPU.v

```
/* =====================================================================
|   NES - 6502 - CPU
|---------------------------------------------------------------------
|   Description:
|       - Main CPU Module
|
|---------------------------------------------------------------------
|   Created by Team Nintendo:  team-nintendo@mit.edu
\=====================================================================*/

module cpu(clock, reset_line, nmi_line, irq_line,
           data_bus_out, data_bus_in, address_bus, RW, clk1, clk2, BT);

//=====================================================================
//  Declare Input and Output Pins
//---------------------------------------------------------------------
    input clock;
    input reset_line, nmi_line, irq_line;    // For Interrupts

    output  [7:0] data_bus_out;                 // Busses
    input  [7:0] data_bus_in;
    output [15:0] address_bus;              //
    output RW;                              // Bus Direction
    output clk1;
    output clk2;
    output BT;

    wire [7:0] data_bus;
    assign data_bus = data_bus_in;
//=====================================================================


//=====================================================================
//  Constants
//---------------------------------------------------------------------
    // Interrupt Vectors
    parameter  RESET_VECTOR_L =   8'hfc;
    parameter  RESET_VECTOR_H =   8'hfd;
    parameter  IRQ_VECTOR_L =     8'hfe;
    parameter  IRQ_VECTOR_H =     8'hff;
    parameter  NMI_VECTOR_L =     8'hfa;
    parameter  NMI_VECTOR_H =     8'hfb;

    parameter  VECTOR_PAGE =      8'hff;
    parameter  STACK_PAGE =       8'h01;
    parameter  ZERO_PAGE =        8'h00;

    // OP_CODES
    parameter OP_NOP =     0;
    parameter OP_ADC = 1;
    parameter OP_AND = 2;
    parameter OP_ASL = 3;
    parameter OP_BCC = 4;
    parameter OP_BCS = 5;
    parameter OP_BEQ = 6;
    parameter OP_BIT = 7;
    parameter OP_BMI = 8;
    parameter OP_BNE = 9;
    parameter OP_BPL = 10;
    parameter OP_BRK = 11;
    parameter OP_BVC = 12;
    parameter OP_BVS = 13;
```

```
parameter OP_CLC = 14;
parameter OP_CLD = 15;
parameter OP_CLI = 16;
parameter OP_CLV = 17;
parameter OP_CMP = 18;
parameter OP_CPX = 19;
parameter OP_CPY = 20;
parameter OP_DEC = 21;
parameter OP_DEX = 22;
parameter OP_DEY = 23;
parameter OP_EOR = 24;
parameter OP_INC = 25;
parameter OP_INX = 26;
parameter OP_INY = 27;
parameter OP_JMP = 28;
parameter OP_JSR = 29;
parameter OP_LDA = 30;
parameter OP_LDX = 31;
parameter OP_LDY = 32;
parameter OP_LSR = 33;
parameter OP_ORA = 34;
parameter OP_PHA = 35;
parameter OP_PHP = 36;
parameter OP_PLA = 37;
parameter OP_PLP = 38;
parameter OP_ROL = 39;
parameter OP_ROR = 40;
parameter OP_RTI = 41;
parameter OP_RTS = 42;
parameter OP_SBC = 43;
parameter OP_SEC = 44;
parameter OP_SED = 45;
parameter OP_SEI = 46;
parameter OP_STA = 47;
parameter OP_STX = 48;
parameter OP_STY = 49;
parameter OP_TAX = 50;
parameter OP_TAY = 51;
parameter OP_TYA = 52;
parameter OP_TSX = 53;
parameter OP_TXA = 54;
parameter OP_TXS = 55;
parameter OP_EBT = 56;

// Addressing Modes
parameter ADDR_ACCUM = 0;
parameter ADDR_IMM =  1;
parameter ADDR_ZERO =  2;
parameter ADDR_ZERO_X =   3;
parameter ADDR_ZERO_Y = 4;
parameter ADDR_ABS =  5;
parameter ADDR_ABS_X = 6;
parameter ADDR_ABS_Y = 7;
parameter ADDR_IMP =  8;
parameter ADDR_REL =  9;
parameter ADDR_IND_X = 10;
parameter ADDR_IND_Y = 11;
parameter ADDR_INDIR = 12;
parameter ADDR_SPECIAL = 13;


// Vector Modes
parameter VM_NORMAL = 0;
parameter VM_RESET = 1;
parameter VM_IRQ = 2;
parameter VM_NMI = 3;


// Custom Boot option
parameter  BOOT_ON_RESET =       1;            // 1 for True, 0 for False
```

```
// INSTR_SEL
parameter INSTR_SEL_HOLD = 0;
parameter INSTR_SEL_LOAD = 1;
parameter INSTR_SEL_FORCE_BRK =  2;

// ADX_SEL and BAX_SEL
parameter ADR_HOLD =      0;
parameter ADR_LATCH_L =   1;
parameter ADR_LATCH_H =   2;
parameter ADR_LOAD_L =    3;
parameter ADR_LOAD_H =    4;
parameter ADR_ABS_INDEX = 5;

// AB_SEL
parameter AB_PCX =    0;  //  {PCH,PCL}
parameter AB_ADX =    1;  //  {ADH,ADL}
parameter AB_BAX =    2;  //  {BAH,BAL}
parameter AB_AD_ZERO =    3;  //  {00,ADL}
parameter AB_BA_ZERO =    4;  //  {00,BAL}
parameter AB_STACK =  5;  //  {01,S}
parameter AB_NMI_LOW = 6;  //  {FF,FA}
parameter AB_NMI_HIGH =   7;  //  {FF,FB}
parameter AB_RESET_LOW =8; //  {FF,FC}
parameter AB_RESET_HIGH=9; //  {FF,FD}
parameter AB_IRQ_LOW = 10; //  {FF,FE}
parameter AB_IRQ_HIGH= 11; //  {FF,FF}

// X_SEL, S_SEL, Y_SEL
parameter REG_HOLD =  0;
parameter REG_LOAD =  1;
parameter REG_INC =       2;
parameter REG_DEC =       3;

// PC_SEL
parameter PC_HOLD =   0;
parameter PC_INC =    1;
parameter PC_LATCH_L = 2;
parameter PC_LATCH_H = 3;
parameter PC_LOAD_L =  4;
parameter PC_LOAD_H =  5;
parameter PC_LATCH_ADX =6;
parameter PC_LATCH_ADXP=7;

// N_SEL
parameter SEL_ALU_N =  1;
parameter SEL_M7   =   2;
// V_SEL
parameter SEL_ALU_V   =   1;
parameter SEL_BIT_V   =   3;
parameter SEL_M6   =   2;
// C_SEL
parameter SEL_ALU_C   =   1;
parameter SEL_BIT_C   =   2;

// SEL_OPER
parameter OPERAND_ZERO =  0;
parameter OPERAND_ONE =   1;
parameter OPERAND_MEM = 2;
//---
parameter OPERAND_ACCUM = 3;
parameter OPERAND_X =     4;
parameter OPERAND_Y =     5;
parameter OPERAND_S =     6;
parameter OPERAND_PCH =   7;
//---
parameter OPERAND_BAH =       3;
parameter OPERAND_BAL =       4;
parameter OPERAND_ADH =       5;
parameter OPERAND_ADL =       6;
parameter OPERAND_PCL =   7;
```

```
    // ALU_FN
    parameter FN_A =   0;
    parameter FN_B =   1;
    parameter FN_ADD = 2;
    parameter FN_SUB = 3;
    parameter FN_AND = 4;
    parameter FN_OR =  5;
    parameter FN_XOR =  6;
    parameter FN_SL =  7;
    parameter FN_SR =  8;
    parameter FN_RL =  9;
    parameter FN_RR =  10;
    parameter FN_ADD_NC = 11;
    parameter FN_SUB_NB = 12;
    parameter FN_STATUS = 13;

//=============================================================================



//=============================================================================
//  Declarations
//-----------------------------------------------------------------------------
    // Reset Circuitry
    wire MCLR;

    // Clock Generator
    wire clk1, clk2;

    // Timing Control Unit
    wire [2:0] cycle;
    reg     INSTR_COMPLETE;

    // Interrupt Controller
    wire IRQ, NMI, RESET_INT;
    reg IRQ_CLEAR, NMI_CLEAR, RESET_CLEAR;
    reg [1:0] vector_mode;

    // To ALU
    reg [3:0]  ALU_FN;
    reg [2:0]  SEL_OPER_A;
    reg [2:0]  SEL_OPER_B;

    // To Registers
    reg        A_LOAD;
    reg [1:0]  X_SEL;
    reg [1:0]  Y_SEL;
    reg [1:0]  S_SEL;

    // Status Register
    reg        STATUS_LD;
    reg [1:0]  N_SEL;
    reg [1:0]  V_SEL;
    reg        BT_CLEAR;
    reg        B_SEL;
    reg        D_SEL;
    reg        I_SEL;
    reg        Z_SEL;
    reg [1:0]  C_SEL;
    reg        BIT_VAL;

    // To Data Bus
    reg        LATCH_OUTPUT;
    reg        LATCH_INPUT;
    reg [1:0]  INSTR_SEL;

    // To Address Bus and Addressing
    reg        RW;
    reg [3:0]  AB_SEL;
    reg [2:0]  ADX_SEL;
    reg [2:0]  BAX_SEL;
```

```
    // To Program Counter
    reg [2:0]  PC_SEL;

    // Instr Decode
    wire [5:0] OP_CODE;

    // Address Mode Decode
    wire [3:0] ADDRESS_MODE;

    // ALU
    reg  [7:0] operand_a;
    reg  [7:0] operand_b;
    reg        previous_aluC;
    reg  [7:0] R;
    wire [7:0] R_out;
    wire aluC, aluZ, aluV, aluN;

    // Status Register
    reg N, V, BT, B, D, I, Z, C;
    wire [7:0] STATUS;

    // Program Counter
    reg [7:0]  PCH;
    reg [7:0]  PCL;
    wire [15:0] PC;
    wire [15:0] PC_PLUS;
    wire [15:0] ADX_PLUS;

    // Key Registers
    reg [7:0] A;
    reg [7:0] S;
    reg [7:0] X;
    reg [7:0] Y;

    // Addressing Registers
    reg [7:0] ADH;
    reg [7:0] ADL;
    reg [7:0] BAH;
    reg [7:0] BAL;

    // Address Bus Output
    reg [7:0] ABH;
    reg [7:0] ABL;

    // Data Bus Buffers
    reg [7:0] output_latch;
    reg [7:0] input_latch;
    reg [7:0] instr;
//=============================================================================



//=============================================================================
//  Modules Used
//-----------------------------------------------------------------------------
    // Reset Circuitry
    Reset             RESET_M(clock, reset_line, MCLR);

    // Clock Generator
    Clock_Generator       CLOCK_M(MCLR, clock, clk1, clk2);

    // Timing Control Unit
    timing_control    TIME_CTRL(MCLR, clk1, INSTR_COMPLETE, cycle);
//=============================================================================



//=============================================================================
//  Interrupt Controller
//-----------------------------------------------------------------------------
```

```
    Interrupt_Control  INT_CTRL(MCLR, clk1, I, irq_line, IRQ_CLEAR,
                                nmi_line, NMI_CLEAR,
                                IRQ, NMI,
                                RESET_INT, RESET_CLEAR);
    // Vector Mode gets set at end of cycle0 or on Reset based on Interrupts
    always@(posedge clk1) begin
        if (MCLR)
            vector_mode <= VM_RESET;
        else begin
            if (cycle==3'd0) begin
                if (RESET_INT || (OP_CODE==OP_EBT))
                    vector_mode <= VM_RESET;
                else if (NMI)
                    vector_mode <= VM_NMI;
                else if (IRQ)
                    vector_mode <= VM_IRQ;
                else
                    vector_mode <= VM_NORMAL;
            end
        end
    end
    always@(MCLR or cycle or vector_mode) begin
        if (MCLR) begin
            NMI_CLEAR = 0;
            IRQ_CLEAR = 0;
            RESET_CLEAR = 0;
        end
        if (cycle==3'd1) begin
            case(vector_mode)
            VM_RESET:
                begin
                    NMI_CLEAR = 0;
                    IRQ_CLEAR = 0;
                    RESET_CLEAR = 1;
                end
            VM_NMI:
                begin
                    NMI_CLEAR = 1;
                    IRQ_CLEAR = 0;
                    RESET_CLEAR = 0;
                end
            VM_IRQ:
                begin
                    NMI_CLEAR = 0;
                    IRQ_CLEAR = 1;
                    RESET_CLEAR = 0;
                end
            default:
                begin
                    NMI_CLEAR = 0;
                    IRQ_CLEAR = 0;
                    RESET_CLEAR = 0;
                end
            endcase
        end
        else begin
            NMI_CLEAR = 0;
            IRQ_CLEAR = 0;
            RESET_CLEAR = 0;
        end
    end
//=============================================================================


//=============================================================================
//  Instruction Decode
//-----------------------------------------------------------------------------
    // Instr Decode
    instr_decode INSTR_DECODE_INST(instr, OP_CODE);
```

```
    // Address Mode Decode
    addr_mode_decode ADDR_DECODE(instr, ADDRESS_MODE);
//---------------------------------------------------------------------------


//===========================================================================
//   Actual Generation of Control Signals
//===========================================================================
    always@(MCLR or OP_CODE or cycle or ADDRESS_MODE or RESET_INT or NMI or IRQ or previous_aluC
or vector_mode or C or
            Z or N or V or aluC) begin
        if (MCLR) begin
            AB_SEL = AB_PCX;
            RW = 1;
            PC_SEL = PC_HOLD;
            ADX_SEL = ADR_HOLD;
            BAX_SEL = ADR_HOLD;
            INSTR_SEL = INSTR_SEL_HOLD;
            LATCH_OUTPUT = 0;
            LATCH_INPUT = 0;
            INSTR_COMPLETE = 1;

            ALU_FN = 0;
            SEL_OPER_A = 0;
            SEL_OPER_B = 0;

            A_LOAD = 0;
            X_SEL = 0;
            Y_SEL = 0;
            S_SEL = 0;

            STATUS_LD = 0;
            N_SEL = 0;
            V_SEL = 0;
            BT_CLEAR = 0;
            B_SEL = 0;
            D_SEL = 0;
            I_SEL = 0;
            Z_SEL = 0;
            C_SEL = 0;
            BIT_VAL = 0;
        end
        else begin
//------RUNNING: CYCLE 0  ------------------------------------------------
            if (cycle==3'd0) begin
                RW = 1;                                 // Read Operation
                LATCH_OUTPUT = 0;
                LATCH_INPUT = 0;
                ADX_SEL = 0;
                BAX_SEL = 0;
                INSTR_COMPLETE = 0;                         // Not Complete

                if ((OP_CODE == OP_JMP) || (OP_CODE == OP_JSR))
                    AB_SEL = AB_ADX;                        // New PC on Address Bus
                else
                    AB_SEL = AB_PCX;                        // PC is on Address Bus

                // RESET INTERRUPT
                if (RESET_INT || (OP_CODE==OP_EBT)) begin
                    if ((OP_CODE == OP_JMP) || (OP_CODE == OP_JSR))
                        PC_SEL = PC_LATCH_ADXP;                 // Increment PC from New PC
                    else
                        PC_SEL = PC_INC;                    // Increment PC
                    INSTR_SEL = INSTR_SEL_FORCE_BRK;  // Force Break Instruction
                end

                // NMI and IRQ INTERRUPTS
                else if (NMI || IRQ) begin
                    if ((OP_CODE == OP_JMP) || (OP_CODE == OP_JSR))
                        PC_SEL = PC_LATCH_ADX;              // Hold PC from New PC
                    else
```

```
                    PC_SEL = PC_HOLD;                // Hold PC
                INSTR_SEL = INSTR_SEL_FORCE_BRK;  // Force Break Instruction
            end

            // Normal Instruction Execution
            else begin
                if ((OP_CODE == OP_JMP) || (OP_CODE == OP_JSR))
                    PC_SEL = PC_LATCH_ADXP;              // Increment PC from New PC
                else
                    PC_SEL = PC_INC;                 // Increment PC
                INSTR_SEL = INSTR_SEL_LOAD;          // Load Instruction
            end

            // FINISH PREVIOUS COMPUTATION:

            // A_LOAD
            if ((OP_CODE==OP_ADC) || (OP_CODE==OP_AND) || ((OP_CODE==OP_ASL) &&
(ADDRESS_MODE==ADDR_ACCUM)) ||
                (OP_CODE==OP_EOR) || (OP_CODE==OP_LDA) || ((OP_CODE==OP_LSR) &&
(ADDRESS_MODE==ADDR_ACCUM)) ||
                (OP_CODE==OP_ORA) || (OP_CODE==OP_PLA) || ((OP_CODE==OP_ROL) &&
(ADDRESS_MODE==ADDR_ACCUM)) ||
                ((OP_CODE==OP_ROR) && (ADDRESS_MODE==ADDR_ACCUM)) || (OP_CODE==OP_SBC) ||
(OP_CODE==OP_TYA) ||
                (OP_CODE==OP_TXA))
                A_LOAD = 1;
            else if (OP_CODE==OP_PLA)
                A_LOAD = 1;
            else
                A_LOAD = 0;

            // X SEL
            if ((OP_CODE==OP_DEX) || (OP_CODE==OP_INX) || (OP_CODE==OP_LDX) ||
(OP_CODE==OP_TAX) || (OP_CODE==OP_TSX))
                X_SEL = REG_LOAD;
            else
                X_SEL = REG_HOLD;

            // Y SEL
            if ((OP_CODE==OP_DEY) || (OP_CODE==OP_INY) || (OP_CODE==OP_LDY) ||
(OP_CODE==OP_TAY))
                Y_SEL = REG_LOAD;
            else
                Y_SEL = REG_HOLD;

            // S_SEL
            if (OP_CODE==OP_TXS)
                S_SEL = REG_LOAD;
            else
                S_SEL = REG_HOLD;

            // ALU_FN
            if (OP_CODE==OP_ADC)
                ALU_FN = 2;
            else if (OP_CODE==OP_SBC)
                ALU_FN = 3;
            else if ((OP_CODE==OP_AND) || (OP_CODE==OP_BIT))
                ALU_FN = 4;
            else if (OP_CODE==OP_ORA)
                ALU_FN = 5;
            else if (OP_CODE==OP_EOR)
                ALU_FN = 6;
            else if ((OP_CODE==OP_ASL) && (ADDRESS_MODE==ADDR_ACCUM))
                ALU_FN = 7;
            else if ((OP_CODE==OP_LSR) && (ADDRESS_MODE==ADDR_ACCUM))
                ALU_FN = 8;
            else if ((OP_CODE==OP_ROL) && (ADDRESS_MODE==ADDR_ACCUM))
                ALU_FN = 9;
            else if ((OP_CODE==OP_ROR) && (ADDRESS_MODE==ADDR_ACCUM))
                ALU_FN = 10;
            else if ((OP_CODE==OP_INX) || (OP_CODE==OP_INY))
```

```
                ALU_FN = 11;
            else if ((OP_CODE==OP_CMP) || (OP_CODE==OP_CPX) || (OP_CODE==OP_CPY) ||
                     (OP_CODE==OP_DEX) || (OP_CODE==OP_DEY))
                ALU_FN = 12;
            else if ((OP_CODE==OP_PLA) || (OP_CODE==OP_PLP))
                ALU_FN = FN_A;
            else
                ALU_FN = 0;

            // SEL_OPER_A
            if ((OP_CODE==OP_PLA) || (OP_CODE==OP_PLP) || (OP_CODE==OP_LDA) ||
(OP_CODE==OP_LDX) || (OP_CODE==OP_LDY))
                SEL_OPER_A = 2;
            else if ((OP_CODE==OP_ADC) || (OP_CODE==OP_AND) || (OP_CODE==OP_BIT) ||
(OP_CODE==OP_CMP) || (OP_CODE==OP_EOR) ||
                     (OP_CODE==OP_ORA) || (OP_CODE==OP_SBC) || (OP_CODE==OP_TAX) ||
(OP_CODE==OP_TAY) ||
                     (((OP_CODE==OP_ASL) || (OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) ||
(OP_CODE==OP_ROR)) && (ADDRESS_MODE==ADDR_ACCUM)))
                SEL_OPER_A = 3;
            else if ((OP_CODE==OP_CPX) || (OP_CODE==OP_DEX) || (OP_CODE==OP_INX) ||
                     (OP_CODE==OP_TXA) || (OP_CODE==OP_TXS))
                SEL_OPER_A = 4;
            else if ((OP_CODE==OP_CPY) || (OP_CODE==OP_DEY) || (OP_CODE==OP_INY) ||
                     (OP_CODE==OP_TYA))
                SEL_OPER_A = 5;
            else if (OP_CODE==OP_TSX)
                SEL_OPER_A = 6;
            else
                SEL_OPER_A = 0;

            // SEL_OPER_B
            if ((OP_CODE==OP_DEX) || (OP_CODE==OP_INX) || (OP_CODE==OP_DEY) ||
(OP_CODE==OP_INY))
                SEL_OPER_B = 1;
            else if ((OP_CODE==OP_ADC) || (OP_CODE==OP_AND) || (OP_CODE==OP_BIT) ||
(OP_CODE==OP_CMP) || (OP_CODE==OP_EOR) ||
                     (OP_CODE==OP_ORA) || (OP_CODE==OP_SBC) || (OP_CODE==OP_CPX) ||
(OP_CODE==OP_CPY))
                SEL_OPER_B = 2;
            else
                SEL_OPER_B = 0;

            // STATUS_LD
            if (OP_CODE==OP_PLP)
                STATUS_LD = 1;
            else
                STATUS_LD = 0;

            // BIT_VAL
            if ((OP_CODE==OP_SEC) || (OP_CODE==OP_SED) || (OP_CODE==OP_SEI))
                BIT_VAL = 1;
            else
                BIT_VAL = 0;

            // BT_CLEAR
            if (OP_CODE==OP_EBT)
                BT_CLEAR = 1;
            else
                BT_CLEAR = 0;

            // B_SEL
            B_SEL = 0;

            // D_SEL
            if ((OP_CODE==OP_CLD) || (OP_CODE==OP_SED))
                D_SEL = 1;
            else
                D_SEL = 0;

            // I_SEL
```

```
                if ((OP_CODE==OP_CLI) || (OP_CODE==OP_SEI))
                    I_SEL = 1;
                else
                    I_SEL = 0;

                // N_SEL and Z_SEL
                if (OP_CODE==OP_BIT)
                    begin
                    N_SEL = SEL_M7;
                    Z_SEL = 1;
                    end
                else if ((OP_CODE==OP_ADC) || (OP_CODE==OP_AND) || (OP_CODE==OP_CMP) ||
(OP_CODE==OP_CPX) || (OP_CODE==OP_CPY) ||
                        (OP_CODE==OP_DEX) || (OP_CODE==OP_DEY) || (OP_CODE==OP_EOR) ||
(OP_CODE==OP_INX) || (OP_CODE==OP_INY) ||
                        (OP_CODE==OP_LDA) || (OP_CODE==OP_LDX) || (OP_CODE==OP_LDY) ||
(OP_CODE==OP_ORA) || (OP_CODE==OP_PLA) ||
                        (OP_CODE==OP_SBC) || (OP_CODE==OP_TAX) || (OP_CODE==OP_TAY) ||
(OP_CODE==OP_TYA) || (OP_CODE==OP_TSX) ||
                        (OP_CODE==OP_TXA) ||
                        (((OP_CODE==OP_ASL) || (OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) ||
(OP_CODE==OP_ROR)) && (ADDRESS_MODE==ADDR_ACCUM))
                        )
                    begin
                    N_SEL = SEL_ALU_N;
                    Z_SEL = 1;
                    end
                else
                    begin
                    N_SEL = 0;
                    Z_SEL = 0;
                    end

                // V_SEL
                if (OP_CODE==OP_BIT)
                    V_SEL = SEL_M6;
                else if (OP_CODE==OP_CLV)
                    V_SEL = SEL_BIT_V;
                else if ((OP_CODE==OP_ADC) || (OP_CODE==OP_SBC))
                    V_SEL = SEL_ALU_V;
                else
                    V_SEL = 0;

                // C_SEL
                if ((OP_CODE==OP_CLC) || (OP_CODE==OP_SEC))
                    C_SEL = SEL_BIT_C;
                else if ((OP_CODE==OP_ADC) || (OP_CODE==OP_CMP) || (OP_CODE==OP_CPX) ||
(OP_CODE==OP_CPY) || (OP_CODE==OP_SBC) ||
                      (((OP_CODE==OP_ASL) || (OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) ||
(OP_CODE==OP_ROR)) && (ADDRESS_MODE==ADDR_ACCUM)))
                    C_SEL = SEL_ALU_C;
                else
                    C_SEL = 0;

            end
//------RUNNING: CYCLE 1  ------------------------------------------------
            else if (cycle==3'd1) begin
                AB_SEL = AB_PCX;                    // PC is on Address Bus
                RW = 1;                             // Read Operation
                INSTR_SEL = INSTR_SEL_HOLD;             // Hold Instruction

                case(ADDRESS_MODE)
                ADDR_ACCUM:
                    begin
                    PC_SEL = PC_HOLD;               // Hold PC
                    LATCH_INPUT = 0;
                    ADX_SEL = 0;
                    BAX_SEL = 0;
                    INSTR_COMPLETE = 1;                 // Instruction Complete
                    end
                ADDR_IMM:
```

```
    begin
    PC_SEL = PC_INC;                // Increment PC
    LATCH_INPUT = 1;                // Load Byte of Data
    ADX_SEL = 0;
    BAX_SEL = 0;
    INSTR_COMPLETE = 1;                 // Instruction Complete
    end
ADDR_ZERO:
    begin
    PC_SEL = PC_INC;                // Increment PC
    LATCH_INPUT = 0;
    ADX_SEL = ADR_LATCH_L;          // Load ADL
    BAX_SEL = 0;
    INSTR_COMPLETE = 0;
    end
ADDR_ZERO_X:
    begin
    PC_SEL = PC_INC;                // Increment PC
    LATCH_INPUT = 0;
    ADX_SEL = 0;
    BAX_SEL = ADR_LATCH_L;          // Load BAL
    INSTR_COMPLETE = 0;
    end
ADDR_ZERO_Y:
    begin
    PC_SEL = PC_INC;                // Increment PC
    LATCH_INPUT = 0;
    ADX_SEL = 0;
    BAX_SEL = ADR_LATCH_L;          // Load BAL
    INSTR_COMPLETE = 0;
    end
ADDR_ABS:
    begin
    PC_SEL = PC_INC;                // Increment PC
    LATCH_INPUT = 0;
    ADX_SEL = ADR_LATCH_L;          // Load ADL
    BAX_SEL = 0;
    INSTR_COMPLETE = 0;
    end
ADDR_ABS_X:
    begin
    PC_SEL = PC_INC;                // Increment PC
    LATCH_INPUT = 0;
    ADX_SEL = 0;
    BAX_SEL = ADR_LATCH_L;          // Load BAL
    INSTR_COMPLETE = 0;
    end
ADDR_ABS_Y:
    begin
    PC_SEL = PC_INC;                // Increment PC
    LATCH_INPUT = 0;
    ADX_SEL = 0;
    BAX_SEL = ADR_LATCH_L;          // Load BAL
    INSTR_COMPLETE = 0;
    end
ADDR_IMP:
    begin
    PC_SEL = PC_HOLD;               // Hold PC
    LATCH_INPUT = 0;
    ADX_SEL = 0;
    BAX_SEL = 0;
    if ((OP_CODE==OP_PHA)||
        (OP_CODE==OP_PHP)||
        (OP_CODE==OP_PLA)||
        (OP_CODE==OP_PLP)||
        (OP_CODE==OP_RTI)||
        (OP_CODE==OP_RTS)
        )   INSTR_COMPLETE = 0;         // Maybe Complete
    else    INSTR_COMPLETE = 1;
    end
ADDR_IND_X:
```

```
        begin
        PC_SEL = PC_INC;                // Increment PC
        LATCH_INPUT = 0;
        ADX_SEL = 0;
        BAX_SEL = ADR_LATCH_L;          // Load BAL
        INSTR_COMPLETE = 0;
        end
    ADDR_IND_Y:
        begin
        PC_SEL = PC_INC;                // Increment PC
        LATCH_INPUT = 0;
        ADX_SEL = ADR_LATCH_L;          // Load ADL
        BAX_SEL = 0;
        INSTR_COMPLETE = 0;
        end
    ADDR_INDIR:
        begin
        PC_SEL = PC_INC;                // Increment PC
        LATCH_INPUT = 0;
        ADX_SEL = 0;
        BAX_SEL = ADR_LATCH_L;          // Load BAL
        INSTR_COMPLETE = 0;
        end
    ADDR_SPECIAL:
        begin
        if (OP_CODE==OP_BRK)
            begin
            if (vector_mode==VM_NORMAL)
                PC_SEL = PC_INC;        // BRK
            else
                PC_SEL = PC_HOLD;
            end
        else
            PC_SEL = PC_HOLD;
        LATCH_INPUT = 0;
        ADX_SEL = 0;
        BAX_SEL = 0;
        INSTR_COMPLETE = 0;
        end
    ADDR_REL:
        begin
        LATCH_INPUT = 1;                // Latch Offset
        ADX_SEL = 0;
        BAX_SEL = 0;
        PC_SEL = PC_INC;
        case(OP_CODE)
            OP_BCC:
                INSTR_COMPLETE = ~(C==1'b0);  // Branch if C=0
            OP_BCS:
                INSTR_COMPLETE = ~(C==1'b1);  // Branch if C=1
            OP_BEQ:
                INSTR_COMPLETE = ~(Z==1'b1);  // Branch if Z=1
            OP_BMI:
                INSTR_COMPLETE = ~(N==1'b1);
            OP_BNE:
                INSTR_COMPLETE = ~(Z==1'b0);
            OP_BPL:
                INSTR_COMPLETE = ~(N==1'b0);
            OP_BVC:
                INSTR_COMPLETE = ~(V==1'b0);
            OP_BVS:
                INSTR_COMPLETE = ~(V==1'b1);
            default:
                INSTR_COMPLETE = 0;
        endcase
        end
    default:
        begin
        PC_SEL = PC_HOLD;               // Hold PC
        LATCH_INPUT = 0;
        ADX_SEL = 0;
```

```
                BAX_SEL = 0;
                INSTR_COMPLETE = 1;                  // Instruction Complete
                end
            endcase

            // Key Registers
            A_LOAD = 0;           // Do not affect Key Registers
            S_SEL = REG_HOLD;
            X_SEL = REG_HOLD;
            Y_SEL = REG_HOLD;

            // Status Register
            STATUS_LD = 0;        // Do not affect Status Register
            N_SEL = 0;
            V_SEL = 0;
            BT_CLEAR = 0;
            B_SEL = 0;
            D_SEL = 0;
            I_SEL = 0;
            Z_SEL = 0;
            C_SEL = 0;
            BIT_VAL = 0;

            // ALU_FN
            if ((ADDRESS_MODE==ADDR_SPECIAL) && (OP_CODE==OP_PHA))
                ALU_FN = FN_A;
            else if ((ADDRESS_MODE==ADDR_SPECIAL) && (OP_CODE==OP_PHP))
                ALU_FN = FN_STATUS;
            else if (OP_CODE==OP_BRK)
                ALU_FN = FN_A;
            else
                ALU_FN = 0;               // Always Pass Operand A

            // SEL_OPER_A
            if (OP_CODE==OP_STA)
                SEL_OPER_A = 3;
            else if (OP_CODE==OP_STX)
                SEL_OPER_A = 4;
            else if (OP_CODE==OP_STY)
                SEL_OPER_A = 5;
            else if (OP_CODE==OP_PHA)
                SEL_OPER_A = 3;
            else if (OP_CODE==OP_BRK)
                SEL_OPER_A = OPERAND_PCH;
            else
                SEL_OPER_A = 0;

            // SEL_OPER_B
            SEL_OPER_B = 0;

            // LATCH OUTPUT
            if ((OP_CODE==OP_STA) || (OP_CODE==OP_STX) || (OP_CODE==OP_STY))
                LATCH_OUTPUT = 1;
            else if ((ADDRESS_MODE==ADDR_SPECIAL) && ((OP_CODE==OP_PHA) || (OP_CODE==OP_PHP)))
                LATCH_OUTPUT = 1;
            else if (OP_CODE==OP_BRK)
                LATCH_OUTPUT = 1;
            else
                LATCH_OUTPUT = 0;

        end
//------RUNNING: CYCLE 2  ------------------------------------------------
        else if (cycle==3'd2) begin

            // AB_SEL
            if (ADDRESS_MODE==ADDR_ZERO)
                AB_SEL = AB_AD_ZERO;
            else if ((OP_CODE==OP_PHA) || (OP_CODE==OP_PHP))
                AB_SEL = AB_STACK;
            else if ((OP_CODE==OP_PLA) || (OP_CODE==OP_PLP))
                AB_SEL = AB_STACK;
```

```
                else if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_IND_X))
                    AB_SEL = AB_BA_ZERO;
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    AB_SEL = AB_AD_ZERO;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    AB_SEL = AB_STACK;
                else if (OP_CODE==OP_RTS)
                    AB_SEL = AB_STACK;
                else if (OP_CODE==OP_BRK)
                    AB_SEL = AB_STACK;
                else if (OP_CODE==OP_RTI)
                    AB_SEL = AB_STACK;
                else //(ADDR_ABS) || (ABS_X) || (ABS_Y)
                    AB_SEL = AB_PCX;

                // RW
                if ((ADDRESS_MODE==ADDR_ZERO) && ( (OP_CODE==OP_STA) || (OP_CODE==OP_STX) ||
(OP_CODE==OP_STY) ))
                    RW = 0;
                else if ((OP_CODE==OP_PHA) || (OP_CODE==OP_PHP))
                    RW = 0;
                else if (OP_CODE==OP_BRK)
                    RW = (vector_mode==VM_RESET);
                else if (OP_CODE==OP_RTI)
                    RW = 1;
                else
                    RW = 1;

                // INSTR_SEL
                INSTR_SEL = INSTR_SEL_HOLD;

                // LATCH_INPUT
                if (ADDRESS_MODE==ADDR_ZERO)
                    LATCH_INPUT = 1;
                else
                    LATCH_INPUT = 0;

                // PC_SEL
                if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y) ||
(ADDRESS_MODE==ADDR_INDIR))
                    PC_SEL = PC_INC;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    PC_SEL = PC_HOLD;
                else if (ADDRESS_MODE==ADDR_ABS)
                    PC_SEL = PC_INC;
                else if (ADDRESS_MODE==ADDR_REL)
                    PC_SEL = PC_LOAD_L;
                else
                    PC_SEL = PC_HOLD;

                // ALU_FN
                if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y) ||
                    (ADDRESS_MODE==ADDR_IND_X) || (ADDRESS_MODE==ADDR_IND_Y))
                    ALU_FN = FN_ADD_NC;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    ALU_FN = FN_A;
                else if (ADDRESS_MODE==ADDR_REL)
                    ALU_FN = FN_ADD_NC;
                else if (OP_CODE==OP_BRK)
                    ALU_FN = FN_B;
                else
                    ALU_FN = 0;

                // SEL_OPER_A
                if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ABS_X) ||
(ADDRESS_MODE==ADDR_IND_X))
                    SEL_OPER_A = OPERAND_X;
                else if ((ADDRESS_MODE==ADDR_ZERO_Y) || (ADDRESS_MODE==ADDR_ABS_Y))
                    SEL_OPER_A = OPERAND_Y;
```

```
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    SEL_OPER_A = OPERAND_ONE;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    SEL_OPER_A = OPERAND_PCH;
                else if (ADDRESS_MODE==ADDR_REL)
                    SEL_OPER_A = OPERAND_MEM;
                else
                    SEL_OPER_A = 0;

                // SEL_OPER_B
                if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y) ||
                    (ADDRESS_MODE==ADDR_IND_X))
                    SEL_OPER_B = OPERAND_BAL;
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    SEL_OPER_B = OPERAND_ADL;
                else if (ADDRESS_MODE==ADDR_REL)
                    SEL_OPER_B = OPERAND_PCL;
                else if (OP_CODE==OP_BRK)
                    SEL_OPER_B = OPERAND_PCL;
                else
                    SEL_OPER_B = 0;

                // LATCH OUTPUT
                if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    LATCH_OUTPUT = 1;
                else if (OP_CODE==OP_BRK)
                    LATCH_OUTPUT = 1;
                else
                    LATCH_OUTPUT = 0;

                // ADX_SEL
                if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE!=OP_JSR))
                    ADX_SEL = ADR_LATCH_H;            // LATCH ADH
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    ADX_SEL = ADR_LOAD_L;             // Load ADL
                else
                    ADX_SEL = 0;

                // BAX_SEL
                if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_IND_X))
                    BAX_SEL = ADR_LOAD_L;             // Result is in BAL
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    BAX_SEL = ADR_LATCH_L;            // Read in BAL
                else if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y))
                    BAX_SEL = ADR_ABS_INDEX;          // BAL gets R, BAH gets input data
                else if (ADDRESS_MODE==ADDR_INDIR)
                    BAX_SEL = ADR_LATCH_H;            // BAH gets data bus
                else
                    BAX_SEL = 0;

                // INSTRUCTION COMPLETE
                if (ADDRESS_MODE==ADDR_ZERO)
                    begin
                    if ((OP_CODE==OP_ASL) || (OP_CODE==OP_DEC) || (OP_CODE==OP_INC) ||
(OP_CODE==OP_LSR) ||
                        (OP_CODE==OP_ROL) || (OP_CODE==OP_ROR))
                        INSTR_COMPLETE = 0;
                    else
                        INSTR_COMPLETE = 1;
                    end
                else if ((OP_CODE==OP_PHA) || (OP_CODE==OP_PHP))
                    INSTR_COMPLETE = 1;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JMP))
                    INSTR_COMPLETE = 1;
                else if (ADDRESS_MODE==ADDR_REL)
                    INSTR_COMPLETE = (((aluC==1'b1) && (input_latch[7]==1'b1)) || ((aluC==1'b0) &&
(input_latch[7]==1'b0)));
                else  //(ADDR_ZERO_X) || (ADDR_ZERO_Y) || (ADDR_ABS) || (ADDR_ABS_INDEX)
                    INSTR_COMPLETE = 0;
```

```
                // Key Registers
                A_LOAD = 0;           // Do not affect Key Registers

                if ((OP_CODE==OP_PHA) || (OP_CODE==OP_PHP))
                    S_SEL = REG_DEC;
                else if ((OP_CODE==OP_PLA) || (OP_CODE==OP_PLP))
                    S_SEL = REG_INC;
                else if (OP_CODE==OP_RTS)
                    S_SEL = REG_INC;
                else if (OP_CODE==OP_BRK)
                    S_SEL = REG_DEC;
                else if (OP_CODE==OP_RTI)
                    S_SEL = REG_INC;
                else
                    S_SEL = REG_HOLD;

                X_SEL = REG_HOLD;
                Y_SEL = REG_HOLD;

                // Status Register
                STATUS_LD = 0;        // Do not affect Status Register
                N_SEL = 0;
                V_SEL = 0;
                BT_CLEAR = 0;
                B_SEL = 0;
                D_SEL = 0;
                I_SEL = 0;
                Z_SEL = 0;
                C_SEL = 0;
                BIT_VAL = 0;

            end
//------RUNNING: CYCLE 3  ------------------------------------------------
            else if (cycle==3'd3) begin

                // AB_SEL
                if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_IND_X))
                    AB_SEL = AB_BA_ZERO;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    AB_SEL = AB_STACK;
                else if (ADDRESS_MODE==ADDR_ABS)
                    AB_SEL = AB_ADX;
                else if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y) ||
(ADDRESS_MODE==ADDR_INDIR))
                    AB_SEL = AB_BAX;
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    AB_SEL = AB_AD_ZERO;
                else if ((OP_CODE==OP_PLA) || (OP_CODE==OP_PLP))
                    AB_SEL = AB_STACK;
                else if ((OP_CODE==OP_RTS) || (OP_CODE==OP_RTI))
                    AB_SEL = AB_STACK;
                else if (OP_CODE==OP_BRK)
                    AB_SEL = AB_STACK;
                else
                    AB_SEL = AB_PCX;

                // RW
                if (((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_ABS)) &&
                      ( (OP_CODE==OP_STA) || (OP_CODE==OP_STX) || (OP_CODE==OP_STY) ))
                    RW = 0;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    RW = 0;
                else if (OP_CODE==OP_BRK)
                    RW = (vector_mode==VM_RESET);
                else if (OP_CODE==OP_RTI)
                    RW = 1;
                else
                    RW = 1;
```

```
                // INSTR_SEL
                INSTR_SEL = INSTR_SEL_HOLD;

                // LATCH_INPUT
                if (((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_ABS) ||
                    (ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y)) &&
(previous_aluC==1'b0))
                    LATCH_INPUT = 1;
                else if ((OP_CODE==OP_PLA) || (OP_CODE==OP_PLP))
                    LATCH_INPUT = 1;
                else
                    LATCH_INPUT = 0;

                // PC_SEL
                if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y))
                    PC_SEL = PC_HOLD;
                else if (OP_CODE==OP_RTS)
                    PC_SEL = PC_LATCH_L;
                else if (ADDRESS_MODE==ADDR_REL)
                    PC_SEL = PC_LOAD_H;
                else
                    PC_SEL = PC_HOLD;

                // ADX_SEL
                if (ADDRESS_MODE==ADDR_IND_X)
                    ADX_SEL = ADR_LATCH_L;        // Latch ADL
                if (ADDRESS_MODE==ADDR_INDIR)
                    ADX_SEL = ADR_LATCH_L;
                else
                    ADX_SEL = 0;

                // BAX_SEL
                if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y))
                    BAX_SEL = ADR_LOAD_H;
                else if (ADDRESS_MODE==ADDR_IND_X)
                    BAX_SEL = ADR_LOAD_L;
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    BAX_SEL = ADR_ABS_INDEX;          // MBB gets data from bus, LSB gets result
from ALU
                else if (ADDRESS_MODE==ADDR_INDIR)
                    BAX_SEL = ADR_LOAD_L;
                else
                    BAX_SEL = 0;

                // INSTRUCTION COMPLETE
                if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    INSTR_COMPLETE = 0;
                else if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ||
(ADDRESS_MODE==ADDR_ABS))
                    begin
                    if ((OP_CODE==OP_ASL) || (OP_CODE==OP_DEC) || (OP_CODE==OP_INC) ||
(OP_CODE==OP_LSR) ||
                         (OP_CODE==OP_ROL) || (OP_CODE==OP_ROR))
                        INSTR_COMPLETE = 0;
                    else
                        INSTR_COMPLETE = 1;
                    end
                else if (((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y)) &&
                        ((OP_CODE==OP_ASL) || (OP_CODE==OP_DEC) || (OP_CODE==OP_INC) ||
(OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) || (OP_CODE==OP_ROR)))
                    INSTR_COMPLETE = 0;
                else if (((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y)) &&
(OP_CODE==OP_STA))
                    INSTR_COMPLETE = 0;
                else if (((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y)) &&
(OP_CODE!=OP_STA))
                    INSTR_COMPLETE = ~previous_aluC;
                else if (ADDRESS_MODE==ADDR_REL)
                    INSTR_COMPLETE = 1;
```

```
                else if ((OP_CODE==OP_PLP) || (OP_CODE==OP_PLA))
                    INSTR_COMPLETE = 1;
                else
                    INSTR_COMPLETE = 0;

                // Arithmetic Operation if one of the following Read-Write Instructions
                if ((ADDRESS_MODE==ADDR_ZERO) &&
                    ((OP_CODE==OP_ASL) || (OP_CODE==OP_DEC) || (OP_CODE==OP_INC) ||
(OP_CODE==OP_LSR) ||
                     (OP_CODE==OP_ROL) || (OP_CODE==OP_ROR)))
                    begin
                    // ALU_FN
                    if (OP_CODE==OP_ASL)
                        ALU_FN = 7;
                    else if (OP_CODE==OP_LSR)
                        ALU_FN = 8;
                    else if (OP_CODE==OP_ROL)
                        ALU_FN = 9;
                    else if (OP_CODE==OP_ROR)
                        ALU_FN = 10;
                    else if (OP_CODE==OP_INC)
                        ALU_FN = 11;
                    else if (OP_CODE==OP_DEC)
                        ALU_FN = 12;
                    else
                        ALU_FN = 0;

                    // SEL_OPER_A
                    SEL_OPER_A = 2;

                    // SEL_OPER_B
                    if ((OP_CODE==OP_DEC) || (OP_CODE==OP_INC))
                        SEL_OPER_B = 1;
                    else
                        SEL_OPER_B = 0;

                    // Status Register
                    STATUS_LD = 0;
                    BIT_VAL = 0;
                    BT_CLEAR = 0;
                    B_SEL = 0;
                    D_SEL = 0;
                    I_SEL = 0;
                    // N_SEL and Z_SEL
                    N_SEL = SEL_ALU_N;
                    Z_SEL = 1;
                    // V_SEL
                    V_SEL = 0;
                    // C_SEL
                    if ((OP_CODE==OP_ASL) || (OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) ||
(OP_CODE==OP_ROR))
                        C_SEL = SEL_ALU_C;
                    else
                        C_SEL = 0;

                    // LATCH_OUTPUT
                    LATCH_OUTPUT = 1;

                    end
                else
                    begin
                    // ALU
                    if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y))
                        ALU_FN = FN_ADD_NC;
                    else if ((ADDRESS_MODE==ADDR_IND_X) || (ADDRESS_MODE==ADDR_IND_Y))
                        ALU_FN = FN_ADD_NC;
                    else if (ADDRESS_MODE==ADDR_INDIR)
                        ALU_FN = FN_ADD_NC;
                    else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                        ALU_FN = FN_B;
                    else if (ADDRESS_MODE==ADDR_REL)
```

```
                    if ((previous_aluC==1'b1) && (input_latch[7]==1'b0))
                        ALU_FN = FN_ADD_NC;
                    else if ((previous_aluC==1'b0) && (input_latch[7]==1'b1))
                        ALU_FN = FN_SUB_NB;
                    else
                        ALU_FN = FN_A;
                else if (OP_CODE==OP_BRK)
                    ALU_FN = FN_STATUS;
                else
                    ALU_FN = 0;

                // SEL_OPER_A
                if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y))
                    if (previous_aluC==1)
                        SEL_OPER_A = OPERAND_ONE;
                    else
                        SEL_OPER_A = OPERAND_ZERO;
                else if ((ADDRESS_MODE==ADDR_IND_X) || (ADDRESS_MODE==ADDR_INDIR))
                    SEL_OPER_A = OPERAND_ONE;
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    SEL_OPER_A = OPERAND_Y;
                else if (ADDRESS_MODE==ADDR_REL)
                    SEL_OPER_A = OPERAND_PCH;
                else
                    SEL_OPER_A = 0;

                // SEL_OPER_B
                if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y))
                    SEL_OPER_B = OPERAND_BAH;
                else if ((ADDRESS_MODE==ADDR_IND_X) || (ADDRESS_MODE==ADDR_IND_Y) ||
(ADDRESS_MODE==ADDR_INDIR))
                    SEL_OPER_B = OPERAND_BAL;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    SEL_OPER_B = OPERAND_PCL;
                else if (ADDRESS_MODE==ADDR_REL)
                    SEL_OPER_B = OPERAND_ONE;
                else
                    SEL_OPER_B = 0;

                // Status Register
                if (OP_CODE==OP_RTI)
                    STATUS_LD = 1;
                else
                    STATUS_LD = 0;        // Do not affect Status Register

                N_SEL = 0;
                V_SEL = 0;
                BT_CLEAR = 0;
                B_SEL = 0;
                D_SEL = 0;
                I_SEL = 0;
                Z_SEL = 0;
                C_SEL = 0;
                BIT_VAL = 0;

                // LATCH_OUTPUT
                if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    LATCH_OUTPUT = 1;
                else if (OP_CODE==OP_BRK)
                    LATCH_OUTPUT = 1;
                else
                    LATCH_OUTPUT = 0;
            end

            // Key Registers
            A_LOAD = 0;           // Do not affect Key Registers

            if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                S_SEL = REG_DEC;
            else if (OP_CODE==OP_RTS)
                S_SEL = REG_INC;
```

```
                else if (OP_CODE==OP_BRK)
                    S_SEL = REG_DEC;
                else if (OP_CODE==OP_RTI)
                    S_SEL = REG_INC;
                else
                    S_SEL = REG_HOLD;

                X_SEL = REG_HOLD;
                Y_SEL = REG_HOLD;

            end
//------RUNNING: CYCLE 4  ------------------------------------------------
            else if (cycle==3'd4) begin

                // AB_SEL
                if (ADDRESS_MODE==ADDR_ZERO)
                    AB_SEL = AB_AD_ZERO;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    AB_SEL = AB_STACK;
                else if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y) ||
(ADDRESS_MODE==ADDR_IND_Y) || (ADDRESS_MODE==ADDR_INDIR))
                    AB_SEL = AB_BAX;
                else if (ADDRESS_MODE==ADDR_IND_X)
                    AB_SEL = AB_BA_ZERO;
                else if ((OP_CODE==OP_RTS) || (OP_CODE==OP_RTI))
                    AB_SEL = AB_STACK;
                else if (OP_CODE==OP_BRK)
                    AB_SEL = AB_STACK;
                else
                    AB_SEL = AB_PCX;

                // RW
                if (ADDRESS_MODE == ADDR_ZERO)
                    RW = 0;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    RW = 0;
                else if (((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y)) &&
                        ( (OP_CODE==OP_STA) || (OP_CODE==OP_STX) || (OP_CODE==OP_STY) ))
                    RW = 0;
                else if (OP_CODE==OP_BRK)
                    RW = (vector_mode==VM_RESET);
                else
                    RW = 1;

                // INSTR_SEL
                INSTR_SEL = INSTR_SEL_HOLD;

                // LATCH_INPUT
                if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y) ||
(ADDRESS_MODE==ADDR_IND_Y))
                    LATCH_INPUT = 1;
                else
                    LATCH_INPUT = 0;

                // PC_SEL
                if (OP_CODE==OP_RTS)
                    PC_SEL = PC_LATCH_H;
                else if (OP_CODE==OP_RTI)
                    PC_SEL = PC_LATCH_L;
                else
                    PC_SEL = PC_HOLD;

                // ADX_SEL
                if (ADDRESS_MODE==ADDR_IND_X)
                    ADX_SEL = ADR_LATCH_H;
                else if (ADDRESS_MODE==ADDR_INDIR)
                    ADX_SEL = ADR_LATCH_H;
                else
                    ADX_SEL = 0;

                // BAX_SEL
```

```
                if (ADDRESS_MODE==ADDR_IND_Y)
                    BAX_SEL = ADR_LOAD_H;
                else
                    BAX_SEL = 0;

                // INSTRUCTION COMPLETE
                if ((ADDRESS_MODE==ADDR_ZERO) || (ADDRESS_MODE==ADDR_ABS_X) ||
(ADDRESS_MODE==ADDR_ABS_Y))
                    if ((OP_CODE==OP_ASL) || (OP_CODE==OP_DEC) || (OP_CODE==OP_INC) ||
(OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) || (OP_CODE==OP_ROR))
                        INSTR_COMPLETE = 0;
                    else
                        INSTR_COMPLETE = 1;
                else if (ADDRESS_MODE==ADDR_IND_Y)
                    if (OP_CODE==OP_STA)
                        INSTR_COMPLETE = 0;                  // Because writing to the wrong
location would be very bad
                    else
                        INSTR_COMPLETE = ~previous_aluC;
                else if (ADDRESS_MODE==ADDR_INDIR)
                    INSTR_COMPLETE = 1;
                else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                    INSTR_COMPLETE = 0;
                else
                    INSTR_COMPLETE = 0;

                // Arithmetic Operation if one of the following Read-Write Instructions
                if (((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_Y) ) &&
                    ((OP_CODE==OP_ASL) || (OP_CODE==OP_DEC) || (OP_CODE==OP_INC) ||
(OP_CODE==OP_LSR) ||
                     (OP_CODE==OP_ROL) || (OP_CODE==OP_ROR)))
                    begin
                    // ALU_FN
                    if (OP_CODE==OP_ASL)
                        ALU_FN = 7;
                    else if (OP_CODE==OP_LSR)
                        ALU_FN = 8;
                    else if (OP_CODE==OP_ROL)
                        ALU_FN = 9;
                    else if (OP_CODE==OP_ROR)
                        ALU_FN = 10;
                    else if (OP_CODE==OP_INC)
                        ALU_FN = 11;
                    else if (OP_CODE==OP_DEC)
                        ALU_FN = 12;
                    else
                        ALU_FN = 0;
                    // SEL_OPER_A
                    SEL_OPER_A = 2;
                    // SEL_OPER_B
                    if ((OP_CODE==OP_DEC) || (OP_CODE==OP_INC))
                        SEL_OPER_B = 1;
                    else
                        SEL_OPER_B = 0;

                    // Status Register
                    STATUS_LD = 0;
                    BIT_VAL = 0;
                    BT_CLEAR = 0;
                    B_SEL = 0;
                    D_SEL = 0;
                    I_SEL = 0;
                    // N_SEL and Z_SEL
                    N_SEL = SEL_ALU_N;
                    Z_SEL = 1;
                    // V_SEL
                    V_SEL = 0;
                    // C_SEL
                    if ((OP_CODE==OP_ASL) || (OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) ||
(OP_CODE==OP_ROR))
                        C_SEL = SEL_ALU_C;
```

```
                else
                    C_SEL = 0;

                // LATCH_OUTPUT
                LATCH_OUTPUT = 1;

                end
            else
                begin
                // ALU
                if (ADDRESS_MODE==ADDR_IND_Y)
                    ALU_FN = FN_ADD_NC;
                else
                    ALU_FN = 0;

                if (ADDRESS_MODE==ADDR_IND_Y)
                    if (previous_aluC)
                        SEL_OPER_A = OPERAND_ONE;
                    else
                        SEL_OPER_A = OPERAND_ZERO;
                else
                    SEL_OPER_A = 0;

                if (ADDRESS_MODE==ADDR_IND_Y)
                    SEL_OPER_B = OPERAND_BAH;
                else
                    SEL_OPER_B = 0;

                // Status Register
                STATUS_LD = 0;        // Do not affect Status Register
                N_SEL = 0;
                V_SEL = 0;
                BT_CLEAR = 0;
                B_SEL = 0;
                D_SEL = 0;
                I_SEL = 0;
                Z_SEL = 0;
                C_SEL = 0;
                BIT_VAL = 0;

                // LATCH_OUTPUT
                LATCH_OUTPUT = 0;
            end

            // Key Registers
            A_LOAD = 0;           // Do not affect Key Registers

            if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                S_SEL = REG_DEC;
            else if (OP_CODE==OP_BRK)
                S_SEL = REG_DEC;
            else if (OP_CODE==OP_RTI)
                S_SEL = REG_INC;
            else
                S_SEL = REG_HOLD;

            X_SEL = REG_HOLD;
            Y_SEL = REG_HOLD;

        end
//------RUNNING: CYCLE 5  ------------------------------------------------
        else if (cycle==3'd5) begin

            // AB_SEL
            if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_X))
                AB_SEL = AB_BA_ZERO;
            else if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                AB_SEL = AB_PCX;
            else if ((ADDRESS_MODE==ADDR_ABS) || (ADDRESS_MODE==ADDR_IND_X))
                AB_SEL = AB_ADX;
            else if (ADDRESS_MODE==ADDR_IND_Y)
```

```
                AB_SEL = AB_BAX;
            else if (OP_CODE==OP_BRK)
                case(vector_mode)
                VM_NORMAL:
                    AB_SEL = AB_IRQ_LOW;      //{FF,FE}
                VM_RESET:
                    AB_SEL = AB_RESET_LOW;    //{FF,FC}
                VM_IRQ:
                    AB_SEL = AB_IRQ_LOW;      //{FF,FE}
                VM_NMI:
                    AB_SEL = AB_NMI_LOW;      //{FF,FA}
                endcase
            else if (OP_CODE==OP_RTI)
                AB_SEL = AB_STACK;
            else
                AB_SEL = AB_PCX;

            // RW
            if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                RW = 1;
            else if ((ADDRESS_MODE==ADDR_ZERO_X) || (ADDRESS_MODE==ADDR_ZERO_X) ||
(ADDRESS_MODE==ADDR_ABS))
                RW = 0;
            else if ((ADDRESS_MODE==ADDR_IND_Y) && (OP_CODE==OP_STA))
                RW = 0;
            else if ((ADDRESS_MODE==ADDR_IND_X) && (OP_CODE==OP_STA))
                RW = 0;
            else if (OP_CODE==OP_BRK)
                RW = 1;
            else
                RW = 1;

            // INSTR_SEL
            INSTR_SEL = INSTR_SEL_HOLD;

            // LATCH_INPUT
            if ((ADDRESS_MODE==ADDR_IND_Y) && (OP_CODE!=OP_STA))
                LATCH_INPUT = 1;
            else if ((ADDRESS_MODE==ADDR_IND_X) && (OP_CODE!=OP_STA))
                LATCH_INPUT = 1;
            else
                LATCH_INPUT = 0;

            // PC_SEL
            if (OP_CODE==OP_RTS)
                PC_SEL = PC_INC;
            else if (OP_CODE==OP_BRK)
                PC_SEL = PC_LATCH_L;
            else if (OP_CODE==OP_RTI)
                PC_SEL = PC_LATCH_H;
            else
                PC_SEL = PC_HOLD;

            // ADX_SEL
            if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                ADX_SEL = ADR_LATCH_H;
            else
                ADX_SEL = 0;

            // BAX_SEL
            BAX_SEL = 0;

            // INSTRUCTION COMPLETE
            if ((ADDRESS_MODE==ADDR_ABS) && (OP_CODE==OP_JSR))
                INSTR_COMPLETE = 1;
            else if (OP_CODE==OP_RTS)
                INSTR_COMPLETE = 1;
            else if (OP_CODE==OP_RTI)
                INSTR_COMPLETE = 1;
            else
                INSTR_COMPLETE = 0;
```

```
                // Arithmetic Operation if one of the following Read-Write Instructions
                if (((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y) ) &&
                    ((OP_CODE==OP_ASL) || (OP_CODE==OP_DEC) || (OP_CODE==OP_INC) ||
(OP_CODE==OP_LSR) ||
                     (OP_CODE==OP_ROL) || (OP_CODE==OP_ROR)))
                    begin
                    // ALU_FN
                    if (OP_CODE==OP_ASL)
                        ALU_FN = 7;
                    else if (OP_CODE==OP_LSR)
                        ALU_FN = 8;
                    else if (OP_CODE==OP_ROL)
                        ALU_FN = 9;
                    else if (OP_CODE==OP_ROR)
                        ALU_FN = 10;
                    else if (OP_CODE==OP_INC)
                        ALU_FN = 11;
                    else if (OP_CODE==OP_DEC)
                        ALU_FN = 12;
                    else
                        ALU_FN = 0;
                    // SEL_OPER_A
                    SEL_OPER_A = 2;
                    // SEL_OPER_B
                    if ((OP_CODE==OP_DEC) || (OP_CODE==OP_INC))
                        SEL_OPER_B = 1;
                    else
                        SEL_OPER_B = 0;

                    // Status Register
                    STATUS_LD = 0;
                    BIT_VAL = 0;
                    BT_CLEAR = 0;
                    B_SEL = 0;
                    D_SEL = 0;
                    I_SEL = 0;
                    // N_SEL and Z_SEL
                    N_SEL = SEL_ALU_N;
                    Z_SEL = 1;
                    // V_SEL
                    V_SEL = 0;
                    // C_SEL
                    if ((OP_CODE==OP_ASL) || (OP_CODE==OP_LSR) || (OP_CODE==OP_ROL) ||
(OP_CODE==OP_ROR))
                        C_SEL = SEL_ALU_C;
                    else
                        C_SEL = 0;

                    // LATCH_OUTPUT
                    LATCH_OUTPUT = 1;

                    end
                else
                    begin
                    // ALU
                    ALU_FN = 0;
                    SEL_OPER_A = 0;
                    SEL_OPER_B = 0;

                    // Status Register
                    STATUS_LD = 0;        // Do not affect Status Register
                    N_SEL = 0;
                    V_SEL = 0;
                    BT_CLEAR = 0;
                    B_SEL = 0;
                    D_SEL = 0;
                    Z_SEL = 0;
                    C_SEL = 0;
```

```
                if ((OP_CODE==OP_BRK) && ((vector_mode==VM_IRQ) || (vector_mode==VM_NMI)))
                    begin
                    I_SEL = 1;
                    BIT_VAL = 1;
                    end
                else
                    begin
                    I_SEL = 0;
                    BIT_VAL = 0;
                    end


                // LATCH_OUTPUT
                LATCH_OUTPUT = 0;
            end

            // Key Registers
            A_LOAD = 0;           // Do not affect Key Registers
            S_SEL = REG_HOLD;
            X_SEL = REG_HOLD;
            Y_SEL = REG_HOLD;

        end
//------RUNNING: CYCLE 6  ------------------------------------------------
        else if (cycle==3'd6) begin

            // AB_SEL
            if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y))
                AB_SEL = AB_BAX;
            else if (OP_CODE==OP_BRK)
                case(vector_mode)
                VM_NORMAL:
                    AB_SEL = AB_IRQ_HIGH;     //{FF,FF}
                VM_RESET:
                    AB_SEL = AB_RESET_HIGH;       //{FF,FD}
                VM_IRQ:
                    AB_SEL = AB_IRQ_HIGH;     //{FF,FF}
                VM_NMI:
                    AB_SEL = AB_NMI_HIGH;     //{FF,FB}
                endcase
            else
                AB_SEL = AB_PCX;

            // RW
            if ((ADDRESS_MODE==ADDR_ABS_X) || (ADDRESS_MODE==ADDR_ABS_Y))
                RW = 0;
            else if (OP_CODE==OP_BRK)
                RW = 1;
            else
                RW = 1;

            // INSTR_SEL
            INSTR_SEL = INSTR_SEL_HOLD;

            // LATCH_INPUT
            LATCH_INPUT = 0;

            // LATCH_OUTPUT
            LATCH_OUTPUT = 0;

            // PC_SEL
            if (OP_CODE==OP_BRK)
                PC_SEL = PC_LATCH_H;
            else
                PC_SEL = PC_HOLD;

            // ADX_SEL
            ADX_SEL = 0;

            // BAX_SEL
            BAX_SEL = 0;
```

```
                // INSTRUCTION COMPLETE
                INSTR_COMPLETE = 1;

                // Key Registers
                A_LOAD = 0;           // Do not affect Key Registers
                S_SEL = REG_HOLD;
                X_SEL = REG_HOLD;
                Y_SEL = REG_HOLD;

                // Status Register
                STATUS_LD = 0;        // Do not affect Status Register
                N_SEL = 0;
                V_SEL = 0;
                BT_CLEAR = 0;
                D_SEL = 0;
                I_SEL = 0;
                Z_SEL = 0;
                C_SEL = 0;
                if (OP_CODE==OP_BRK)
                    if (vector_mode==VM_NORMAL)
                        begin
                        B_SEL = 1;
                        BIT_VAL = 1;
                        end
                    else
                        begin
                        B_SEL = 1;
                        BIT_VAL = 0;
                        end
                else
                    begin
                    B_SEL = 0;
                    BIT_VAL = 0;
                    end

                // Arithmetic Operations
                ALU_FN = 0;           // Don't Care About Arithmetic Operation
                SEL_OPER_A = 0;
                SEL_OPER_B = 0;


            end
//--------------------------------------------------------------------------
            else begin
                AB_SEL = AB_PCX;                // PC is on Address Bus
                RW = 1;                         // Read Operation
                INSTR_SEL = INSTR_SEL_HOLD;          // Hold Instruction
                LATCH_OUTPUT = 0;
                PC_SEL = PC_HOLD;               // Hold PC
                LATCH_INPUT = 0;
                ADX_SEL = 0;
                BAX_SEL = 0;
                INSTR_COMPLETE = 1;                 // Instruction Complete

                // Key Registers
                A_LOAD = 0;           // Do not affect Key Registers
                S_SEL = REG_HOLD;
                X_SEL = REG_HOLD;
                Y_SEL = REG_HOLD;

                // Status Register
                STATUS_LD = 0;        // Do not affect Status Register
                N_SEL = 0;
                V_SEL = 0;
                BT_CLEAR = 0;
                B_SEL = 0;
                D_SEL = 0;
                I_SEL = 0;
                Z_SEL = 0;
                C_SEL = 0;
```

```
                BIT_VAL = 0;

                // Arithmetic Operations
                ALU_FN = 0;           // Don't Care About Arithmetic Operation
                SEL_OPER_A = 0;
                SEL_OPER_B = 0;
            end
//-------------------------------------------------------------------------
        end
    end

//=========================================================================



//=========================================================================
//  ALU - Including Operand Selection
//-------------------------------------------------------------------------
    always@(SEL_OPER_A or input_latch or A or X or Y or S or PCH) begin
        case(SEL_OPER_A)
        OPERAND_ZERO:
            operand_a = 8'h00;
        OPERAND_ONE:
            operand_a = 8'h01;
        OPERAND_MEM:
            operand_a = input_latch;
        OPERAND_ACCUM:
            operand_a = A;
        OPERAND_X:
            operand_a = X;
        OPERAND_Y:
            operand_a = Y;
        OPERAND_S:
            operand_a = S;
        OPERAND_PCH:
            operand_a = PCH;
        endcase
    end

    always@(SEL_OPER_B or input_latch or BAH or BAL or ADH or ADL or PCL) begin
        case(SEL_OPER_B)
        OPERAND_ZERO:
            operand_b = 8'h00;
        OPERAND_ONE:
            operand_b = 8'h01;
        OPERAND_MEM:
            operand_b = input_latch;
        OPERAND_BAH:
            operand_b = BAH;
        OPERAND_BAL:
            operand_b = BAL;
        OPERAND_ADH:
            operand_b = ADH;
        OPERAND_ADL:
            operand_b = ADL;
        OPERAND_PCL:
            operand_b = PCL;
        endcase
    end


    alu  ALU(
        .alu_fn (ALU_FN),
        .A (operand_a),
        .B (operand_b),
        .Cin (C),
        .R (R_out),
        .Cout (aluC),
        .Zout (aluZ),
        .Vout (aluV),
        .Nout (aluN)
```

```
    );

    // Overide...
    always@(R_out or ALU_FN or STATUS) begin
        if (ALU_FN==FN_STATUS)
            R = STATUS;
        else
            R = R_out;
    end

    always@(posedge clk1) begin
        previous_aluC <= aluC;
    end
//=============================================================================


//=============================================================================
//  STATUS Register - Individual Bits have individual functions
//-----------------------------------------------------------------------------
    always@(posedge clk1) begin
        if (STATUS_LD) begin
            N <= data_bus[7];
            V <= data_bus[6];
            BT <= BT;
            B <= data_bus[4];
            D <= data_bus[3];
            I <= data_bus[2];
            Z <= data_bus[1];
            C <= data_bus[0];
        end
        else begin
            if (N_SEL == SEL_ALU_N)       // N - Negative bit
                N <= aluN;
            else if (N_SEL == SEL_M7)
                N <= input_latch[7];
            else
                N <= N;

            if (V_SEL == SEL_ALU_V)
                V <= aluV;            // V - Overflow bit
            else if (V_SEL == SEL_BIT_V)
                V <= BIT_VAL;
            else if (V_SEL == SEL_M6)
                V <= input_latch[6];
            else
                V <= V;

            if (MCLR)
                BT <= BOOT_ON_RESET;
            else if (BT_CLEAR)        // BT - Boot bit
                BT <= 0;
            else
                BT <= BT;

            if (MCLR)                 // B - Break Command bit
                B <= 0;
            else if (B_SEL)
                B <= BIT_VAL;
            else
                B <= B;

            if (D_SEL)                // D - Decimal bit
                D <= BIT_VAL;
            else
                D <= D;

            if (MCLR)                 // I - Interrupt Disable bit
                I <= 1'b1;            //     Int. Disabled on Startup
            else if (I_SEL)
                I <= BIT_VAL;
            else
```

```
                I <= I;

            if (Z_SEL)                // Z - Zero bit
                Z <= aluZ;
            else
                Z <= Z;

            if (C_SEL == SEL_ALU_C)       // C - Carry bit
                C <= aluC;
            else if (C_SEL == SEL_BIT_C)
                C <= BIT_VAL;
            else
                C <= C;
        end
    end

    assign STATUS = {N, V, BT, B, D, I, Z, C};

//=============================================================================


//=============================================================================
//  Program Counter
//-----------------------------------------------------------------------------
    always@(posedge clk1) begin
        // PCH
        if (MCLR)
            PCH <= 8'h80;
        else begin
            case (PC_SEL)
            PC_HOLD:
                PCH <= PCH;
            PC_INC:
                PCH <= PC_PLUS[15:8];
            PC_LATCH_H:
                PCH <= data_bus;
            PC_LATCH_L:
                PCH <= PCH;
            PC_LOAD_H:
                PCH <= R;
            PC_LOAD_L:
                PCH <= PCH;
            PC_LATCH_ADX:
                PCH <= ADH;
            PC_LATCH_ADXP:
                PCH <= ADX_PLUS[15:8];
            endcase
        end

        // PCL
        if (MCLR)
            PCL <= 8'h00;
        else begin
            case (PC_SEL)
            PC_HOLD:
                PCL <= PCL;
            PC_INC:
                PCL <= PC_PLUS[7:0];
            PC_LATCH_H:
                PCL <= PCL;
            PC_LATCH_L:
                PCL <= data_bus;
            PC_LOAD_H:
                PCL <= PCL;
            PC_LOAD_L:
                PCL <= R;
            PC_LATCH_ADX:
                PCL <= ADL;
            PC_LATCH_ADXP:
                PCL <= ADX_PLUS[7:0];
            endcase
```

```
        end
    end

    assign PC = {PCH, PCL};
    assign PC_PLUS = PC + 1;
    assign ADX_PLUS = {ADH,ADL} + 1;

//=============================================================================


//=============================================================================
//  Key Registers: Accumulator, Stack, X and Y
//-----------------------------------------------------------------------------
    always@(posedge clk1) begin
        if (A_LOAD)                   // Accumulator
            A <= R;
        else
            A <= A;

        if (S_SEL == REG_HOLD)        // Stack Pointer
            S <= S;
        else if (S_SEL == REG_INC)
            S <= S + 1;
        else if (S_SEL == REG_DEC)
            S <= S - 1;
        else
            S <= R;

        if (X_SEL == REG_HOLD)        // X Index
            X <= X;
        else if (X_SEL == REG_INC)
            X <= X + 1;
        else if (X_SEL == REG_DEC)
            X <= X - 1;
        else
            X <= R;

        if (Y_SEL == REG_HOLD)        // Y Index
            Y <= Y;
        else if (Y_SEL == REG_INC)
            Y <= Y + 1;
        else if (Y_SEL == REG_DEC)
            Y <= Y - 1;
        else
            Y <= R;
    end
//=============================================================================

//=============================================================================
//  Address Bus Buffer - always valid
//-----------------------------------------------------------------------------

    // ADL and ADH
    always@(posedge clk1) begin
        case(ADX_SEL)
        ADR_HOLD:
            begin
            ADH<=ADH;
            ADL<=ADL;
            end
        ADR_LATCH_L:
            begin
            ADH<=ADH;
            ADL<=data_bus;
            end
        ADR_LATCH_H:
            begin
            ADH<=data_bus;
            ADL<=ADL;
            end
        ADR_LOAD_L:
```

```
        begin
        ADH<=ADH;
        ADL<=R;
        end
    ADR_LOAD_H:
        begin
        ADH<=R;
        ADL<=ADL;
        end
    ADR_ABS_INDEX:
        begin
        ADH <= data_bus;
        ADL <= R;
        end
    endcase
end

// BAL and BAH
always@(posedge clk1) begin
    case(BAX_SEL)
    ADR_HOLD:
        begin
        BAH <= BAH;
        BAL <= BAL;
        end
    ADR_LATCH_L:
        begin
        BAH <= BAH;
        BAL <= data_bus;
        end
    ADR_LATCH_H:
        begin
        BAH <= data_bus;
        BAL <= BAL;
        end
    ADR_LOAD_L:
        begin
        BAH <= BAH;
        BAL <= R;
        end
    ADR_LOAD_H:
        begin
        BAH <= R;
        BAL <= BAL;
        end
    ADR_ABS_INDEX:
        begin
        BAH <= data_bus;
        BAL <= R;
        end
    endcase
end

// Address Bus
always@(AB_SEL or PCH or PCL or ADH or ADL or BAH or BAL or S) begin
    case(AB_SEL)
    AB_PCX:
        begin
        ABH = PCH;
        ABL = PCL;
        end
    AB_ADX:
        begin
        ABH = ADH;
        ABL = ADL;
        end
    AB_BAX:
        begin
        ABH = BAH;
        ABL = BAL;
        end
```

```
        AB_AD_ZERO:
            begin
            ABH = ZERO_PAGE;
            ABL = ADL;
            end
        AB_BA_ZERO:
            begin
            ABH = ZERO_PAGE;
            ABL = BAL;
            end
        AB_STACK:
            begin
            ABH = STACK_PAGE;
            ABL = S;
            end
        AB_NMI_LOW:
            begin
            ABH = VECTOR_PAGE;
            ABL = NMI_VECTOR_L;
            end
        AB_NMI_HIGH:
            begin
            ABH = VECTOR_PAGE;
            ABL = NMI_VECTOR_H;
            end
        AB_RESET_LOW:
            begin
            ABH = VECTOR_PAGE;
            ABL = RESET_VECTOR_L;
            end
        AB_RESET_HIGH:
            begin
            ABH = VECTOR_PAGE;
            ABL = RESET_VECTOR_H;
            end
        AB_IRQ_LOW:
            begin
            ABH = VECTOR_PAGE;
            ABL = IRQ_VECTOR_L;
            end
        AB_IRQ_HIGH:
            begin
            ABH = VECTOR_PAGE;
            ABL = IRQ_VECTOR_H;
            end
        endcase
    end

    assign address_bus = {ABH,ABL};

//=============================================================================


//=============================================================================
//  Data Bus Buffers - Data Bus is driven only while RW is low and clk2 is high
//                   - Data bus value can be latched into Input
//                   - Instruction Register is loaded from data bus
//-----------------------------------------------------------------------------
    always@(posedge clk1) begin
        if (LATCH_OUTPUT)
            output_latch <= R;
        else
            output_latch <= output_latch;
    end

    always@(posedge clk1) begin
        if (LATCH_INPUT)
            input_latch <= data_bus;
        else
            input_latch <= input_latch;
        if (INSTR_SEL == INSTR_SEL_HOLD)
```

```
            instr <= instr;
        else if (INSTR_SEL == INSTR_SEL_LOAD)
            instr <= data_bus;
        else // (INSTR_SEL == INSTR_SEL_FORCE_BRK)
            instr <= 8'h00;    // Break Instruction to service Interrupts
    end

    assign data_bus_out = (~RW && clk2) ? output_latch : 8'hFF;
//=========================================================================

Endmodule
```

Instr_Decode.v

```
module instr_decode(instr, OP_CODE);
    input  [7:0] instr;
    output [5:0] OP_CODE;

    // OP_CODES
    parameter OP_NOP =    0;
    parameter OP_ADC = 1;
    parameter OP_AND = 2;
    parameter OP_ASL = 3;
    parameter OP_BCC = 4;
    parameter OP_BCS = 5;
    parameter OP_BEQ = 6;
    parameter OP_BIT = 7;
    parameter OP_BMI = 8;
    parameter OP_BNE = 9;
    parameter OP_BPL = 10;
    parameter OP_BRK = 11;
    parameter OP_BVC = 12;
    parameter OP_BVS = 13;
    parameter OP_CLC = 14;
    parameter OP_CLD = 15;
    parameter OP_CLI = 16;
    parameter OP_CLV = 17;
    parameter OP_CMP = 18;
    parameter OP_CPX = 19;
    parameter OP_CPY = 20;
    parameter OP_DEC = 21;
    parameter OP_DEX = 22;
    parameter OP_DEY = 23;
    parameter OP_EOR = 24;
    parameter OP_INC = 25;
    parameter OP_INX = 26;
    parameter OP_INY = 27;
    parameter OP_JMP = 28;
    parameter OP_JSR = 29;
    parameter OP_LDA = 30;
    parameter OP_LDX = 31;
    parameter OP_LDY = 32;
    parameter OP_LSR = 33;
    parameter OP_ORA = 34;
    parameter OP_PHA = 35;
    parameter OP_PHP = 36;
    parameter OP_PLA = 37;
    parameter OP_PLP = 38;
    parameter OP_ROL = 39;
    parameter OP_ROR = 40;
    parameter OP_RTI = 41;
    parameter OP_RTS = 42;
    parameter OP_SBC = 43;
    parameter OP_SEC = 44;
    parameter OP_SED = 45;
    parameter OP_SEI = 46;
    parameter OP_STA = 47;
```

```
parameter OP_STX = 48;
parameter OP_STY = 49;
parameter OP_TAX = 50;
parameter OP_TAY = 51;
parameter OP_TYA = 52;
parameter OP_TSX = 53;
parameter OP_TXA = 54;
parameter OP_TXS = 55;
parameter OP_EBT = 56;

// Decode of Instruction into Operand and Addressing (Combinational Logic)
reg [5:0] OP_CODE;
reg [3:0] ADDRESS_MODE;

always@(instr) begin
 // ADC Instruction
    if (instr==8'h69)  OP_CODE = OP_ADC;
    else if (instr==8'h65)  OP_CODE = OP_ADC;
    else if (instr==8'h75)  OP_CODE = OP_ADC;
    else if (instr==8'h6d)  OP_CODE = OP_ADC;
    else if (instr==8'h7d)  OP_CODE = OP_ADC;
    else if (instr==8'h79)  OP_CODE = OP_ADC;
    else if (instr==8'h61)  OP_CODE = OP_ADC;
    else if (instr==8'h71)  OP_CODE = OP_ADC;
// AND Instruction
    else if (instr==8'h29)  OP_CODE = OP_AND;
    else if (instr==8'h25)  OP_CODE = OP_AND;
    else if (instr==8'h35)  OP_CODE = OP_AND;
    else if (instr==8'h2d)  OP_CODE = OP_AND;
    else if (instr==8'h3d)  OP_CODE = OP_AND;
    else if (instr==8'h39)  OP_CODE = OP_AND;
    else if (instr==8'h21)  OP_CODE = OP_AND;
    else if (instr==8'h31)  OP_CODE = OP_AND;
// ASL Instruction
    else if (instr==8'h0a)  OP_CODE = OP_ASL;
    else if (instr==8'h06)  OP_CODE = OP_ASL;
    else if (instr==8'h16)  OP_CODE = OP_ASL;
    else if (instr==8'h0e)  OP_CODE = OP_ASL;
    else if (instr==8'h1e)  OP_CODE = OP_ASL;
// BCC Instruction
    else if (instr==8'h90)  OP_CODE = OP_BCC;
// BCS Instruction
    else if (instr==8'hb0)  OP_CODE = OP_BCS;
// BEQ Instruction
    else if (instr==8'hf0)  OP_CODE = OP_BEQ;
// BIT Instruction
    else if (instr==8'h24)  OP_CODE = OP_BIT;
    else if (instr==8'h2c)  OP_CODE = OP_BIT;
// BMI Instruction
    else if (instr==8'h30)  OP_CODE = OP_BMI;
// BNE Instruction
    else if (instr==8'hd0)  OP_CODE = OP_BNE;
// BPL Instruction
    else if (instr==8'h10)  OP_CODE = OP_BPL;
// BRK Instruction
    else if (instr==8'h00)  OP_CODE = OP_BRK;
// BVC Instruction
    else if (instr==8'h50)  OP_CODE = OP_BVC;
// BVS Instruction
    else if (instr==8'h70)  OP_CODE = OP_BVS;
// CLC Instruction
    else if (instr==8'h18)  OP_CODE = OP_CLC;
// CLD Instruction
    else if (instr==8'hd8)  OP_CODE = OP_CLD;
// CLI Instruction
    else if (instr==8'h58)  OP_CODE = OP_CLI;
// CLV Instruction
    else if (instr==8'hB8)  OP_CODE = OP_CLV;
// CMP Instruction
    else if (instr==8'hc9)  OP_CODE = OP_CMP;
    else if (instr==8'hc5)  OP_CODE = OP_CMP;
```

```
    else if (instr==8'hd5)  OP_CODE = OP_CMP;
    else if (instr==8'hcd)  OP_CODE = OP_CMP;
    else if (instr==8'hdd)  OP_CODE = OP_CMP;
    else if (instr==8'hd9)  OP_CODE = OP_CMP;
    else if (instr==8'hc1)  OP_CODE = OP_CMP;
    else if (instr==8'hd1)  OP_CODE = OP_CMP;
// CPX Instruction
    else if (instr==8'he0)  OP_CODE = OP_CPX;
    else if (instr==8'he4)  OP_CODE = OP_CPX;
    else if (instr==8'hec)  OP_CODE = OP_CPX;
// CPY Instruction
    else if (instr==8'hc0)  OP_CODE = OP_CPY;
    else if (instr==8'hc4)  OP_CODE = OP_CPY;
    else if (instr==8'hcc)  OP_CODE = OP_CPY;
// DEC Instruction
    else if (instr==8'hc6)  OP_CODE = OP_DEC;
    else if (instr==8'hd6)  OP_CODE = OP_DEC;
    else if (instr==8'hce)  OP_CODE = OP_DEC;
    else if (instr==8'hde)  OP_CODE = OP_DEC;
// DEX Instruction
    else if (instr==8'hca)  OP_CODE = OP_DEX;
// DEY Instruction
    else if (instr==8'h88)  OP_CODE = OP_DEY;
// EOR Instruction
    else if (instr==8'h49)  OP_CODE = OP_EOR;
    else if (instr==8'h45)  OP_CODE = OP_EOR;
    else if (instr==8'h55)  OP_CODE = OP_EOR;
    else if (instr==8'h4d)  OP_CODE = OP_EOR;
    else if (instr==8'h5d)  OP_CODE = OP_EOR;
    else if (instr==8'h59)  OP_CODE = OP_EOR;
    else if (instr==8'h41)  OP_CODE = OP_EOR;
    else if (instr==8'h51)  OP_CODE = OP_EOR;
// INC Instruction
    else if (instr==8'he6)  OP_CODE = OP_INC;
    else if (instr==8'hf6)  OP_CODE = OP_INC;
    else if (instr==8'hee)  OP_CODE = OP_INC;
    else if (instr==8'hfe)  OP_CODE = OP_INC;
// INX Instruction
    else if (instr==8'he8)  OP_CODE = OP_INX;
// INY Instruction
    else if (instr==8'hc8)  OP_CODE = OP_INY;
// JMP Instruction
    else if (instr==8'h4c)  OP_CODE = OP_JMP;
    else if (instr==8'h6c)  OP_CODE = OP_JMP;
// JSR Instruction
    else if (instr==8'h20)  OP_CODE = OP_JSR;
// LDA Instruction
    else if (instr==8'ha9)  OP_CODE = OP_LDA;
    else if (instr==8'ha5)  OP_CODE = OP_LDA;
    else if (instr==8'hb5)  OP_CODE = OP_LDA;
    else if (instr==8'had)  OP_CODE = OP_LDA;
    else if (instr==8'hbd)  OP_CODE = OP_LDA;
    else if (instr==8'hb9)  OP_CODE = OP_LDA;
    else if (instr==8'ha1)  OP_CODE = OP_LDA;
    else if (instr==8'hb1)  OP_CODE = OP_LDA;
// LDX Instruction
    else if (instr==8'ha2)  OP_CODE = OP_LDX;
    else if (instr==8'ha6)  OP_CODE = OP_LDX;
    else if (instr==8'hb6)  OP_CODE = OP_LDX;
    else if (instr==8'hae)  OP_CODE = OP_LDX;
    else if (instr==8'hbe)  OP_CODE = OP_LDX;
// LDY Instruction
    else if (instr==8'ha0)  OP_CODE = OP_LDY;
    else if (instr==8'ha4)  OP_CODE = OP_LDY;
    else if (instr==8'hb4)  OP_CODE = OP_LDY;
    else if (instr==8'hac)  OP_CODE = OP_LDY;
    else if (instr==8'hbc)  OP_CODE = OP_LDY;
// LSR Instruction
   else if (instr==8'h4a)  OP_CODE = OP_LSR;
    else if (instr==8'h46)  OP_CODE = OP_LSR;
    else if (instr==8'h56)  OP_CODE = OP_LSR;
```

```
    else if (instr==8'h4e)  OP_CODE = OP_LSR;
    else if (instr==8'h5e)  OP_CODE = OP_LSR;
// ORA Instruction
    else if (instr==8'h09)  OP_CODE = OP_ORA;
    else if (instr==8'h05)  OP_CODE = OP_ORA;
    else if (instr==8'h15)  OP_CODE = OP_ORA;
    else if (instr==8'h0d)  OP_CODE = OP_ORA;
    else if (instr==8'h1d)  OP_CODE = OP_ORA;
    else if (instr==8'h19)  OP_CODE = OP_ORA;
    else if (instr==8'h01)  OP_CODE = OP_ORA;
    else if (instr==8'h11)  OP_CODE = OP_ORA;
// PHA Instruction
    else if (instr==8'h48)  OP_CODE = OP_PHA;
// PHP Instruction
    else if (instr==8'h08)  OP_CODE = OP_PHP;
// PLA Instruction
    else if (instr==8'h68)  OP_CODE = OP_PLA;
// PLP Instruction
    else if (instr==8'h28)  OP_CODE = OP_PLP;
// ROL Instruction
    else if (instr==8'h2a)  OP_CODE = OP_ROL;
    else if (instr==8'h26)  OP_CODE = OP_ROL;
    else if (instr==8'h36)  OP_CODE = OP_ROL;
    else if (instr==8'h2e)  OP_CODE = OP_ROL;
    else if (instr==8'h3e)  OP_CODE = OP_ROL;
// ROR Instruction
    else if (instr==8'h6a)  OP_CODE = OP_ROR;
    else if (instr==8'h66)  OP_CODE = OP_ROR;
    else if (instr==8'h76)  OP_CODE = OP_ROR;
    else if (instr==8'h6e)  OP_CODE = OP_ROR;
    else if (instr==8'h7e)  OP_CODE = OP_ROR;
// RTI Instruction
    else if (instr==8'h40)  OP_CODE = OP_RTI;
// RTS Instruction
    else if (instr==8'h60)  OP_CODE = OP_RTS;
// SBC Instruction
    else if (instr==8'he9)  OP_CODE = OP_SBC;
    else if (instr==8'he5)  OP_CODE = OP_SBC;
    else if (instr==8'hf5)  OP_CODE = OP_SBC;
    else if (instr==8'hed)  OP_CODE = OP_SBC;
    else if (instr==8'hfd)  OP_CODE = OP_SBC;
    else if (instr==8'hf9)  OP_CODE = OP_SBC;
    else if (instr==8'he1)  OP_CODE = OP_SBC;
    else if (instr==8'hf1)  OP_CODE = OP_SBC;
// SEC Instruction
    else if (instr==8'h38)  OP_CODE = OP_SEC;
// SED Instruction
    else if (instr==8'hf8)  OP_CODE = OP_SED;
// SEI Instruction
    else if (instr==8'h78)  OP_CODE = OP_SEI;
// STA Instruction
    else if (instr==8'h85)  OP_CODE = OP_STA;
    else if (instr==8'h95)  OP_CODE = OP_STA;
    else if (instr==8'h8d)  OP_CODE = OP_STA;
    else if (instr==8'h9d)  OP_CODE = OP_STA;
    else if (instr==8'h99)  OP_CODE = OP_STA;
    else if (instr==8'h81)  OP_CODE = OP_STA;
    else if (instr==8'h91)  OP_CODE = OP_STA;
// STX Instruction
    else if (instr==8'h86)  OP_CODE = OP_STX;
    else if (instr==8'h96)  OP_CODE = OP_STX;
    else if (instr==8'h8e)  OP_CODE = OP_STX;
// STY Instruction
    else if (instr==8'h84)  OP_CODE = OP_STY;
    else if (instr==8'h94)  OP_CODE = OP_STY;
    else if (instr==8'h8c)  OP_CODE = OP_STY;
// TAX Instruction
    else if (instr==8'haa)  OP_CODE = OP_TAX;
// TAY Instruction
    else if (instr==8'ha8)  OP_CODE = OP_TAY;
// TYA Instruction
```

```
        else if (instr==8'h98)  OP_CODE = OP_TYA;
    // TSX Instruction
        else if (instr==8'hba)  OP_CODE = OP_TSX;
    // TXA Instruction
        else if (instr==8'h8a)  OP_CODE = OP_TXA;
    // TXS Instruction
        else if (instr==8'h9a)  OP_CODE = OP_TXS;
    // EBT Custom Instruction
        else if (instr==8'hff)  OP_CODE = OP_EBT;
    // NOP Instruction for ALL other Operands
        else                    OP_CODE = OP_NOP;
    end

endmodule
```

Interrupt_Control.v

```
/* ======================================================================
|   NES - 6502 - Interrupt Control
|----------------------------------------------------------------------
|   Description:
|       - Generates the two interrupt signals (RESET done by timer)
|       - The Reset Signal is Clearable, the others clear automatically
|       - NMI and IRQ are generated on the phase 2 clock
|       - RESET is generated based on the main clock but is held
|
|----------------------------------------------------------------------
|   Created by Team Nintendo:  team-nintendo@mit.edu
\======================================================================*/

module Interrupt_Control(MCLR, clk1,
                         IRQ_INHIBIT, IRQ_LINE, IRQ_CLEAR,
                         NMI_LINE, NMI_CLEAR,
                         IRQ, NMI,
                         RESET_INT, RESET_CLEAR);
    input MCLR, clk1;
    input IRQ_INHIBIT, IRQ_LINE, IRQ_CLEAR;
    input NMI_LINE, NMI_CLEAR, RESET_CLEAR;
    output IRQ, NMI, RESET_INT;

    reg IRQ;
    reg NMI;
    reg RESET_INT;

    reg NMI_A;   //NMI_LINE ---> A ---> B
    reg NMI_B;

    always@(posedge clk1) begin
        if (MCLR) begin
            NMI_A <= 1;
            NMI_B <= 1;
            NMI <= 0;
        end
        else begin
            NMI_B <= NMI_A;
            NMI_A <= NMI_LINE;
            if (~NMI_LINE && ~NMI_A && NMI_B)
                NMI <= 1;
            else if (NMI_CLEAR)
                NMI <= 0;
            else
                NMI <= NMI;
        end
    end

    always@(posedge clk1) begin
        if (MCLR)
            RESET_INT <= 1;
```

```
        else if (RESET_CLEAR)
            RESET_INT <= 0;
        else
            RESET_INT <= RESET_INT;
    end

    always@(posedge clk1) begin
        if (MCLR)
            IRQ <= 0;
        else if (~IRQ_LINE && ~IRQ_INHIBIT)
            IRQ <= 1;
        else if (IRQ_CLEAR)
            IRQ <= 0;
        else
            IRQ <= IRQ;
    end

endmodule
```

NES_CPU.v

```
module  nes_cpu(clock, reset_line, nmi_line, irq_line,
                  data_bus, address_bus, RW, clk1, clk2,
                  PRG_CS, PPU_CS, RAM_CS, BOOT_CS,
                  PRG_CE, PPU_DBE,
                  OE1, OE2, STROBE, Out);
    input clock;
    input reset_line, nmi_line;
    inout irq_line;

    inout  [7:0] data_bus;                  // Busses
    output [15:0] address_bus;
    output RW;                              // Bus Direction
    output clk1;
    output clk2;

    output PRG_CS, PPU_CS, RAM_CS, BOOT_CS;
    output PRG_CE, PPU_DBE;

    output OE1, OE2, STROBE;

    output [10:0] Out;

wire [7:0] data_from_ram;
wire [7:0] data_from_rom;
wire [7:0] data_bus_to_cpu;
wire [7:0] data_bus_from_cpu;

wire BT;
wire irq;

//=================================================================================
//   Chip Select Signals
//---------------------------------------------------------------------------------
assign PRG_CS = (BT) ? 1'b0 : (clk2 && address_bus[15]);
assign PPU_CS = ((address_bus[15:13]==3'b001) && clk2);
assign RAM_CS = clk2 && (address_bus[15:8] < 8'h08);
assign BOOT_CS = (BT) ? (clk2 && address_bus[15]) : 1'b0;

assign PPU_DBE = ~PPU_CS;
assign PRG_CE = ~PRG_CS;
//---------------------------------------------------------------------------------

//=================================================================================
//      CPU
//---------------------------------------------------------------------------------
cpu     SIX_FIVE_O_TWO(
        .clock (clock),
```

```
        .reset_line (reset_line),
        .nmi_line (nmi_line),
        .irq_line (irq_line),
        .data_bus_out (data_bus_from_cpu),
        .data_bus_in (data_bus_to_cpu),
        .address_bus (address_bus),
        .RW (RW),
        .clk1 (clk1),
        .clk2 (clk2),
        .BT (BT)
);
//=======================================================================

//=======================================================================
//      SRAM
//-----------------------------------------------------------------------
/*reg SRAM_WE;
reg [3:0] counter;

always@(posedge clock) begin
    if (clk1) begin
        counter <= 0;
        SRAM_WE <= RAM_CS && ~RW;
    end
    else begin
        if (counter < 4'd4) begin
            SRAM_WE <= RAM_CS && ~RW;
            counter <= counter + 1;
        end
        else begin
            SRAM_WE <= 0;
            counter <= counter;
        end
    end
end


sram sram_inst (
    .address ( address_bus[9:0] ),                  // Ten Address Lines = 1K
    .inclock ( clock ),
    .we ( SRAM_WE ),
    .data ( data_bus_from_cpu ),
    .q ( data_from_ram )
    );*/

//=======================================================================


//=======================================================================
//      PROGRAM ROM
//-----------------------------------------------------------------------
prg_rpm prg_rpm_inst (
    .address ( address_bus[10:0] ),                 // Eleven Address Lines = 2048 bytes
    .q ( data_from_rom )
    );
//=======================================================================


//=======================================================================
//      JOYSTICK INPUT  $4016 and $4017
//-----------------------------------------------------------------------
    reg OE1, OE2, STROBE;
    wire JOYSTICK;

    always@(posedge clk1) begin
        if (~reset_line) begin
            STROBE <= 0;
        end
        else if (~RW) begin
            if ((address_bus[15:0]==16'h4016) || (address_bus[15:0]==16'h4017))
                STROBE <= data_bus[0];
```

```
            else STROBE <= STROBE;
        end
        else
            STROBE <= STROBE;
    end

    always@(RW or clk2 or address_bus[15:0]) begin
        OE1 = ~((address_bus[15:0]==16'h4016) && RW && clk2);
        OE2 = ~((address_bus[15:0]==16'h4017) && RW && clk2);
    end

    assign JOYSTICK = (((address_bus[15:0]==16'h4016) || (address_bus[15:0]==16'h4017)) && RW &&
clk2);
//=================================================================================


//=================================================================================
//      Custom Output Module  $4018 and $4019 (Write Only)
//      $4018 = | Out7 | Out6 | Out5 | Out4 | Out3 | Out2 | Out1  | Out0 |
//      $4019 = |      |      |      |      |      | OutA | Out9  | Out8 |
//---------------------------------------------------------------------------------
    reg [10:0] Out;

    always@(posedge clk1) begin
        if (~reset_line)
            Out <= 11'd0;
        else if ((~RW) && (address_bus[15:0]==16'h4018))
            Out <= {Out[10:8], data_bus_from_cpu[7:0]};
        else if ((~RW) && (address_bus[15:0]==16'h4019))
            Out <= {data_bus_from_cpu[2:0], Out[7:0]};
        else
            Out <= Out;
    end
//=================================================================================



//=================================================================================
//      Custom Timer Module  $4020  (Write Only)
//  | Period[6:0] | Enabled |
//---------------------------------------------------------------------------------
    reg [6:0] Period;
    reg Enabled;
    reg [6:0] count;
    reg [7:0] prescale;
    wire Timer_Force_IRQ;

    always@(posedge clk1) begin
        if (~reset_line) begin
            Enabled <= 0;
            Period <= 7'd0;
            count <= 0;
            prescale <= 0;
        end
        else if ((~RW) && (address_bus[15:0]==16'h4020)) begin
            Period <= data_bus_from_cpu[7:1];
            Enabled <= data_bus_from_cpu[0];
            count <= 0;
            prescale <= 0;
        end
        else begin
            Period <= Period;
            Enabled <= Enabled;
            if (prescale < 8'd233) begin
                prescale <= prescale + 1;
                count <= count;
            end
            else begin
                prescale <= 0;
                if (count<Period)
                    count <= count + 1;
```

```
                else
                    count <= Period;
            end
        end
    end

    assign Timer_Force_IRQ = ((count == Period) && (Enabled));
//=============================================================================


//=============================================================================
//     Output
//-----------------------------------------------------------------------------
assign irq_line = (Timer_Force_IRQ) ? 1'b0 : 1'bz;

assign data_bus_to_cpu =  (BOOT_CS) ?    data_from_rom :
                          /*(RAM_CS) ?       data_from_ram :*/
                          (JOYSTICK) ?   {7'b0100000,data_bus[0]} :
                                         data_bus;

wire FLOAT_BUS;
assign FLOAT_BUS = (~clk2 || (RW && ~(BOOT_CS))) ? 1'b1 : 1'b0 ; // RAM_CS

wire [7:0] output_value;
assign output_value =  (BOOT_CS)   ?      data_from_rom :
                       /*(RAM_CS) ?   data_from_ram :*/
                                      data_bus_from_cpu;


assign data_bus =  (FLOAT_BUS) ? 8'hzz : output_value;

endmodule
```

Reset.v

```
/* =============================================================================
|   NES - 6502 - Reset
|-----------------------------------------------------------------------------
|   Description:
|       - Generates the MCLR signal from the reset button
|       - Also implements the six-cycle wait upon restart
|
|   reset_line ----> |A| ----> |B|
|         |            |         |      ______
|         |            |         \------|     )
|         |            \----------------| NAND )-----MCLR
|         \-----------------------------|_____)
|
|-----------------------------------------------------------------------------
|   Created by Team Nintendo:  team-nintendo@mit.edu
\=============================================================================*/

module Reset(clock, reset_line, MCLR);
    input clock, reset_line;
    output MCLR;

    reg A;
    reg B;

    always@(posedge clock)
    begin
        B <= A;
        A <= reset_line;
    end

    assign MCLR = ~(reset_line && A && B);
endmodule
```

Timing_Control.v

```
/* ======================================================================
|   NES - 6502 - Timing Control
|       Keeps track of the instruction cycle and timing.
|       Cycle 0 through 7 are normal instructions where cycle 0
|            is the Instruction Fetch....
|       Cycle 8 through 15 are the cycles for startup after reset beginning
|           with cycle 8.
|       Instruction Length does NOT include the fetch...
|
|   Created by Team Nintendo:  team-nintendo@mit.edu
|
\======================================================================*/

module timing_control(MCLR, clk1, instr_complete, cycle);
    input   MCLR, clk1;
    input    instr_complete;
    output  [2:0] cycle;

    reg [2:0] cycle;

    always @(posedge clk1)
    begin
        if (MCLR)
        begin
            cycle <= 3'b000;
        end
        else begin
            if (instr_complete || (cycle==3'b111))
                cycle <= 0;
            else
                cycle <= cycle +1;
        end

    end

endmodule
```

Appendix E: PPU Verilog

ppu.v

```
module PPU(reset, clock, priority_mux_output); // simple inputs
                                               //scanline_index, scanline_x_coordinate, // 9-bit
inputs
                                               //background_RD, background_ALE, // simple outputs
                                               //background_out, // 5-bit output: two pattern bits,
priority, and two color select bits
                                               //background_PPU_address, // 14-bit output
                                               //background_PPU_data, // 8-bit input
                                               //r2001_out, // left side background clipping
                                               //r2006_0_out, // x-scroll
                                               //r2006_8_out, // y-scroll
                                               //r2000_0_out, // x_scroll_nametable_select
                                               //r2000_1_out, // y_scroll_nametable_select
                                               //r2000_4_out);

    // +++++++++++++++++++++++++++++++ //
    // ------  INPUTS / OUTPUTS  ------ //
    // ------------------------------- //

    input reset, clock;
    // input R_W, CE;   !!!!!!!!!!!!!!!!!!!!!!!!
    // input [2:0] CPU_address; !!!!!!!!!!!!!!!!!!!
    //input [1:0] select;

    //inout [7:0] CPU_data;  !!!!!!!!!!!!!!!
    //  output CPU_data_out;
    //inout [7:0] AD;   !!!!!!!!!!!!!!!!!!!!!!!!!!!!
    //  output [7:0] AD_out, AD;

    //output RD, WE, ALE, VBL; // NOTE!!! VBL goes to processor, vblank goes to monitor
    //output hblank, vblank, hsync_VGA, vsync_VGA;
    //output [13:8] PA;
    output [5:0] priority_mux_output;

//  output PPU_5370_clock, hblank; // simple inputs (to background renderer)
//  output [8:0] scanline_index, scanline_x_coordinate; // 9-bit inputs (to background renderer)
//
//  output background_RD, background_ALE; // simple outputs
//  output [4:0] background_out; // 5-bit output: two pattern bits, priority, and two color
select bits
//  output [13:0] background_PPU_address; // 14-bit output
//  output [7:0] background_PPU_data; // 8-bit input
//  output r2001_out; // left side background clipping
//  output [7:0] r2006_0_out; // x-scroll
//  output [7:0] r2006_8_out; // y-scroll
//  output r2000_0_out; // x_scroll_nametable_select
//  output r2000_1_out; // y_scroll_nametable_select
//  output r2000_4_out;

    // +++++++++++++++++++++ //
    // ------  WIRES  ------ //
    // --------------------- //

    wire [13:0] PPU_address, sprite_PPU_address, background_PPU_address;
    wire [7:0] PPU_data, sprite_sprite_RAM_address, sprite_RAM_data_out, sprite_PPU_data,
background_PPU_data, sprite_RAM_address;
    wire sprite_RAM_RD;
    wire PPU_5370_clock; // 5.370 MHz clock
    wire [4:0] pixel, sprite_to_mux;
    wire [3:0] background_to_mux;
    wire sprite_RD, sprite_ALE, background_RD, background_ALE;
    wire CPU_is_reading, CPU_is_writing;
```

```
wire [8:0] scanline_index, scanline_x_coordinate;
wire more_than_8_sprites;
wire [4:0] sprite_out;
wire [3:0] background_out;
wire hsync, vsync;
wire [7:0] palette_RAM_data_out;
wire priority_mux_palette_reading;
wire [4:0] priority_mux_palette_address;
 wire [5:0] priority_mux_output;
wire VBL, hblank, vblank, RD;
wire primary_pixel_being_rendered;
wire [7:0] AD; //  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
wire CE, R_W; // !!!!!!!!!!!!!!!!!!!!!!!!!!!!
wire [2:0] CPU_address;  // !!!!!!!!!!!!!!!!!!!!!!!!!!!
wire [7:0] CPU_data;   // !!!!!!!!!!!!!!!!!!!!!!!!
wire [13:8] sprite_PA;
wire CPU_data_out;
wire [7:0] AD_out;
wire ALE, sprite_pattern_table_selection, attribute_WE, nametable_WE, WE;


assign sprite_PA = sprite_PPU_address[13:8];
// ++++++++++++++++++++++++++ //
// ------  REGISTERS  ------ //
// ------------------------ //

reg [7:0] R2000, R2001, R2003, R2004, R2007;
reg [7:5] R2002;
reg [15:0] R2006, R2005;
reg R2005_flag1, R2005_flag2, R2006_flag1, R2006_flag2, R2007_flag, R2007_flag2;
reg R2004_flag; // this helps us determine whether R2003 should be incremented
reg CPU_WE, CPU_ALE, CPU_RD, CPU_sprite_RAM_RD, sprite_RAM_WE;
reg [7:0] sprite_RAM_data_in, CPU_sprite_RAM_address, CPU_PPU_data;
reg [13:0] CPU_PPU_address;
reg [7:0] palette_RAM_data_in;
reg palette_RAM_WE, palette_RAM_active;
reg vblank_previous;
reg [4:0] palette_RAM_address;
reg primary_object_collision;
reg delay_state, ALE_flag;
reg [13:8] PA;
reg [1:0] R2007_delay_state, palette_state;
reg RD_flag, nametable_WE_flag, attribute_WE_flag;



// ++++++++++++++++++++++ //
// ------  MODULES  ------ //
// ---------------------- //


// +----------------------------+ //    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// |   test controller module   | //
// +----------------------------+ //

PPU_test_controller test_controller(clock, reset, vblank, CPU_address, CPU_data, CE, R_W);

// internal VRAM modules... nametable and attribute table
reg [6:0] nametable_address;
wire [7:0] latch_data_in, latch_data_out;
reg [2:0] attribute_address;
reg [5:0] CHR_ROM_address;
reg [7:0] nametable_data_in, attribute_data_in;
wire [7:0] AD_CHR_ROM, AD_nametable, AD_attribute;

PPU_nametable_RAM internal_nametable( nametable_address,
                                      nametable_WE,
                                      nametable_data_in,
                                      AD_nametable);

PPU_attribute_RAM internal_attribute_table(  attribute_address,
```

```
                                        attribute_WE,
                                        attribute_data_in,
                                        AD_attribute);

    always @(posedge clock)
        begin
            if (ALE) // writing an address
                begin
                    if (PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & PA[9] & PA[8] & AD_out[7] &
AD_out[6] & ~AD_out[5] & ~AD_out[4] & ~AD_out[3])
                        attribute_address <= latch_data_out[2:0];
                    else if (PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & ~PA[9])
                        nametable_address <= latch_data_out[6:0];
                    else if (~PA[13])
                        CHR_ROM_address <= latch_data_out[5:0];

                end
            else if (~RD | ~WE) // reading or writing
                begin
                    nametable_data_in <= AD_out;
                    attribute_data_in <= AD_out;
                end
        end


    // internal CHR ROM module
    PPU_CHR_ROM internal_CHR_ROM( CHR_ROM_address,
                                  AD_CHR_ROM);

    assign AD = (~RD) ?
                ((PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & PA[9] & PA[8]) ? AD_attribute :
                    (PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & ~PA[9]) ? AD_nametable : AD_CHR_ROM) :
8'b00000000;

    // internal latch chip
        PPU_latch internal_latch(clock, ALE, AD_out, latch_data_out);

    // !!!!!!!!!!!!!!!!!!!! END OF EXTRA TESTING MODULES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    // +-----------------+ //
    // |   palette RAM   | //
    // +-----------------+ //
    PPU_palette_RAM palette_RAM(palette_RAM_address,
                                palette_RAM_WE,
                                palette_RAM_data_in,
                                palette_RAM_data_out);

    assign priority_mux_palette_reading = (scanline_index >= 21 && scanline_index <= 261 &&
hblank); // will read only when the pixels are being displayed

    // +----------------+ //
    // |   sprite RAM   | //
    // +----------------+ //
    PPU_sprite_RAM sprite_RAM(sprite_RAM_address,
                              sprite_RAM_WE,
                              sprite_RAM_data_in,
                              sprite_RAM_data_out);

    assign sprite_RAM_address = (((CPU_address == 3'b100) && (CPU_is_writing || CPU_is_reading))
|| sprite_RAM_WE) ?
                                    CPU_sprite_RAM_address : sprite_sprite_RAM_address;

    // +-------------------+ //
    // |   clock divider   | //   Outputs 5.370 MHz clock
    // +-------------------+ //
    PPU_clock_divider our_PPU_clock_divider(clock, PPU_5370_clock);

    // +-----------------------+ //
    // |   VGA output module   | //   Handles the output to the screen
    // +-----------------------+ //
```

```
    table2vga VGA_output(reset,clock,vblank,hblank,scanline_x_coordinate,scanline_index); //these
are for VGA display


    // +-----------------------------+ //
    // |   sprite rendering engine   | //
    // +-----------------------------+ //
    PPU_sprite_renderer sprite_renderer(reset, PPU_5370_clock, hblank, scanline_index,
scanline_x_coordinate,
                                        sprite_sprite_RAM_address, sprite_RAM_data_out,
sprite_PPU_address, AD,
                                        more_than_8_sprites, R2001[2], // left side object clipping
                                        sprite_out, R2000[5], sprite_ALE, sprite_RD,
primary_pixel_being_rendered, sprite_pattern_table_selection);

//assign sprite_out = 0;
//assign sprite_ALE = 0;
//assign sprite_RD = 0;
//assign sprite_PPU_address = 0;

    assign sprite_pattern_table_selection = R2000[3];
    assign sprite_PPU_data = (!sprite_RD) ? AD : 0;

    // +---------------------------------+ //
    // |   background rendering engine   | //
    // +---------------------------------+ //
    PPU_background_renderer background_rendered(reset, PPU_5370_clock, hblank, // simple inputs
                                                scanline_index, scanline_x_coordinate, // 9-bit
inputs
                                                background_RD, background_ALE, // simple outputs
                                                background_out, // 5-bit output: two pattern bits,
priority, and two color select bits
                                                background_PPU_address, // 14-bit output
                                                background_PPU_data, // 8-bit input
                                                R2001[1], // left side background clipping
                                                R2006[7:0], // x-scroll
                                                R2006[15:8], // y-scroll
                                                R2000[0], // x_scroll_nametable_select
                                                R2000[1], // y_scroll_nametable_select
                                                R2000[4]); // playfield_pattern_table_select

    assign background_PPU_data = (!background_RD) ? AD : 0;   /// !!!!!!!!!!!!!!!!!!!!! should
be AD
    //assign background_out = 0; // !!!
    //assign background_RD = 0; // !!!
    //assign background_ALE = 0; // !!!
    //assign background_PPU_address = 0; // !!!


    // ++++++++++++++++++++++++++++++ //
    // ------  I/O  PROTOCOLS  ------ //
    // ------------------------------ //

    always @(posedge clock)
        begin
            if (CPU_ALE)
                PA = CPU_PPU_address[13:8];
            else if (sprite_ALE)
                PA = sprite_PPU_address[13:8];
            else if (background_ALE)
                PA = background_PPU_address[13:8];
            else
                PA = PA;
        end

    assign AD_out = !(!CPU_WE | CPU_ALE | sprite_ALE | background_ALE) ? 8'b00000000 : // 8'hz :
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                     (!CPU_WE) ? CPU_PPU_data :
                    (CPU_ALE) ? CPU_PPU_address[7:0] :
                    (sprite_ALE) ? sprite_PPU_address[7:0] : background_PPU_address[7:0];
```

```
    assign RD = (RD_flag && (!sprite_RD || !background_RD || !CPU_RD)) ?
                  0 : 1; // RD low if anyone is reading, else high
    always @(posedge clock)
        begin
        if (!sprite_RD || !background_RD || !CPU_RD)
            RD_flag <= 1;
        else
            RD_flag <= 0;
        end

    assign WE = CPU_WE;
    assign nametable_WE = (PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & PA[9] & PA[8] & !CPU_WE ) ? 0 :
1;
    assign attribute_WE = (PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & ~PA[9] & !CPU_WE) ? 0 : 1;


    //assign nametable_WE = (PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & PA[9] & PA[8] & !CPU_WE) ? 0
: 1;
    //assign attribute_WE = (PA[13] & ~PA[12] & ~PA[11] & ~PA[10] & ~PA[9] & !CPU_WE) ? 0 : 1;

    assign ALE = (ALE_flag && (sprite_ALE || background_ALE || CPU_ALE)) ?
                  1 : 0; // ALE high if anyone requests it, else low.
    always @(posedge clock)
    begin
        if (sprite_ALE || background_ALE || CPU_ALE)
            ALE_flag <= 1;
        else
            ALE_flag <= 0;
    end

    assign VBL = (R2002[7] && R2000[7]) ? 0 : 1; // logical NAND between the two registers

    assign CPU_is_reading = (!CE && R_W);
    assign CPU_is_writing = (!CE && !R_W);


    // +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ //
    // ------  CPU DATA LINES (including access to registers)  ------ //
    // ------------------------------------------------------------- //

    assign CPU_data_out = (!CPU_is_reading) ? 8'b00000000 : /// 8'hz : // if not reading,
tristate pins   !!!!!!!!!!!!!!!!!!!!!!!!!
                      (CPU_address == 3'b010) ? {R2002, 5'b00000} :
                      (CPU_address == 3'b100) ? R2004 :
                      (CPU_address == 3'b111) ? R2007 :
                      //8'b00000000; // other ports should never be read
                      (CPU_address == 3'b000) ? R2000 :
                      (CPU_address == 3'b001) ? R2001 :
                      (CPU_address == 3'b011) ? R2003 :
                      (CPU_address == 3'b101) ? R2005[10:3] :
                      (CPU_address == 3'b110) ? R2006[10:3] : 8'b00000000;




    // +++++++++++++++++++++++++++++++++++++++++++++++++ //
    // ------  INITIALIZATION & REGISTER CONTROL  ------ //
    // ------------------------------------------------- //



    always @(posedge clock)
    begin

        if (reset)
            begin

                // zero registers
                R2004_flag <= 0;
                R2005_flag1 <= 0;
                R2005_flag2 <= 0;
```

```
                R2006_flag1 <= 0;
                R2006_flag2 <= 0;
                R2007_flag <= 0;
                R2007_flag2 <= 0;
                R2000 <= 0;
                R2001 <= 0;
                R2002 <= 0;
                R2003 <= 0;
                R2004 <= 0;
                R2005 <= 0;
                R2006 <= 0;
                R2007 <= 0;

                // standardize outputs
                CPU_WE <= 1;
                CPU_ALE <= 0;
                CPU_RD <= 1;
                sprite_RAM_WE <= 0;
                sprite_RAM_data_in <= 0;
                CPU_sprite_RAM_address <= 0;
                CPU_PPU_data <= 0;
                CPU_PPU_address <= 0;
                palette_RAM_data_in <= 0;
                palette_RAM_WE <= 0;
                palette_RAM_active <= 0;
                palette_RAM_address <= 0;
                vblank_previous <= 0;
                delay_state <= 0;
                CPU_sprite_RAM_RD <= 0;
                R2007_delay_state <= 0;
                palette_state <= 0;

            end

        else // not reset
            begin

                // ------------------------------------------- //
                // handling the finer details of register reads //  flags need to be set, and some
registers need to fetch data
                // ------------------------------------------- //

                if (CPU_is_reading)
                    case (CPU_address)
                        3'b100 :   // R2004 : sprite RAM data
                            begin
                                R2004_flag <= 1;
                                R2004 <= sprite_RAM_data_out; // NOTE!!! sprite RAM WE signal is
expected to be low
                            end
                        3'b111 :   // R2007 : VRAM data access
                            begin
                                R2007 <= AD;
                                CPU_RD <= 0;
                                R2007_flag <= 1;
                            end
                    endcase


                // PALETTE RAM signals
                if (priority_mux_palette_reading)
                    palette_RAM_address <= priority_mux_palette_address;
                else if (ALE)
                    palette_RAM_address <= AD_out[4:0];   ////////// !!!!!!!!!!!!!!!!!!!!!!!!
this should be AD

                if (ALE)
                    begin
                        if ((PA[13] && PA[12] && PA[11] && PA[10] && PA[9] && PA[8]) && (!AD[7])
&& (!AD[6]) && (!AD[5]))
                            palette_RAM_active <= 1;
```

```
                    else
                        palette_RAM_active <= 0;
                end



            // -------------------------------------- //
            // handling access to individual registers //
            // -------------------------------------- //


            // +----------+ //
            // |  $2000   | //
            // +----------+ //
            if (CPU_is_writing && CPU_address == 3'b000)
                R2000 <= CPU_data;


            // +----------+ //
            // |  $2001   | //
            // +----------+ //
            if (CPU_is_writing && CPU_address == 3'b001)
                R2001 <= CPU_data;


            // +----------+ //
            // |  $2002   | //
            // +----------+ //
            if (!CPU_is_reading)
                begin
                    R2002[5] <= more_than_8_sprites; // signal comes from sprite renderer
                    R2002[6] <= primary_object_collision; // signal comes from priority mux

                    if (vblank_previous && !vblank) // vblank has just gone low
                        begin
                            R2002[7] <= 1;
                            vblank_previous <= vblank;
                        end
                    else
                        vblank_previous <= vblank;

                    // all others bits in R2002 are set to zero by default, and not even
included in our module
                end
            else if (CPU_is_reading && CPU_address == 3'b010)
                begin
                    R2002[7] <= 0; // when reading R2002, the vblank flag is reset
                    R2005_flag1 <= 0;
                    R2006_flag1 <= 0;

                    if (!hblank && !R2000[7])
                        R2000[1:0] <= 2'b00; // reset the nametable select bits
                end


            // +----------+ //
            // |  $2003   | //
            // +----------+ //

            if (CPU_is_writing && CPU_address == 3'b011)
                begin
                    R2003 <= CPU_data;
                    CPU_sprite_RAM_address <= CPU_data;
                end
            else if (R2004_flag && !CPU_is_reading && !CPU_is_writing) // CPU has finished i/o
to R2004
                begin


                    if (delay_state)
                        begin
```

```
                        R2003 <= R2003 + 1;
                        R2004_flag <= 0;
                        delay_state <= 0;
                        CPU_sprite_RAM_address <= CPU_sprite_RAM_address + 1;
                    end
                else
                    begin
                        sprite_RAM_WE <= 0;
                        delay_state <= 1;
                    end

            end


        // +----------+ //
        // |  $2004   | //
        // +----------+ //
        if (CPU_is_writing && CPU_address == 3'b100)
            begin
                R2004 <= CPU_data;

                // put information from data lines to sprite RAM
                sprite_RAM_data_in <= CPU_data;

                case (delay_state)
                    0:
                        delay_state <= 1;
                    1:
                        begin
                            sprite_RAM_WE <= 1;
                            R2004_flag <= 1;
                            delay_state <= 0;
                        end
                endcase
            end


        // +----------+ //
        // |  $2005   | //
        // +----------+ //
        if (CPU_is_writing && CPU_address == 3'b101)
            begin
                R2005_flag2 <= 1; // signify that R2005 has been written to
                if (R2005_flag1 == 0) // first write to R2005 ?
                    R2005[7:0] <= CPU_data; // x-scroll value
                else
                    R2005[15:8] <= CPU_data;  // y-scroll value
            end
        else if (!CPU_is_writing && R2005_flag2) // CPU has finished writing to R2005
            begin
                R2005_flag2 <= 0;
                R2005_flag1 <= ~R2005_flag1; // every other time, the higher byte is
written to
            end


        // +----------+ //
        // |  $2006   | //
        // +----------+ //
        if (CPU_is_writing && CPU_address == 3'b110)
            begin
                R2006_flag2 <= 1; // signify that R2006 has been written to
                if (R2006_flag1 == 0) // first write to R2006 ?
                    R2006[15:8] <= CPU_data;
                else
                    begin
                        R2006[7:0] <= CPU_data;
                        CPU_ALE <= 1;
                        CPU_PPU_address <= R2006[13:0];
                    end
            end
```

```
            else if (!CPU_is_writing && R2006_flag2) // CPU has finished writing to R2006, or
R2007
                begin
                    case(delay_state)
                        0:
                            begin
                                delay_state <= 1;
                                if (!R2007_flag2)
                                    begin
                                        R2006_flag1 <= ~R2006_flag1; // every other time, the
higher byte is written to
                                        CPU_ALE <= 0;
                                    end


                            end
                        1:
                            begin
                                if (R2007_flag2)
                                    begin
                                        CPU_ALE <= 0;
                                        R2007_flag2 <= 0;
                                    end
                                delay_state <= 0;
                                R2006_flag2 <= 0;
                            end
                    endcase
                end


            // +----------+ //
            // |  $2007   | //
            // +----------+ //
            if (CPU_is_writing && CPU_address == 3'b111)
                begin
                    R2007 <= CPU_data;
                    R2007_flag <= 1;
                    if (palette_RAM_active) // valid palette RAM address was written.
                        begin
                            case (palette_state)
                                0 : begin
                                        palette_RAM_data_in <= CPU_data;
                                        palette_state <= 1;
                                    end
                                1 : begin
                                        palette_RAM_WE <= 1;
                                        palette_state <= 2;
                                    end
                                2 : palette_state <= 3;
                                3 : begin
                                        palette_state <= 0;
                                        palette_RAM_WE <= 1;
                                    end
                            endcase
                        end
                    else
                        begin
                            CPU_PPU_data <= R2007;
                            CPU_WE <= 0;
                        end
                end
            else if (!CPU_is_writing && !CPU_is_reading && R2007_flag) // CPU has finished I/O
with R2007
                begin

                    case(R2007_delay_state)
                        0:
                            begin
                                R2007_delay_state <= 1;
                                CPU_WE <= 1;
                                CPU_RD <= 1;
```

```
                                palette_RAM_WE <= 0;

                                // increment PPU address port, R2006
                                if (R2000[2])
                                    begin
                                        CPU_PPU_address <= R2006 + 32;
                                        R2006 <= R2006 + 32;
                                    end
                                else
                                    begin
                                        CPU_PPU_address <= R2006 + 1;
                                        R2006 <= R2006 + 1;
                                    end

                                CPU_ALE <= 1;
                            end
                        1:
                            R2007_delay_state <= 2;
                        2:
                            begin
                                R2007_delay_state <= 0;
                                R2007_flag <= 0;
                                R2006_flag2 <= 1;
                                R2007_flag2 <= 1;
                            end
                        default: R2007_delay_state <= 0;
                    endcase

                end



            end // not reset

    end // always block



    // +++++++++++++++++++++++++++ //
    // ------  PRIORITY MUX  ------ //
    // --------------------------- //

            // ***  STEP 1: Determine what should be displayed
            // R2001[3] = enable playfield display
            // R2001[4] = enable object (sprite) display ... see priority_mux_output for handling

        assign sprite_to_mux = (R2001[4]) ? sprite_out : 0;
        assign background_to_mux = background_out;  // this ensures that all background colors
are the same in the palette: the 0th element
            // *** STEP 2: Determining pixel output. Note these rules:
            //  1. If sprite is invisible, display background pixel
            //  2. Else if sprite has high priority, display sprite.
            //  3. Else display background, except if background is zero.
        assign priority_mux_palette_address = (sprite_to_mux[4:3] == 0) ? {1'b0,
background_to_mux[1:0], background_to_mux[3:2]} : // RULE 1
                                              (sprite_to_mux[2]) ? {1'b1, sprite_to_mux[1:0],
sprite_to_mux[4:3]} : // RULE 2
                                                  (background_to_mux[3:2] == 0) ? {1'b1,
sprite_to_mux[1:0], sprite_to_mux[4:3]} : // RULE 3
                                                      {1'b0, background_to_mux[1:0],
background_to_mux[3:2]};

        always @(posedge clock)
            begin
                if (reset || scanline_index == 20)
                    primary_object_collision <= 0;
                else if ((sprite_to_mux[4:3] != 0) && (background_to_mux[3:2] != 0) &&
(primary_pixel_being_rendered))
                    primary_object_collision <= 1;
            end
```

```
       assign priority_mux_output = (!vblank || !hblank) ? 6'b001101 : // output black during
hblank, vblank
                                    ((sprite_to_mux[4:3] == 0) && !R2001[3]) ?  // full
background color should be shown
                                        (
                                            (R2001[7:5] == 3'b000) ? 6'b001101 : // full black
                                            (R2001[7:5] == 3'b001) ? 6'b010110 : // full red
                                            (R2001[7:5] == 3'b010) ? 6'b010010 : // full green
                                            (R2001[7:5] == 3'b100) ? 6'b011010 : 6'b001101 //
full blue, and if the bits are wrong output full black
                                        ) : palette_RAM_data_out[5:0]; // else just show the
palette data

    // --- END OF PRIORITY MUX --- //


endmodule
```

table2vga.v

```
module table2vga(reset,clk,vblank,hblank,hcount_global,vcount_global); //these are for VGA
display

  input reset, clk;              // delivered by PPU
  output vblank,hblank;          // For the PPU's use ONLY -- NOT REAL
  output [8:0] vcount_global, hcount_global;


  parameter color_min = 6'b000000;
  parameter color_max = 6'b111111;

  reg pcount;       // used to generate pixel clock
  wire en = (pcount == 0);
  always @ (posedge clk) pcount <= ~pcount;

  //////////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////////
  //***
  //***  Blanking and Count Signals FOR THE PPU
  //***
  //////////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////////

      reg hblank,vblank;
      reg [9:0] hcount;      // pixel number on current line
      reg [9:0] vcount;        // line number

      //////////////////////////////////////////////////////////////////
      //////////////////////////////////////////////////////////////////
      // "Fake" PPU display inputs
      wire hreset,hblankon;
      assign hblankon = en & (hcount == 511);
      assign hreset   = en & (hcount == 681);
      // vertical: 262 lines
      // display 240 lines
      wire vsyncon,vsyncoff,vreset,vblankon;
      assign vblankon = hreset & (vcount == 239); //CRITICAL KEY!!
      assign vreset   = hreset & (vcount == 262);

      wire [8:0] vcount_global, hcount_global;
      assign vcount_global = (vcount >= 242) ? (vcount - 242) : (vcount + 21);
      assign hcount_global = hcount[9:1];

      //////////////////////////////////////////////////////////////////
      //////////////////////////////////////////////////////////////////

      // Blanking and Count
```

```
      always @(posedge clk) begin
        if (reset) begin
            hcount <= 0;
            hblank <= 1;
            vcount <= 0;
            vblank <= 1;
        end // if statement, reset case
        else begin
            hcount <= en ? (hreset ? 10'b0000000000 : hcount + 1) : hcount;
            hblank <= hreset ? 1 : hblankon ? 0 : hblank; // active low

            vcount <= hreset ? (vreset ? 10'b0000000000 : vcount + 1) : vcount;
            vblank <= vreset ? 1 : vblankon ? 0 : vblank; // active low
        end // else
      end

  //////////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////////
  //***
  //***  Sync Signals FOR THE VGA SCREEN OUTPUT
  //***
  //////////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////////

      reg hsync_VGA,vsync_VGA,hblank_VGA,vblank_VGA;
      reg [9:0] hcount_VGA;      // pixel number on current line
      reg [9:0] vcount_VGA;        // line number

      // horizontal: 341 pixels
      // display 270 "pixels" per line
      // 256 output and 14 extra lines on the sides
      wire hsyncon_VGA,hsyncoff_VGA,hreset_VGA,hblankon_VGA;
      assign hblankon_VGA = en & (hcount_VGA == 269);
      assign hsyncon_VGA  = en & (hcount_VGA == 279);
      assign hsyncoff_VGA = en & (hcount_VGA == 320);
      assign hreset_VGA   = en & (hcount_VGA == 340);

      // vertical: 525 lines
      // display 480 lines
      wire vsyncon_VGA,vsyncoff_VGA,vreset_VGA,vblankon_VGA;
      assign vblankon_VGA = hreset_VGA & (vcount_VGA == 479);
      assign vsyncon_VGA  = hreset_VGA & (vcount_VGA == 490);
      assign vsyncoff_VGA = hreset_VGA & (vcount_VGA == 492);
      assign vreset_VGA   = hreset_VGA & (vcount_VGA == 525);

      // Blanking and Count

      always @(posedge clk) begin
        if (reset) begin
            hcount_VGA <= 0;
            hblank_VGA <= 0;
            hsync_VGA  <= 1;
            vcount_VGA <= 0;
            vblank_VGA <= 0;
            vsync_VGA  <= 1;
        end // if statement, reset case

        else begin
            hcount_VGA <= en ? (hreset_VGA ? 10'b0000000000 : hcount_VGA + 1) : hcount_VGA;
            hblank_VGA <= hreset_VGA ? 0 : hblankon_VGA ? 1 : hblank_VGA;  // hblank is avtive
high
            hsync_VGA <= hsyncon_VGA ? 0 : hsyncoff_VGA ? 1 : hsync_VGA;   // hsync is active low

            vcount_VGA <= hreset_VGA ? (vreset_VGA ? 10'b0000000000 : vcount_VGA + 1) :
vcount_VGA;
            vblank_VGA <= vreset_VGA ? 0 : vblankon_VGA ? 1 : vblank_VGA;  // hblank is avtive
high
            vsync_VGA <= vsyncon_VGA ? 0 : vsyncoff_VGA ? 1 : vsync_VGA;   // vsync is active low
        end // else
      end
```

```
endmodule
```

ppu_sprite_renderer.v

```
module PPU_sprite_renderer(reset, PPU_5370_clock, hblank, scanline_counter, scanline_clock,
                           sprite_RAM_address, sprite_RAM_data, PPU_address, PPU_data,
                           more_than_eight_sprites, left_side_object_clipping, sprite_out,
                           sprite_size, ALE, RD, primary_pixel, pattern_table_selection);
    // ------------------------------ //
    // ------ INPUT / OUTPUT  ------ //
    // ------------------------------ //

    input PPU_5370_clock, reset, hblank, left_side_object_clipping,
           sprite_size, pattern_table_selection; // 0 = 8x8 sprite, 1 = 8x16 sprite;
    input [8:0] scanline_counter;
    input [8:0] scanline_clock;
    input [7:0] sprite_RAM_data, PPU_data;

    output more_than_eight_sprites, ALE, RD;
    output [4:0] sprite_out;
    output [13:0] PPU_address;
    output [7:0] sprite_RAM_address;
    output primary_pixel;

    // -------------------------------- //
    // ------ REGISTERS & WIRES  ------ //
    // -------------------------------- //

    reg [4:0] sprite_out;
    reg [7:0] sprite_RAM_address;
    reg [7:0] range_comparator;
    reg [4:0] temporary_sprite_RAM_address;
    reg [7:0] temporary_sprite_RAM_data_in;
    reg more_than_eight_sprites, vertical_flip;
    reg [13:0] PPU_address;
    reg [3:0] range_search_state;
    //reg [7:0] sprite_buffers_data;
    reg [3:0] memory_fetch_state;
    reg [3:0] sprite_fetching_number, range_search_sprite_number;
    reg [7:0] tile_index_number;
    reg temporary_sprite_RAM_WE, ALE, RD;
    reg horizontal_flip_flag;
    reg [2:0]  sprite_0_attributes,
               sprite_1_attributes,
               sprite_2_attributes,
               sprite_3_attributes,
               sprite_4_attributes,
               sprite_5_attributes,
               sprite_6_attributes,
               sprite_7_attributes;
    reg [7:0] low_bit_load_register, high_bit_load_register, x_counter_load;
    reg primary_pixel, primary_object_in_range, primary_object_will_be_rendered;

    wire [7:0] sprite_buffers_data;
    wire [8:0] scanline_clock; // counts upward during each scanline. A pixel gets rendered each
count.
    wire sprite_range_search, sprite_vram_memory_fetch, sprite_rendering,
pattern_table_selection;
    wire [1:0] buffer_0_out,
               buffer_1_out,
               buffer_2_out,
               buffer_3_out,
               buffer_4_out,
               buffer_5_out,
               buffer_6_out,
```

```
                buffer_7_out;

    wire [7:0]  buffer_0_output_high,
                buffer_1_output_high,
                buffer_2_output_high,
                buffer_3_output_high,
                buffer_4_output_high,
                buffer_5_output_high,
                buffer_6_output_high,
                buffer_7_output_high;

    wire [7:0] temporary_sprite_RAM_data_out;

    // ---------------------- //
    // ------ MODULES  ------ //
    // ---------------------- //

        // temporary sprite RAM (holds 4*8 = 32 bytes)
    temporary_sprite_RAM temp_sprite_RAM( temporary_sprite_RAM_address,
                                          temporary_sprite_RAM_WE,
                                          temporary_sprite_RAM_data_in,
                                          temporary_sprite_RAM_data_out);

        // sprite buffer module (need 8 of them)
    PPU_sprite_buffer buffer0(sprite_buffers_data, low_bit_load_register[0],
high_bit_load_register[0], x_counter_load[0],
                              PPU_5370_clock, buffer_0_out, reset, hblank);
    PPU_sprite_buffer buffer1(sprite_buffers_data, low_bit_load_register[1],
high_bit_load_register[1], x_counter_load[1],
                              PPU_5370_clock, buffer_1_out, reset, hblank);
    PPU_sprite_buffer buffer2(sprite_buffers_data, low_bit_load_register[2],
high_bit_load_register[2], x_counter_load[2],
                              PPU_5370_clock, buffer_2_out, reset, hblank);
    PPU_sprite_buffer buffer3(sprite_buffers_data, low_bit_load_register[3],
high_bit_load_register[3], x_counter_load[3],
                              PPU_5370_clock, buffer_3_out, reset, hblank);
    PPU_sprite_buffer buffer4(sprite_buffers_data, low_bit_load_register[4],
high_bit_load_register[4], x_counter_load[4],
                              PPU_5370_clock, buffer_4_out, reset, hblank);
    PPU_sprite_buffer buffer5(sprite_buffers_data, low_bit_load_register[5],
high_bit_load_register[5], x_counter_load[5],
                              PPU_5370_clock, buffer_5_out, reset, hblank);
    PPU_sprite_buffer buffer6(sprite_buffers_data, low_bit_load_register[6],
high_bit_load_register[6], x_counter_load[6],
                              PPU_5370_clock, buffer_6_out, reset, hblank);
    PPU_sprite_buffer buffer7(sprite_buffers_data, low_bit_load_register[7],
high_bit_load_register[7], x_counter_load[7],
                              PPU_5370_clock, buffer_7_out, reset, hblank);




    // ------------------------------ //
    // ------ SPRITE RENDERING  ------ //
    // ------------------------------ //


    assign sprite_range_search = ((scanline_counter >= 20) && (scanline_counter <= 259) &&
                                  (scanline_clock > 0) && (scanline_clock <= 256)) ? 1 : 0;
    assign sprite_vram_memory_fetch = ((scanline_counter >= 20) && (scanline_counter <= 259)
                                    && (scanline_clock > 256) && (scanline_clock <= 320)) ? 1 :
0;
    assign sprite_rendering = ((scanline_counter >= 21) && (scanline_counter <= 260)
                                    && (scanline_clock >=0) && (scanline_clock <= 256)) ? 1 :
0;



    always @(posedge PPU_5370_clock)
    begin
```

```
        if (reset)
            begin

                //blank output
                sprite_out <= 4'b0000;
                //sprite_buffers_data <= 0;
                tile_index_number <= 0;
                horizontal_flip_flag <= 0;
                sprite_0_attributes <= 0;
                sprite_1_attributes <= 0;
                sprite_2_attributes <= 0;
                sprite_3_attributes <= 0;
                sprite_4_attributes <= 0;
                sprite_5_attributes <= 0;
                sprite_6_attributes <= 0;
                sprite_7_attributes <= 0;
                low_bit_load_register <= 0;
                high_bit_load_register <= 0;
                x_counter_load <= 0;

                // reset external memory access
                sprite_RAM_address <= 0;
                PPU_address <= 0;
                ALE <= 0;
                RD <= 1;

                // reset temporary_sprite_RAM access
                temporary_sprite_RAM_address <= 0;
                temporary_sprite_RAM_WE <= 0;
                temporary_sprite_RAM_data_in <= 0;

                // reset range search
                range_search_state <= 0;
                range_search_sprite_number <= 0;
                range_comparator <= 0;
                more_than_eight_sprites <= 0;
                primary_object_in_range <= 0;

                // reset memory fetch
                memory_fetch_state <= 0;
                sprite_fetching_number <= 1;
                vertical_flip <= 0;
                primary_object_will_be_rendered <= 0;

                primary_pixel <= 0;

            end

        else if  (scanline_clock > 320) // hblank is almost over
            begin
                // reset external memory access
                sprite_RAM_address <= 0;
                PPU_address <= 0;
                ALE <= 0;
                RD <= 1;

                // reset temporary_sprite_RAM access
                temporary_sprite_RAM_address <= 0;
                temporary_sprite_RAM_WE <= 0;
                temporary_sprite_RAM_data_in <= 0;

                // reset range search
                range_search_state <= 0;
                range_search_sprite_number <= 0;
                range_comparator <= 0;
                    // NOTE!!! DO NOT reset more_than_eight_sprites, since it should stay active
for the whole frame

                // reset memory fetch
                memory_fetch_state <= 0;
                sprite_fetching_number <= 1;
```

```
                vertical_flip <= 0;

                primary_pixel <= 0;
                horizontal_flip_flag <= 0;
                tile_index_number <= 0;

            end

        else if (scanline_counter == 20 && more_than_eight_sprites == 1)
            begin
                more_than_eight_sprites <= 0;
            end

        else
            begin

                // +-------------------------+ //
                // | sprite_lookup procedure | //
                // +-------------------------+ //

                // STEP 1: Look through each "y-value" of sprites in sprite RAM.
                //         If a sprite is in range of the current scanline, its
                //         information should be stored in the temporary sprite
                //         memory (internal to this module).
                //
                // STEP 2: During clock cycles 256..320, use the nametable data of
                //         the stored sprites to get pattern information from
                //         the pattern tables. Then, store all the temporary sprite
                //         memory data to the sprite buffers that render pixels in
                //         real time.


                // ************** //
                // --- STEP 1 --- //
                // ************** //

                if (sprite_range_search)
                    begin
                        case (range_search_state)
                            0 :
                                begin
                                    range_comparator <= (scanline_counter - 20 - sprite_RAM_data);
                                    range_search_state <= 1;
                                end
                            1 :
                                begin
                                    if ((!sprite_size && (range_comparator <= 7)) || (sprite_size
&& (range_comparator <=15)))
                                        begin // sprite is in range

                                            if (sprite_RAM_address == 0) // primary object
found!
                                                primary_object_in_range <= 1;

                                            if (range_search_sprite_number < 8)
                                                begin
                                                    sprite_RAM_address <= sprite_RAM_address +
2;

                                                    range_search_state <= 2;
                                                    range_search_sprite_number <=
range_search_sprite_number + 1;
                                                end
                                            else
                                                begin
                                                    more_than_eight_sprites <= 1;
                                                    range_search_state <= 8;
                                                    sprite_RAM_address <= 0;
                                                end
                                        end
                                    else // sprite is not in range
                                        begin
```

```
                                    if (sprite_RAM_address >= 252) // this was the last
sprite possible
                                        begin
                                            sprite_RAM_address <= 0;
                                            temporary_sprite_RAM_address <= 1;
                                            range_search_state <= 8;
                                        end
                                    else // keep searching for more
                                        begin
                                            range_search_state <= 0;
                                            sprite_RAM_address <= sprite_RAM_address + 4;
                                        end

                                end
                            end

                        2 :
                            begin
                                if (sprite_RAM_data[7]) // vertical flipping
                                    temporary_sprite_RAM_data_in <= {sprite_RAM_data[6:5],
sprite_RAM_data[1:0], ~range_comparator[3:0]};
                                else
                                    temporary_sprite_RAM_data_in <= {sprite_RAM_data[6:5],
sprite_RAM_data[1:0], range_comparator[3:0]};
                                // NOTE: memory is stored as follows: 7) x-flip, 6) priority,
5:4) color, 3:0) row value

                                temporary_sprite_RAM_WE <= 1;
                                range_search_state <= 3;
                            end

                        3 :
                            begin
                                sprite_RAM_address <= sprite_RAM_address - 1;
                                temporary_sprite_RAM_address <= temporary_sprite_RAM_address +
1;
                                range_search_state <= 4;
                                temporary_sprite_RAM_WE <= 0;
                            end

                        4 :
                            begin
                                temporary_sprite_RAM_data_in <= sprite_RAM_data;
                                range_search_state <= 5;
                                temporary_sprite_RAM_WE <= 1;
                            end

                        5 :
                            begin
                                sprite_RAM_address <= sprite_RAM_address + 2;
                                temporary_sprite_RAM_address <= temporary_sprite_RAM_address +
1;
                                range_search_state <= 6;
                                temporary_sprite_RAM_WE <= 0;
                            end

                        6 :
                            begin
                                temporary_sprite_RAM_data_in <= sprite_RAM_data;
                                range_search_state <= 7;
                                temporary_sprite_RAM_WE <= 1;
                            end

                        7 :
                            begin
                                temporary_sprite_RAM_WE <= 0;
                                if (sprite_RAM_address >= 252) // last sprite to be read from
memory
                                    range_search_state <= 8;
                                else // keep looking for more sprites
                                    begin
```

```
                                        temporary_sprite_RAM_address <=
temporary_sprite_RAM_address + 1;
                                        sprite_RAM_address <= sprite_RAM_address + 1;
                                        range_search_state <= 0;
                                    end
                                end

                            8 : // a stalling state
                                begin
                                    range_search_state <= 8;
                                    tile_index_number <= temporary_sprite_RAM_data_out;
                                    temporary_sprite_RAM_WE <= 0; // enable reads from temporary
sprite RAM for the memory fetch
                                    temporary_sprite_RAM_address <= 1;
                                    sprite_RAM_address <= 0;
                                end

                        endcase
                    end // sprite range search



                // ************** //
                // --- STEP 2 --- //
                // ************** //

                else if (sprite_vram_memory_fetch)
                    begin
                        case(memory_fetch_state)
                            0 :
                                begin
                                    primary_object_will_be_rendered <= 0;
                                    if (sprite_fetching_number > range_search_sprite_number) //
looked through all of them
                                        memory_fetch_state <= 8;
                                    else
                                        begin
                                            temporary_sprite_RAM_address <=
temporary_sprite_RAM_address - 1;
                                            memory_fetch_state <= 1;
                                        end
                                end
                            1 :
                                begin
                                    if (sprite_size) // sprites are large
                                        PPU_address <= {1'b0, temporary_sprite_RAM_data_out[0],
tile_index_number, 1'b0, temporary_sprite_RAM_data_out[3:1]};
                                    else // sprites are small
                                        PPU_address <= {1'b0, tile_index_number, 1'b0,
temporary_sprite_RAM_data_out[2:0]}; // pattern_table_selection

                                    horizontal_flip_flag <= temporary_sprite_RAM_data_out[7];
                                    case (sprite_fetching_number)
                                        1 : sprite_0_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                        2 : sprite_1_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                        3 : sprite_2_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                        4 : sprite_3_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                        5 : sprite_4_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                        6 : sprite_5_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                        7 : sprite_6_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                        8 : sprite_7_attributes <=
temporary_sprite_RAM_data_out[6:4];
                                    endcase
                                    ALE <= 1;
```

```
                                memory_fetch_state <= 2;
                            end
                        2 :
                            begin
                                ALE <= 0;
                                RD <= 0;

                                // **** NOTE!!! **** sprite_buffers_data handling was moved to
an "assign" statement due to a time constraint on the read period

                                //if (horizontal_flip_flag)
                                //  sprite_buffers_data <= {PPU_data[0], PPU_data[1],
PPU_data[2], PPU_data[3], PPU_data[4], PPU_data[5], PPU_data[6], PPU_data[7]};
                                //else
                                //  sprite_buffers_data <= PPU_data;

                                case (sprite_fetching_number)
                                    1 : low_bit_load_register <= 8'b00000001;
                                    2 : low_bit_load_register <= 8'b00000010;
                                    3 : low_bit_load_register <= 8'b00000100;
                                    4 : low_bit_load_register <= 8'b00001000;
                                    5 : low_bit_load_register <= 8'b00010000;
                                    6 : low_bit_load_register <= 8'b00100000;
                                    7 : low_bit_load_register <= 8'b01000000;
                                    8 : low_bit_load_register <= 8'b10000000;
                                endcase

                                memory_fetch_state <= 3;
                            end
                        3 :
                            begin
                                low_bit_load_register <= 0;
                                PPU_address <= PPU_address + 8;  // REGARDLESS OF SPRITE
SIZE!!!

                                RD <= 1; // stop reading
                                ALE <= 1; // start writing address
                                memory_fetch_state <= 4;
                            end
                        4 :
                            begin
                                ALE <= 0;
                                RD <= 0;

                                //if (horizontal_flip_flag)
                                //  sprite_buffers_data <= {PPU_data[0], PPU_data[1],
PPU_data[2], PPU_data[3], PPU_data[4], PPU_data[5], PPU_data[6], PPU_data[7]};
                                //else
                                //  sprite_buffers_data <= PPU_data;

                                case (sprite_fetching_number)
                                    1 : high_bit_load_register <= 8'b00000001;
                                    2 : high_bit_load_register <= 8'b00000010;
                                    3 : high_bit_load_register <= 8'b00000100;
                                    4 : high_bit_load_register <= 8'b00001000;
                                    5 : high_bit_load_register <= 8'b00010000;
                                    6 : high_bit_load_register <= 8'b00100000;
                                    7 : high_bit_load_register <= 8'b01000000;
                                    8 : high_bit_load_register <= 8'b10000000;
                                endcase

                                memory_fetch_state <= 5;
                            end
                        5 :
                            begin
                                ALE <= 0; // write another address
                                RD <= 1; // stop reading
                                high_bit_load_register <= 0;

                                temporary_sprite_RAM_address <= temporary_sprite_RAM_address +
2;
```

```
                                memory_fetch_state <= 6;
                            end
                        6 :
                            begin
                                high_bit_load_register <= 0;
                                //sprite_buffers_data <= temporary_sprite_RAM_data_out; //
store the x-coordinate

                                case (sprite_fetching_number)
                                    1 : x_counter_load <= 8'b00000001;
                                    2 : x_counter_load <= 8'b00000010;
                                    3 : x_counter_load <= 8'b00000100;
                                    4 : x_counter_load <= 8'b00001000;
                                    5 : x_counter_load <= 8'b00010000;
                                    6 : x_counter_load <= 8'b00100000;
                                    7 : x_counter_load <= 8'b01000000;
                                    8 : x_counter_load <= 8'b10000000;
                                endcase

                                memory_fetch_state <= 7;
                            end
                        7 :
                            begin
                                x_counter_load <= 0;
                                memory_fetch_state <= 0;
                                sprite_fetching_number <= sprite_fetching_number + 1;
                                temporary_sprite_RAM_address <= temporary_sprite_RAM_address +
2;

                                tile_index_number <= temporary_sprite_RAM_data_out;
                            end

                        8 :
                            begin
                                memory_fetch_state <= 8;
                                if (primary_object_in_range)
                                    begin
                                        primary_object_will_be_rendered <= 1;
                                        primary_object_in_range <= 0;
                                    end
                            end
                    endcase
                end // memory fetch



            // ********************************** //
            // ---  sprite_rendering procedure  --- //
            // ********************************** //

            // NOTE: This happens in PARALLEL with sprite range search.

            if (sprite_rendering)  // NOTE: priority is determined by sprite number
                begin
                    if (left_side_object_clipping && (scanline_clock <= 8))
                        sprite_out <= 0;
                    else if (buffer_0_out != 0)
                        begin
                            if (primary_object_will_be_rendered)
                                primary_pixel <= 1; // tells the priority mux to look out for
primary-object-to-playfield collision

                            sprite_out <= {buffer_0_out, sprite_0_attributes};
                        end
                    else if (buffer_1_out != 0)
                        begin
                            sprite_out <= {buffer_1_out, sprite_1_attributes};
                            primary_pixel <= 0;
                        end
                    else if (buffer_2_out != 0)
                        begin
                            sprite_out <= {buffer_2_out, sprite_2_attributes};
```

```
                            primary_pixel <= 0;
                        end
                    else if (buffer_3_out != 0)
                        begin
                            sprite_out <= {buffer_3_out, sprite_3_attributes};
                            primary_pixel <= 0;
                        end
                    else if (buffer_4_out != 0)
                        begin
                            sprite_out <= {buffer_4_out, sprite_4_attributes};
                            primary_pixel <= 0;
                        end
                    else if (buffer_5_out != 0)
                        begin
                            sprite_out <= {buffer_5_out, sprite_5_attributes};
                            primary_pixel <= 0;
                        end
                    else if (buffer_6_out != 0)
                        begin
                            sprite_out <= {buffer_6_out, sprite_6_attributes};
                            primary_pixel <= 0;
                        end
                    else if (buffer_7_out != 0)
                        begin
                            sprite_out <= {buffer_7_out, sprite_7_attributes};
                            primary_pixel <= 0;
                        end
                    else
                        sprite_out <= 0;
                end // sprite_rendering

            end // not reset
        end // always block


        assign sprite_buffers_data = (memory_fetch_state == 5 || memory_fetch_state == 3) ?
                                        ((horizontal_flip_flag) ? {PPU_data[0], PPU_data[1],
PPU_data[2], PPU_data[3], PPU_data[4], PPU_data[5], PPU_data[6], PPU_data[7]} :
                                            PPU_data) :
                                        (memory_fetch_state == 7) ? temporary_sprite_RAM_data_out :
8'b00000000;


endmodule
```

ppu_sprite_buffer.v

```
module PPU_sprite_buffer(buffer_data, load_signal_low, load_signal_high, x_counter_load,
PPU_5370_clock, buffer_output, reset, hblank, buffer_output_high);
    input PPU_5370_clock, reset, load_signal_low, load_signal_high, hblank, x_counter_load;
    input [7:0] buffer_data;
    output [1:0] buffer_output;
    output [7:0] buffer_output_high;

    reg [8:0] x_clock;
    reg [1:0] buffer_output;
    reg [7:0] buffer_output_high, buffer_output_low;
    reg [4:0] shift_state;

    always @(negedge PPU_5370_clock) // NOTE! *NEG*edge so that data that is loaded only for one
clock cycle can be read
    begin

        if (reset)
            begin
                x_clock = 350;
                shift_state = 0;
                buffer_output = 2'b00;
```

```
    end
else if (!hblank)
    begin
        // load all values during hblank
        shift_state = 0;
        buffer_output = 2'b00;
        if (load_signal_low)
            buffer_output_low = buffer_data;

        if (load_signal_high)
            buffer_output_high = buffer_data;

        if (x_counter_load)
            x_clock = {1'b0, buffer_data};
    end
else // not reset or /hblank
    begin

        if (x_clock > 0)
            x_clock = x_clock - 1;
        else // x-clock == 0
            begin
                case (shift_state)
                    0:
                        begin
                            buffer_output = {buffer_output_high[7], buffer_output_low[7]};
                            shift_state = 1;
                        end
                    1:
                        begin
                            buffer_output = {buffer_output_high[6], buffer_output_low[6]};
                            shift_state = 2;
                        end
                    2:
                        begin
                            buffer_output = {buffer_output_high[5], buffer_output_low[5]};
                            shift_state = 3;
                        end
                    3:
                        begin
                            buffer_output = {buffer_output_high[4], buffer_output_low[4]};
                            shift_state = 4;
                        end
                    4:
                        begin
                            buffer_output = {buffer_output_high[3], buffer_output_low[3]};
                            shift_state = 5;
                        end
                    5:
                        begin
                            buffer_output = {buffer_output_high[2], buffer_output_low[2]};
                            shift_state = 6;
                        end
                    6:
                        begin
                            buffer_output = {buffer_output_high[1], buffer_output_low[1]};
                            shift_state = 7;
                        end
                    7:
                        begin
                            buffer_output = {buffer_output_high[0], buffer_output_low[0]};
                            shift_state = 8;
                        end
                    8:
                        begin
                            buffer_output = 2'b00;
                            buffer_output_low = 0;
                            buffer_output_high = 0;
                            x_clock = 350;
                            shift_state = 8;
                        end
```

```
                    endcase
                end // x_clock == 0
            end // not reset
        end // always block


endmodule
```

ppu_background_renderer.v

```
module PPU_background_renderer
          (reset, PPU_5370_clock, hblank, // simple inputs
           scanline_index, scanline_x_coordinate, // 9-bit inputs
           RD, ALE, // simple outputs
           background_out, // 4-bit output: two pattern bits and two color select bits
           PPU_address, // 14-bit output
           PPU_data, // 8-bit input
           left_side_background_clipping,
           x_scroll,
           y_scroll,
           x_scroll_nametable_select,
           y_scroll_nametable_select,
           playfield_pattern_table_select);

    // ----------------------------- //
    // ------ INPUT / OUTPUT  ------ //
    // ----------------------------- //

    input reset, PPU_5370_clock, hblank, x_scroll_nametable_select,
           y_scroll_nametable_select, playfield_pattern_table_select,
           left_side_background_clipping;
    input [8:0] scanline_index, scanline_x_coordinate;
    input [7:0] PPU_data, x_scroll, y_scroll;

    output RD, ALE;
    output [3:0] background_out;
    output [13:0] PPU_address;

    // ------------------------------- //
    // ------ REGISTERS & WIRES  ------ //
    // ------------------------------- //

    reg RD, ALE;
    reg [13:0] PPU_address;

    reg [1:0] pattern_data, color_index;

    assign background_out = {pattern_data, color_index};
    //assign background_out = ((scanline_index < 46) && (scanline_x_coordinate < 100) &&
(scanline_x_coordinate > 80)) ? 4'b0001 : 4'b0011;

    wire background_rendering, background_rendering_next, background_output;
    wire [3:0]  name_table;
    wire [13:0] name_table_address;
    wire [7:0]  attribute_table;
    wire [13:0] attribute_table_address;
    wire [2:0]  attribute_low_bit, attribute_high_bit;
    wire [13:0] pattern_table_address_one, pattern_table_address_two;
    wire [2:0]  fine_horizontal_scroll;

    assign fine_horizontal_scroll = x_scroll[2:0];

    reg [13:0] name_table_address_temp;
    reg [15:0] pattern_bitmap_one, pattern_bitmap_two;
    reg [15:0] palette_data_one, palette_data_two;
    reg [2:0]  fetch_phase_counter;

    reg [7:0] name_table_temp, attribute_table_temp, pattern_data_one_temp;
```

```
    // ---------------------- //
    // ------ MODULES  ------ //
    // ---------------------- //

    //EMPTRY FOR NOW

// ---------------------------------- //
// ------ BACKGROUND RENDERING  ------ //
// ---------------------------------- //

    assign background_rendering = ((scanline_index >= 21) && (scanline_index <= 260) &&
                (scanline_x_coordinate >= 1) && (scanline_x_coordinate <= 251)) ? 1 : 0;

    assign background_rendering_next = ((scanline_index >= 20) && (scanline_index <= 259) &&
                (scanline_x_coordinate >= 320) && (scanline_x_coordinate <= 335)) ? 1 : 0;

    assign background_output = ((scanline_index >= 21) && (scanline_index <= 260) &&
                (scanline_x_coordinate >= 0) && (scanline_x_coordinate <= 255)) ? 1 : 0;
// ---------------------------------- //
// ------ BACKGROUND RENDERING  ------ //
// ---------------------------------- //

// Documentation from "NES Documentation" by yoshi@parodius.com
// Diagram of the Memory Map accessed by these fetch modules!
//    +---------+-------+-------+--------------------+
//    | Address | Size  | Flags | Description        |
//    +---------+-------+-------+--------------------+
//    | $0000   | $1000 | C     | Pattern Table #0   |
//    | $1000   | $1000 | C     | Pattern Table #1   |
//    | $2000   | $3C0  |       | Name Table #0      |
//    | $23C0   | $40   |       | Attribute Table #0 |
//    | $2400   | $3C0  |  M    | Name Table #1      |
//    | $27C0   | $40   |       | Attribute Table #1 |
//    | $2800   | $3C0  |  M    | Name Table #2      |
//    | $2BC0   | $40   |       | Attribute Table #2 |
//    | $2C00   | $3C0  |  M    | Name Table #3      |
//    | $2FC0   | $40   |       | Attribute Table #3 |
//    | $3000   | $F00  |   U   |                    |
//    | $3F00   | $10   |       | Image Palette #1   |
//    | $3F10   | $10   |       | Sprite Palette #1  |
//    | $3F20   | $E0   |    P  | Palette Mirror     |
//    +---------+-------+-------+--------------------+

    // ---------------------------------- //
    // ---Name table address constructor--- //
    // ---------------------------------- //

    assign name_table = y_scroll_nametable_select ?
        (x_scroll_nametable_select ? 4'b1011 : 4'b1010) : (x_scroll_nametable_select ? 4'b1001 :
4'b1000);

    assign name_table_address = {name_table, y_scroll[7:3], x_scroll[7:3]};
    //assign name_table_address = 14'b10000000000000;

    // ---------------------------------------- //
    // ---Attribute table address constructor--- //
    // ---------------------------------------- //

    assign attribute_table = y_scroll_nametable_select ?
        (x_scroll_nametable_select ? 8'b10111111 : 8'b10101111) : (x_scroll_nametable_select ?
8'b10011111 : 8'b10001111);

    assign attribute_table_address = {attribute_table, y_scroll[7:5], x_scroll[7:5]};
    //assign attribute_table_address = 14'b10001111000000;

    // -------------------------------- //
    // ---Attribute byte bit selector--- //
    // -------------------------------- //

    // Documentation from "NES Documentation" by yoshi@parodius.com
```

```
    //      +------------+------------+
    //      |  Square 0  |  Square 1  |  #0-F represents an 8x8 tile
    //      |   #0  #1   |   #4  #5   |
    //      |   #2  #3   |   #6  #7   |  Square [x] represents four (4) 8x8 tiles
    //      +------------+------------+   (i.e. a 16x16 pixel grid)
    //      |  Square 2  |  Square 3  |
    //      |   #8  #9   |   #C  #D   |
    //      |   #A  #B   |   #E  #F   |
    //      +------------+------------+
    //
    //    The actual format of the attribute byte is the following (and corris-
    //    ponds to the above example):
    //
    //       Attribute Byte
    //         (Square #)
    //      ----------------
    //          33221100
    //          ||||||+--- Upper two (2) colour bits for Square 0 (Tiles #0,1,2,3)
    //          ||||+----- Upper two (2) colour bits for Square 1 (Tiles #4,5,6,7)
    //          ||+------- Upper two (2) colour bits for Square 2 (Tiles #8,9,A,B)
    //          +--------- Upper two (2) colour bits for Square 3 (Tiles #C,D,E,F)

    assign attribute_low_bit = name_table_address_temp[1] ?
        (name_table_address_temp[6] ? 6 : 2) : (name_table_address_temp[6] ? 4 : 0);

    assign attribute_high_bit = name_table_address_temp[1] ?
        (name_table_address_temp[6] ? 7 : 3) : (name_table_address_temp[6] ? 5 : 1);

    // -------------------------------------- //
    // ---Pattern table address constructor--- //
    // -------------------------------------- //

    assign pattern_table_address_one = {1'b0, playfield_pattern_table_select, name_table_temp,
1'b0, y_scroll[2:0]};
    assign pattern_table_address_two = {1'b0, playfield_pattern_table_select, name_table_temp,
1'b1, y_scroll[2:0]};

    // -------------------------------------- //
    // ---Main always loop on system clock--- //
    // -------------------------------------- //

    always @ (posedge PPU_5370_clock)
    begin
        if (reset)
            begin
                //blank output
                pattern_data <= 0;
                color_index  <= 0;

                // reset registers and outputs
                ALE <= 0;  // address_latch_enable inactive
                RD <= 1;   // RD(not) is inactive
                pattern_bitmap_one  <= 0;
                pattern_bitmap_two  <= 0;
                palette_data_one   <= 0;
                palette_data_two   <= 0;
                fetch_phase_counter <= 0;

                // reset temp stuff
                name_table_address_temp   <= 0;
                name_table_temp           <= 0;
                attribute_table_temp  <= 0;
                pattern_data_one_temp  <= 0;

                // reset all other registers
                PPU_address <= 0;
                pattern_data <= 0;
                color_index <= 0;

            end
```

```
        else
            begin
            if (background_rendering_next || background_rendering)

                case(fetch_phase_counter)
                    0 : begin  //name table byte fetch, cc 1
                            PPU_address <= name_table_address; //LATCH NAME ADDRESS
                            name_table_address_temp <= name_table_address; // store temp address
                            ALE <= 1;  // address_latch_enable active
                            RD <= 1;   // RD(not) is inactive

                            //load latched data at the beginning (7 to 0 transition) of each new
title fetch phase

                            pattern_bitmap_two <= {PPU_data,pattern_bitmap_two[8:1]};
                            pattern_bitmap_one <= {pattern_data_one_temp,pattern_bitmap_one[8:1]};

                            //shift over palette_data_one and palette_data_two in preparation for
the new byte
                            palette_data_one[7:0] <= palette_data_one[8:1];
                            palette_data_two[7:0] <= palette_data_two[8:1];

                            case(attribute_low_bit)
                                0 : begin
                                        palette_data_one[15] <= attribute_table_temp[0];
                                        palette_data_one[14] <= attribute_table_temp[0];
                                        palette_data_one[13] <= attribute_table_temp[0];
                                        palette_data_one[12] <= attribute_table_temp[0];
                                        palette_data_one[11] <= attribute_table_temp[0];
                                        palette_data_one[10] <= attribute_table_temp[0];
                                        palette_data_one[9]  <= attribute_table_temp[0];
                                        palette_data_one[8]  <= attribute_table_temp[0];
                                    end
                                2 : begin
                                        palette_data_one[15] <= attribute_table_temp[2];
                                        palette_data_one[14] <= attribute_table_temp[2];
                                        palette_data_one[13] <= attribute_table_temp[2];
                                        palette_data_one[12] <= attribute_table_temp[2];
                                        palette_data_one[11] <= attribute_table_temp[2];
                                        palette_data_one[10] <= attribute_table_temp[2];
                                        palette_data_one[9]  <= attribute_table_temp[2];
                                        palette_data_one[8]  <= attribute_table_temp[2];
                                    end
                                4 : begin
                                        palette_data_one[15] <= attribute_table_temp[4];
                                        palette_data_one[14] <= attribute_table_temp[4];
                                        palette_data_one[13] <= attribute_table_temp[4];
                                        palette_data_one[12] <= attribute_table_temp[4];
                                        palette_data_one[11] <= attribute_table_temp[4];
                                        palette_data_one[10] <= attribute_table_temp[4];
                                        palette_data_one[9]  <= attribute_table_temp[4];
                                        palette_data_one[8]  <= attribute_table_temp[4];
                                    end
                                6 : begin
                                        palette_data_one[15] <= attribute_table_temp[6];
                                        palette_data_one[14] <= attribute_table_temp[6];
                                        palette_data_one[13] <= attribute_table_temp[6];
                                        palette_data_one[12] <= attribute_table_temp[6];
                                        palette_data_one[11] <= attribute_table_temp[6];
                                        palette_data_one[10] <= attribute_table_temp[6];
                                        palette_data_one[9]  <= attribute_table_temp[6];
                                        palette_data_one[8]  <= attribute_table_temp[6];
                                    end
                            endcase

                            case(attribute_high_bit)
                                1 : begin
                                        palette_data_two[15] <= attribute_table_temp[1];
                                        palette_data_two[14] <= attribute_table_temp[1];
                                        palette_data_two[13] <= attribute_table_temp[1];
                                        palette_data_two[12] <= attribute_table_temp[1];
```

```
                    palette_data_two[11] <= attribute_table_temp[1];
                    palette_data_two[10] <= attribute_table_temp[1];
                    palette_data_two[9]  <= attribute_table_temp[1];
                    palette_data_two[8]  <= attribute_table_temp[1];
                end
            3 : begin
                    palette_data_two[15] <= attribute_table_temp[3];
                    palette_data_two[14] <= attribute_table_temp[3];
                    palette_data_two[13] <= attribute_table_temp[3];
                    palette_data_two[12] <= attribute_table_temp[3];
                    palette_data_two[11] <= attribute_table_temp[3];
                    palette_data_two[10] <= attribute_table_temp[3];
                    palette_data_two[9]  <= attribute_table_temp[3];
                    palette_data_two[8]  <= attribute_table_temp[3];
                end
            5 : begin
                    palette_data_two[15] <= attribute_table_temp[5];
                    palette_data_two[14] <= attribute_table_temp[5];
                    palette_data_two[13] <= attribute_table_temp[5];
                    palette_data_two[12] <= attribute_table_temp[5];
                    palette_data_two[11] <= attribute_table_temp[5];
                    palette_data_two[10] <= attribute_table_temp[5];
                    palette_data_two[9]  <= attribute_table_temp[5];
                    palette_data_two[8]  <= attribute_table_temp[5];
                end
            7 : begin
                    palette_data_two[15] <= attribute_table_temp[7];
                    palette_data_two[14] <= attribute_table_temp[7];
                    palette_data_two[13] <= attribute_table_temp[7];
                    palette_data_two[12] <= attribute_table_temp[7];
                    palette_data_two[11] <= attribute_table_temp[7];
                    palette_data_two[10] <= attribute_table_temp[7];
                    palette_data_two[9]  <= attribute_table_temp[7];
                    palette_data_two[8]  <= attribute_table_temp[7];
                end
        endcase

        //increment to next case
        fetch_phase_counter <= 1;
    end
1 : begin  //name table byte fetch, cc 2
        ALE <= 0;  // address_latch_enable inavtive
        RD <= 0;   // RD(not) is active

        //increment to next case
        fetch_phase_counter <= 2;
    end
2 : begin  //attribute table byte fetch, cc 1
         name_table_temp <= PPU_data;
        PPU_address <= attribute_table_address; //LATCH ATTRIBUTE ADDRESS
        ALE <= 1;  // address_latch_enable active
        RD <= 1;   // RD(not) is inactive

        //increment to next case
        fetch_phase_counter <= 3;
    end
3 : begin  //attribute table byte fetch, cc 2
        ALE <= 0;  // address_latch_enable inavtive
        RD <= 0;   // RD(not) is active

        //increment to next case
        fetch_phase_counter <= 4;
    end
4 : begin  //pattern table bitmap #1 fetch, cc 1
        attribute_table_temp <= PPU_data;
        PPU_address <= pattern_table_address_one; //LATCH PATTERN BIT 1 ADDRESS
        ALE <= 1;  // address_latch_enable active
        RD <= 1;   // RD(not) is inactive

        //increment to next case
        fetch_phase_counter <= 5;
```

```
                    end
                5 : begin  //pattern table bitmap #1 fetch, cc 2
                        ALE <= 0;  // address_latch_enable inavtive
                        RD <= 0;   // RD(not) is active

                        //increment to next case
                        fetch_phase_counter <= 6;
                    end
                6 : begin  //pattern table bitmap #2, cc 1
                        pattern_data_one_temp <= PPU_data;
                        PPU_address <= pattern_table_address_two; //LATCH PATTERN BIT 2 ADDRESS
                        ALE <= 1;  // address_latch_enable active
                        RD <= 1;   // RD(not) is inactive

                        //increment to next case
                        fetch_phase_counter <= 7;
                    end
                7 : begin  //pattern table bitmap #2, cc 2
                        ALE <= 0;  // address_latch_enable inavtive
                        RD <= 0;   // RD(not) is active

                        //increment to next case
                        fetch_phase_counter <= 0;
                    end
            endcase

        // when background is not being rendered, set the state machine counter to zero
        else
            begin
                fetch_phase_counter <= 0;
                ALE <= 0;  // address_latch_enable inactive
                RD <= 1;   // RD(not) is inactive
            end

        // shift the pattern_bitmap and pallete_select_data right one bit
        // This happens only when it is NOT fetch_phase zero
        // That phase must write to pattern_bitmap and palette_data, so it does the
incrementing itself
        if ((fetch_phase_counter != 0) && (background_output || background_rendering_next))
            begin
                pattern_bitmap_one <= {1'b0, pattern_bitmap_one[15:1]};
                pattern_bitmap_two <= {1'b0, pattern_bitmap_two[15:1]};
                palette_data_one <= {1'b0, palette_data_one[15:1]};
                palette_data_two <= {1'b0, palette_data_two[15:1]};
            end

        if (background_output)
            begin
                // set the PPU data output
                case (fine_horizontal_scroll)
                    0 : begin
                            pattern_data[1] <= pattern_bitmap_two[0];
                            pattern_data[0] <= pattern_bitmap_one[0];
                            color_index[1]  <= palette_data_two[0];
                            color_index[0]  <= palette_data_one[0];
                        end
                    1 : begin
                            pattern_data[1] <= pattern_bitmap_two[1];
                            pattern_data[0] <= pattern_bitmap_one[1];
                            color_index[1]  <= palette_data_two[1];
                            color_index[0]  <= palette_data_one[1];
                        end
                    2 : begin
                            pattern_data[1] <= pattern_bitmap_two[2];
                            pattern_data[0] <= pattern_bitmap_one[2];
                            color_index[1]  <= palette_data_two[2];
                            color_index[0]  <= palette_data_one[2];
                        end
                    3 : begin
                            pattern_data[1] <= pattern_bitmap_two[3];
                            pattern_data[0] <= pattern_bitmap_one[3];
```

```
                            color_index[1]  <= palette_data_two[3];
                            color_index[0]  <= palette_data_one[3];
                        end
                    4 : begin
                            pattern_data[1] <= pattern_bitmap_two[4];
                            pattern_data[0] <= pattern_bitmap_one[4];
                            color_index[1]  <= palette_data_two[4];
                            color_index[0]  <= palette_data_one[4];
                        end
                    5 : begin
                            pattern_data[1] <= pattern_bitmap_two[5];
                            pattern_data[0] <= pattern_bitmap_one[5];
                            color_index[1]  <= palette_data_two[5];
                            color_index[0]  <= palette_data_one[5];
                        end
                    6 : begin
                            pattern_data[1] <= pattern_bitmap_two[6];
                            pattern_data[0] <= pattern_bitmap_one[6];
                            color_index[1]  <= palette_data_two[6];
                            color_index[0]  <= palette_data_one[6];
                        end
                    7 : begin
                            pattern_data[1] <= pattern_bitmap_two[7];
                            pattern_data[0] <= pattern_bitmap_one[7];
                            color_index[1]  <= palette_data_two[7];
                            color_index[0]  <= palette_data_one[7];
                        end
                endcase
            end // if statement determining background_output
        // If this is not background_output time, set background_out to all zeros
        else
            begin
                pattern_data[1] <= 0;
                pattern_data[0] <= 0;
                color_index[1]  <= 0;
                color_index[0]  <= 0;
            end // else statment for background_output generator

        end // else statement for the not-reset condition

    end // main always block for fetch and output

endmodule
```

ppu_test_controller.v

```
module PPU_test_controller(clock, reset, vblank, address, data_out, DBE, R_W);
    input clock, reset, vblank;
    output [2:0] address;
    output [7:0] data_out;
    output DBE, R_W;

    reg [2:0] clock_counter;
    reg slow_clock;
    reg [6:0] ROM_address;
    reg [1:0] state;
    reg [2:0] address;
    reg [7:0] data;
    reg R_W, DBE, R_W_flag;
    reg vblank_previous;
    reg [6:0] counter;

    wire [7:0] ROM_out;

    assign data_out = (!DBE) ? data : 0;
    //-----
    PPU_controller_ROM controller_ROM(ROM_address, ROM_out);
    //-----
```

```
    always @(posedge clock)
        begin
            if (reset)
                begin
                    clock_counter <= 1;
                    slow_clock <= 0;
                    ROM_address <= 0;
                    state <= 0;
                end
            else if (clock_counter < 6)
                clock_counter <= clock_counter + 1;
            else
                begin
                    clock_counter <= 1;
                    vblank_previous <= vblank;
                        begin
                            case (state)
                                0 :
                                    begin
                                        R_W_flag <= ROM_out[4];
                                        DBE <= 1;
                                        address <= ROM_out[3:0];
                                        if (ROM_out == 8) // wait for vblank
                                            begin
                                                ROM_address <= ROM_address + 1;
                                                state <= 2;
                                            end
                                        else if (ROM_out == 9) // stop processing
                                            state <= 3;
                                        else
                                            begin
                                                ROM_address <= ROM_address + 1;
                                                state <= 1;
                                            end
                                    end
                                1 :
                                    begin
                                        data <= ROM_out;
                                        ROM_address <= ROM_address + 1;
                                        state <= 0;
                                        R_W <= 0; // R_W_flag; // change this if you want to read
                                        DBE <= 0;
                                    end
                                2 :
                                    begin
                                        if (!vblank && vblank_previous)
                                            counter <= counter + 1;
                                        else if (counter == 30)
                                            begin
                                                state <= 0;
                                                counter <= 0;
                                            end
                                    end
                                3:
                                    state <= 3;
                            endcase

                        end
                end
        end

endmodule
```

colortable.v

```
module colortable(index,red,green,blue);
```

```
    input  [5:0] index;
    output [7:0] red,green,blue;

    table_red   instance_red(index, red);
    table_green instance_green(index, green);
    table_blue  instance_blue(index, blue);

endmodule
```

ppu_clock_divider.v

```
module PPU_clock_divider (clock, PPU_5370_kHz);
    input clock;
    output PPU_5370_kHz;

    reg counter;
    reg PPU_5370_kHz;

    always @(posedge clock)
        begin
            if (counter)
                begin
                    counter <= 0;
                    PPU_5370_kHz <= ~PPU_5370_kHz;
                end
            else
                counter <= 1;
        end
endmodule
```

Appendix F: PPU ROM Files

table_red.mif

```
WIDTH = 6;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE %
DEPTH = 64;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL VALUE %


ADDRESS_RADIX = HEX;  % Address and data radixes are optional, default is hex %
DATA_RADIX = BIN;     % Valid radixes = BIN,DEC,HEX or OCT  %

CONTENT BEGIN

00 : 011111; % ADDRESS :  VALUE %
01 : 000000;
02 : 000000;
03 : 010000;
04 : 101000;
05 : 110001;
06 : 101110;
07 : 100010;
08 : 010110;
09 : 000011;
0A : 000001;
0B : 000000;
0C : 000000;
0D : 000000;
0E : 000001;
0F : 000001;
10 : 110001;
11 : 000000;
12 : 000111;
13 : 100000;
14 : 111010;
15 : 111111;
16 : 111111;
17 : 110101;
18 : 110000;
19 : 001101;
1A : 000001;
1B : 000000;
1C : 000000;
1D : 000111;
1E : 000010;
1F : 000010;
20 : 111111;
21 : 000011;
22 : 011001;
23 : 110100;
24 : 111111;
25 : 111111;
26 : 111111;
27 : 111111;
28 : 111110;
29 : 100111;
2A : 001010;
2B : 000010;
2C : 000001;
2D : 010111;
2E : 000010;
2F : 000010;
30 : 111111;
31 : 101001;
32 : 101100;
33 : 110110;
```

```
34 : 111111;
35 : 111111;
36 : 111111;
37 : 111111;
38 : 111111;
39 : 110101;
3A : 101001;
3B : 101000;
3C : 100110;
3D : 110111;
3E : 000100;
3F : 000100;

END;
```

table_green.mif

```
WIDTH = 6;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE %
DEPTH = 64;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL VALUE %


ADDRESS_RADIX = HEX;  % Address and data radixes are optional, default is hex %
DATA_RADIX = BIN;     % Valid radixes = BIN,DEC,HEX or OCT  %

CONTENT BEGIN

00 : 011111; % ADDRESS :  VALUE %
01 : 001111;
02 : 000100;
03 : 000000;
04 : 000000;
05 : 000000;
06 : 000001;
07 : 000101;
08 : 001011;
09 : 010001;
0A : 010010;
0B : 010001;
0C : 001111;
0D : 000000;
0E : 000001;
0F : 000001;
10 : 110001;
11 : 011101;
12 : 010101;
13 : 001101;
14 : 001011;
15 : 001010;
16 : 001000;
17 : 001100;
18 : 011000;
19 : 011111;
1A : 100011;
1B : 100010;
1C : 100110;
1D : 000111;
1E : 000010;
1F : 000010;
20 : 111111;
21 : 110101;
22 : 101000;
23 : 011111;
24 : 010001;
25 : 010111;
26 : 100001;
27 : 100110;
28 : 101110;
29 : 111000;
```

```
2A : 111100;
2B : 111100;
2C : 111110;
2D : 010111;
2E : 000010;
2F : 000010;
30 : 111111;
31 : 111110;
32 : 111011;
33 : 101010;
34 : 101001;
35 : 101010;
36 : 110100;
37 : 111011;
38 : 111101;
39 : 111001;
3A : 111011;
3B : 111100;
3C : 111111;
3D : 110111;
3E : 000100;
3F : 000100;

END;
```

table_blue.mif

```
WIDTH = 6;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE %
DEPTH = 64;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL VALUE %


ADDRESS_RADIX = HEX;  % Address and data radixes are optional, default is hex %
DATA_RADIX = BIN;     % Valid radixes = BIN,DEC,HEX or OCT  %

CONTENT BEGIN

00 : 011111; % ADDRESS :  VALUE %
01 : 101001;
02 : 101011;
03 : 100101;
04 : 010111;
05 : 001001;
06 : 000000;
07 : 000000;
08 : 000000;
09 : 000000;
0A : 000000;
0B : 001011;
0C : 011001;
0D : 000000;
0E : 000001;
0F : 000001;
10 : 110001;
11 : 100110;
12 : 111111;
13 : 111110;
14 : 101101;
15 : 010011;
16 : 000000;
17 : 000000;
18 : 000000;
19 : 000000;
1A : 000000;
1B : 010101;
1C : 110010;
1D : 000111;
1E : 000010;
1F : 000010;
```

```
20 : 111111;
21 : 111111;
22 : 111111;
23 : 111111;
24 : 111100;
25 : 100010;
26 : 001100;
27 : 000100;
28 : 000111;
29 : 000011;
2A : 001101;
2B : 101000;
2C : 111111;
2D : 010111;
2E : 000010;
2F : 000010;
30 : 111111;
31 : 111111;
32 : 111111;
33 : 111010;
34 : 111110;
35 : 101100;
36 : 101011;
37 : 101001;
38 : 100110;
39 : 100100;
3A : 101011;
3B : 110110;
3C : 111110;
3D : 110111;
3E : 000100;
3F : 000100;

END;
```

ppu_control_rom.mif

```
WIDTH = 8;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE %
DEPTH = 128;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL VALUE %


ADDRESS_RADIX = DEC;  % Address and data radixes are optional, default is hex %
DATA_RADIX = BIN;     % Valid radixes = BIN,DEC,HEX or OCT  %

CONTENT BEGIN
 % NOTE! Every even address number, including 0, contains a PPU address.
         Every odd  address number contains data to be sent to that address.
         An address of "8" will cause the module to stop writing until the next vblank %

        % wait for first vblank %
        00 : 00001000;

        % tell R2000, R2001 how PPU should work %
        01 : 00000000;
        02 : 10000000;
        03 : 00000001;
        04 : 00011000;

        % write sprite data %
        05 : 00000100;  %R2004 is used to write SPRITE data%
        06 : 01000000;  % y-coordinate%
        07 : 00000100;
        08 : 00000001;  % second pattern : arrow %
        09 : 00000100;
        10 : 00100000;  % attributes: no flip, background priority, 0th palette %
        11 : 00000100;
        12 : 00000000;  % x-coordinate%
        13 : 00000110;  % R2006, write address $23C0 to reach the ATTRIBUTE table %
```

```
        14 : 00100011;
        15 : 00000110;
        16 : 11000000;  % first attribute address written %
        17 : 00000111;  % write data with R2007 %
        18 : 00000000;  % ----------  use the 0th palette for first four tiles ------------ %
        19 : 00000111;
        20 : 01010101;  % --------------use 1st palette for second four tiles --------------%
        21 : 00000110;  % write another address to R2006, this time to reach the NAMETABLE %
        22 : 00100000;
        23 : 00000110;
        24 : 00000000;  % write to byte 0 %
        25 : 00000111;
        26 : 00000001;  % ----------------  first tile holds the background vertical pattern-----
--  0: nothing, 1: arrow, 2: vert grad, 3: hor grad%
        27 : 00000110;  % write background palette data %
        28 : 00111111;
        29 : 00000110;
        30 : 00000000;  % background color address %
        31 : 00000111;
        32 : 00010110;  % ---------background: white %
        33 : 00000111;
        34 : 00010110;  % ---------first color: red %
        35 : 00000111;
        36 : 00100010;  % ---------second color: light blue %
        37 : 00000111;
        38 : 00101100;  % ---------third color: light green %
        39 : 00000110;  % jump to sprite palette %
        40 : 00111111;
        41 : 00000110;
        42 : 00010000;
        43 : 00000111;
        44 : 00010110;  % background color: red %
        45 : 00000111;
        46 : 00001101;  % first color: black %
        47 : 00000111;
        48 : 00101000;  % second color: yellow %
        49 : 00000111;
        50 : 00100111;  % third color: dark yellow %
        51 : 00001000;  % 8 = WAIT UNTIL NEXT VBLANK!!! %
        52 : 00000011;  %R2003%
        53 : 00000011;
        54 : 00000100;  %R2004%
        55 : 00000001;
        56 : 00001000;
        57 : 00000011;  %R2003%
        58 : 00000011;
        59 : 00000100;  %R2004%
        60 : 00000010;
        61 : 00001000;
        62 : 00000011;  %R2003%
        63 : 00000011;
        64 : 00000100;  %R2004%
        65 : 00000011;
        66 : 00001000;
        67 : 00000011;  %R2003%
        68 : 00000011;
        69 : 00000100;  %R2004%
        70 : 00000100;
        71 : 00001000;
        72 : 00000011;  %R2003%
        73 : 00000011;
        74 : 00000100;  %R2004%
        75 : 00000101;
        76 : 00001000;
        77 : 00000011;  %R2003%
        78 : 00000011;
        79 : 00000100;  %R2004%
        80 : 00000110;
        81 : 00001000;
        82 : 00000011;  %R2003%
        83 : 00000011;
```

```
        84 : 00000100;  %R2004%
        85 : 00000111;
        86 : 00001000;
        87 : 00000011;  %R2003%
        88 : 00000011;
        89 : 00000100;  %R2004%
        90 : 00001000;
        91 : 00001000;
        92 : 00000011;  %R2003%
        93 : 00000011;
        94 : 00000100;  %R2004%
        95 : 00001001;
        96 : 00001000;
        97 : 00000011;  %R2003%
        98 : 00000011;
        99 : 00000100;  %R2004%
       100 : 00001010;
       101 : 00001000;
       102 : 00000011;  %R2003%
       103 : 00000011;
       104 : 00000100;  %R2004%
       105 : 00001011;
       106 : 00001000;
       107 : 00000011;  %R2003%
       108 : 00000010;  % give new attribute %
       109 : 00000100;  %R2004%
       110 : 01000000;  % horizontal flip, HIGH priority %
       111 : 00001000;
       112 : 00000011;  %R2003%
       113 : 00000011;
       114 : 00000100;  %R2004%
       115 : 00001010;
       116 : 00001000;
       117 : 00000011;  %R2003%
       118 : 00000011;
       119 : 00000100;  %R2004%
       120 : 00001001;
       121 : 00001000;
       122 : 00000011;  %R2003%
       123 : 00000010;
       124 : 00000100;  %R2004%
       125 : 11000000;
       126 : 00001001;
       127 : 00001001;

END;
```

ppu_chr_rom.mif

```
% This MIF file has been autogenerated by the Team Nintendo Pattern Table Generator %
% The file contains the memory description of a pattern table to be used as a Character Rom %
% This pattern table contains 4 8x8 tiles. %

WIDTH =   8;      % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE %
DEPTH =  64;    % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL VALUE %


ADDRESS_RADIX = HEX;  % Address and data radixes are optional, default is hex %
DATA_RADIX =    HEX;  % Valid radixes = BIN,DEC,HEX or OCT  %

CONTENT BEGIN

% Pattern Table Index #0 : EMPTY TILE %
       0      :      0;
       1      :      0;
       2      :      0;
       3      :      0;
       4      :      0;
```

```
        5       :       0;
        6       :       0;
        7       :       0;
        8       :       0;
        9       :       0;
        A       :       0;
        B       :       0;
        C       :       0;
        D       :       0;
        E       :       0;
        F       :       0;
% Pattern Table Index #1 : ARROW %
        10      :       0;
        11      :       78;
        12      :       4;
        13      :       2;
        14      :       FF;
        15      :       2;
        16      :       4;
        17      :       8;
        18      :       0;
        19      :       0;
        1A      :       0;
        1B      :       2;
        1C      :       3;
        1D      :       2;
        1E      :       0;
        1F      :       0;
% Pattern Table Index #2 : SPRITE GRADIENT %
        20      :       CC;
        21      :       CC;
        22      :       CC;
        23      :       CC;
        24      :       CC;
        25      :       CC;
        26      :       CC;
        27      :       CC;
        28      :       F0;
        29      :       F0;
        2A      :       F0;
        2B      :       F0;
        2C      :       F0;
        2D      :       F0;
        2E      :       F0;
        2F      :       F0;
% Pattern Table Index #3 : BACKGROUND GRADIENT %
        30      :       3;
        31      :       3;
        32      :       3;
        33      :       3;
        34      :       3;
        35      :       3;
        36      :       3;
        37      :       3;
        38      :       0;
        39      :       0;
        3A      :       0;
        3B      :       0;
        3C      :       0;
        3D      :       0;
        3E      :       0;
        3F      :       0;

% End of File %

END;
```

Appendix G: Super Steal'em Tic-Tac-Toe Source Code

```
ctrl1          equ     $00   ; Right Left Down Up Start Select B A
ctrl2          equ     $01
frame          equ     $02

red_out        equ     $03  ; +2  top row
green_out      equ     $06  ; +2  top row

temp           equ     $09
int_flag       equ     $0A

count          equ     $0B
menu_frame     equ     $0C

player         equ     $0D
winner         equ     $0E
prev_ctrl      equ     $0F

cursorx        equ     $10 ; +1
cursory        equ     $12 ; +1

gameboard      equ     $20
                       ; +8


control        equ     $30
sw_player_flag equ     $31
flash_count    equ     $32
flash          equ     $33
mask1          equ     $34
mask2          equ     $35
player_start   equ     $36

steals         equ     $50 ; +1
lastx          equ     $52 ; +1
lasty          equ     $54 ; +1
game_mode      equ     $56


data_out       equ     $4018
row_out        equ     $4019
timer          equ     $4020



;-------------------------------------
       org     $F800
reset_v:

       sei             ; disable interrupts
       ldx     #$ff
       txs             ; reset the stack

       lda     #$FF
       sta     player_start

setup:

       lda     #$00    ; initialize controller variables
       sta     ctrl1
       sta     ctrl2

       lda     #$03    ; initialize frame
       sta     frame

       lda     #$00
       sta     int_flag
```

```
        lda     #$55    ; clear timer
        sta     timer
        cli             ; enable interrupts


start:  lda     #$20
        sta     count
        lda     #$08
        sta     menu_frame

menu:
        lda     #$00
        sta     int_flag

        dec     count
        bne     display_frame

        lda     #$20
        sta     count
        dec     menu_frame
        bne     display_frame

        lda     #$08
        sta     menu_frame

display_frame:
        ldx     menu_frame
        dex
        lda     menu_red_row_1, X
        sta     red_out
        ldx     menu_frame
        dex
        lda     menu_red_row_2, X
        sta     red_out+1
        ldx     menu_frame
        dex
        lda     menu_red_row_3, X
        sta     red_out+2

        ldx     menu_frame
        dex
        lda     menu_grn_row_1, X
        sta     green_out
        ldx     menu_frame
        dex
        lda     menu_grn_row_2, X
        sta     green_out+1
        ldx     menu_frame
        dex
        lda     menu_grn_row_3, X
        sta     green_out+2

check_for_start:
        lda     ctrl1
        ora     ctrl2
        and     #$0C
        bne     begin_game


m_wait: lda     int_flag        ; wait for synch int
        beq     m_wait

        jmp     menu

;-------------------------------------
begin_game:
        lda     player_start
        sec
        sbc     #$FF
        beq     pick_player_to_start
```

```
        lda     player_start
        beq     start_player1
        jmp     start_player2

pick_player_to_start:
        lda     ctrl1
        and     #$08
        bne     start_player1

start_player1:
        lda     #$00
        sta     player
        jmp     clear_gameboard


start_player2:
        lda     #$01
        sta     player

clear_gameboard:
        lda     #$00
        sta     game_mode

        lda     ctrl1
        ora     ctrl2
        and     #$04
        beq     normal_mode

        lda     #$01
        sta     game_mode

normal_mode:
        lda     #$00
        sta     prev_ctrl

        sta     gameboard       ; clear gameboard
        sta     gameboard + 1
        sta     gameboard + 2
        sta     gameboard + 3
        sta     gameboard + 4
        sta     gameboard + 5
        sta     gameboard + 6
        sta     gameboard + 7
        sta     gameboard + 8

        lda     #$01
        sta     cursorx
        sta     cursory

        lda     #$FF
        sta     lastx
        sta     lastx+1
        sta     lasty
        sta     lasty+1

        lda     #$03
        sta     steals
        sta     steals+1

        lda     #$00
        sta     sw_player_flag
        sta     flash
        lda     #$0A
        sta     flash_count

game_loop:
        lda     #$00
        sta     int_flag

        lda     sw_player_flag
```

```
        beq     donot_switch
        jsr     switch_player

donot_switch:
        lda     #$00
        sta     sw_player_flag

        jsr     user_input

        jsr     display_board

        jsr     check_for_end
        beq     game_wait
        jmp     game_end

game_wait:
        lda     int_flag        ; wait for synch int
        beq     game_wait

        jmp     game_loop
;-----------------------------------
display_board:
        lda     #$00
        sta     red_out
        sta     red_out+1
        sta     red_out+2
        sta     green_out
        sta     green_out+1
        sta     green_out+2

        ldx     gameboard
        dex
        bne     db1
        lda     green_out+0
        ora     #%00000100
        sta     green_out+0
db1:
        dex
        bne     db2
        lda     red_out+0
        ora     #%00000100
        sta     red_out+0
db2:

        ldx     gameboard+1
        dex
        bne     db3
        lda     green_out+0
        ora     #%00000010
        sta     green_out+0
db3:
        dex
        bne     db4
        lda     red_out+0
        ora     #%00000010
        sta     red_out+0
db4:


        ldx     gameboard+2
        dex
        bne     db5
        lda     green_out+0
        ora     #%00000001
        sta     green_out+0
db5:
        dex
        bne     db6
        lda     red_out+0
        ora     #%00000001
        sta     red_out+0
```

```
db6:


        ldx     gameboard+3
        dex
        bne     db7
        lda     green_out+1
        ora     #%00000100
        sta     green_out+1
db7:
        dex
        bne     db8
        lda     red_out+1
        ora     #%00000100
        sta     red_out+1
db8:

        ldx     gameboard+4
        dex
        bne     db9
        lda     green_out+1
        ora     #%00000010
        sta     green_out+1
db9:
        dex
        bne     db10
        lda     red_out+1
        ora     #%00000010
        sta     red_out+1
db10:


        ldx     gameboard+5
        dex
        bne     db11
        lda     green_out+1
        ora     #%00000001
        sta     green_out+1
db11:
        dex
        bne     db12
        lda     red_out+1
        ora     #%00000001
        sta     red_out+1
db12:


        ldx     gameboard+6
        dex
        bne     db13
        lda     green_out+2
        ora     #%00000100
        sta     green_out+2
db13:
        dex
        bne     db14
        lda     red_out+2
        ora     #%00000100
        sta     red_out+2
db14:


        ldx     gameboard+7
        dex
        bne     db15
        lda     green_out+2
        ora     #%00000010
        sta     green_out+2
db15:
        dex
        bne     db16
```

```
        lda     red_out+2
        ora     #%00000010
        sta     red_out+2
db16:


        ldx     gameboard+8
        dex
        bne     db17
        lda     green_out+2
        ora     #%00000001
        sta     green_out+2
db17:
        dex
        bne     db18
        lda     red_out+2
        ora     #%00000001
        sta     red_out+2
db18:

draw_cursor:
        lda     #%11111011
        ldx     cursorx
        beq     finish_mask
        ror     A
        dex
        beq     finish_mask
        ror     A
finish_mask:
        sta     mask1

        lda     #%00000000
        sta     mask2

        lda     flash
        beq     perform_flash

        lda     #%00000100
        ldx     cursorx
        beq     finish_mask2
        ror     A
        dex
        beq     finish_mask2
        ror     A
finish_mask2:
        sta     mask2

perform_flash:
        lda     player
        beq     cursor_player0

cursor_player1:
        ldx     cursory
        lda     red_out, X
        and     mask1
        ora     mask2
        sta     red_out, X
        jmp     finish_flash

cursor_player0:
        ldx     cursory
        lda     green_out, X
        and     mask1
        ora     mask2
        sta     green_out, X
        jmp     finish_flash

finish_flash:
        dec     flash_count
        bne     exit_disp
```

```
        lda     #$0A
        sta     flash_count

        dec     flash
        beq     exit_disp

        lda     #$01
        sta     flash

exit_disp:
        rts
;-------------------------------------
user_input:
        ldx     player          ; get controller data
        lda     ctrl1, X
        sta     control

        eor     prev_ctrl
        sta     temp
        bne     user_did_input
        rts

user_did_input:
ch_A:   ror     temp
        bcs     bch_A
ch_B:   ror     temp
        bcs     bch_B
ch_SE:  ror     temp
        bcs     bch_SELECT
ch_ST:  ror     temp
        bcs     bch_START
ch_U:   ror     temp
        bcs     bch_UP
ch_D:   ror     temp
        bcs     bch_DOWN
ch_L:   ror     temp
        bcs     bch_LEFT
ch_R:   ror     temp
        bcs     bch_RIGHT

buttons_done:
        lda     control
        sta     prev_ctrl
        rts
;-------------------------------------
bch_A:
        lda     control
        and     #%00000001
        bne     ch_B
        jsr     A_rel
        jmp     ch_B
bch_B:
        lda     control
        and     #%00000010
        bne     ch_SE
        jsr     B_rel
        jmp     ch_SE
bch_SELECT:
        lda     control
        and     #%00000100
        bne     ch_ST
        jsr     SELECT_rel
        jmp     ch_ST
bch_START:
        lda     control
        and     #%00001000
        bne     ch_U
        jsr     START_rel
        jmp     ch_U
bch_UP:
        lda     control
```

```
        and     #%00010000
        bne     ch_D
        jsr     UP_rel
        jmp     ch_D
bch_DOWN:
        lda     control
        and     #%00100000
        bne     ch_L
        jsr     DOWN_rel
        jmp     ch_L
bch_LEFT:
        lda     control
        and     #%01000000
        bne     ch_R
        jsr     LEFT_rel
        jmp     ch_R
bch_RIGHT:
        lda     control
        and     #%10000000
        bne     buttons_done
        jsr     RIGHT_rel
        jmp     buttons_done
;-------------------------------------
A_rel:
        jsr     get_contents
        tay
        bne     try_steal

        ldx     player
        inx
        txa
        jsr     store_contents

        ldx     player
        lda     cursorx
        sta     lastx, X
        lda     cursory
        sta     lasty, X

        lda     #$01
        sta     sw_player_flag

        rts

try_steal:
        lda     game_mode
        beq     ex_A_rel

        ldx     player          ; check remaining steals
        lda     steals, X
        beq     ex_A_rel

        lda     player
        jsr     other_player
        tax
        lda     lastx, X
        sec
        sbc     cursorx
        bne     do_steal

        lda     player
        jsr     other_player
        tax
        lda     lasty, X
        sec
        sbc     cursory
        beq     ex_A_rel

do_steal:
        ldx     player
        lda     cursorx
```

```
        sta     lastx, X
        lda     cursory
        sta     lasty, X

        ldx     player
        dec     steals, X

        inx
        txa
        jsr     store_contents

        lda     #$01
        sta     sw_player_flag

ex_A_rel:
        rts
;-------------------------------------
B_rel:
        rts
;-------------------------------------
SELECT_rel:
        rts
;-------------------------------------
START_rel:
        rts
;-------------------------------------
UP_rel:
        lda     cursory
        beq     ex_UP_rel
        dec     cursory
ex_UP_rel:
        rts
;-------------------------------------
DOWN_rel:
        lda     cursory
        sec
        sbc     #$02
        beq     ex_DOWN_rel
        inc     cursory
ex_DOWN_rel:
        rts
;-------------------------------------
LEFT_rel:
        lda     cursorx
        beq     ex_LEFT_rel
        dec     cursorx
ex_LEFT_rel:
        rts
;-------------------------------------
RIGHT_rel:
        lda     cursorx
        sec
        sbc     #$02
        beq     ex_RIGHT_rel
        inc     cursorx
ex_RIGHT_rel:
        rts
;-------------------------------------
switch_player:
        dec     player
        beq     done_switch_player

        lda     #$01
        sta     player

done_switch_player:
        lda     #$00
        sta     prev_ctrl

        rts
;-------------------------------------
```

```
other_player:
        tay
        dey
        beq     ex_other_player

        ldy     #$01
        tya

ex_other_player:
        tya
        rts
;-------------------------------------
game_end:
        lda     #$FF
        sec
        sbc     winner
        beq     game_ended_cat

        lda     winner
        beq     error_winner

        dec     winner
        beq     player0_won
        jmp     player1_won

game_ended_cat:
        lda     #$FF
        sta     red_out
        sta     red_out+2
        sta     green_out
        sta     green_out+2
        lda     #$04
        sta     red_out+1
        sta     green_out+1
        lda     player
        sta     player_start
        jmp     wait_reset

player0_won:
        lda     #$05
        sta     green_out
        sta     green_out+2
        lda     #$02
        sta     green_out+1
        lda     #$00
        sta     red_out
        sta     red_out+1
        sta     red_out+2
        lda     #$01
        sta     player_start
        jmp     wait_reset

player1_won:
        lda     #$05
        sta     red_out
        sta     red_out+2
        lda     #$02
        sta     red_out+1
        lda     #$00
        sta     green_out
        sta     green_out+1
        sta     green_out+2
        lda     #$00
        sta     player_start
        jmp     wait_reset

error_winner:
        lda     #$FF
        sta     red_out
        sta     red_out+1
        sta     red_out+2
```

```
        sta     green_out
        sta     green_out+1
        sta     green_out+2
        jmp     freeze
;-------------------------------------
wait_reset:
        lda     ctrl1
        and     #$0C
        sec
        sbc     #$0C
        bne     wait_reset

wait_reset_release:
        lda     ctrl1
        bne     wait_reset_release

        jmp     setup
;-------------------------------------
freeze: jmp     freeze
;-------------------------------------
check_for_end:
check_row_one:
        lda     gameboard+0
        sec
        sbc     gameboard+1
        bne     check_row_two

        lda     gameboard+1
        sec
        sbc     gameboard+2
        bne     check_row_two

        lda     gameboard+0
        beq     check_row_two

        lda     gameboard+0
        sta     winner
        rts

check_row_two:
        lda     gameboard+3
        sec
        sbc     gameboard+4
        bne     check_row_three

        lda     gameboard+4
        sec
        sbc     gameboard+5
        bne     check_row_three

        lda     gameboard+3
        beq     check_row_three

        lda     gameboard+3
        sta     winner
        rts

check_row_three:
        lda     gameboard+6
        sec
        sbc     gameboard+7
        bne     check_col_one

        lda     gameboard+7
        sec
        sbc     gameboard+8
        bne     check_col_one

        lda     gameboard+6
        beq     check_col_one
```

```
        lda     gameboard+6
        sta     winner
        rts

check_col_one:
        lda     gameboard+0
        sec
        sbc     gameboard+3
        bne     check_col_two

        lda     gameboard+3
        sec
        sbc     gameboard+6
        bne     check_col_two

        lda     gameboard+0
        beq     check_col_two

        lda     gameboard+0
        sta     winner
        rts

check_col_two:
        lda     gameboard+1
        sec
        sbc     gameboard+4
        bne     check_col_three

        lda     gameboard+4
        sec
        sbc     gameboard+7
        bne     check_col_three

        lda     gameboard+1
        beq     check_col_three

        lda     gameboard+1
        sta     winner
        rts

check_col_three:
        lda     gameboard+2
        sec
        sbc     gameboard+5
        bne     check_diag_one

        lda     gameboard+5
        sec
        sbc     gameboard+8
        bne     check_diag_one

        lda     gameboard+2
        beq     check_diag_one

        lda     gameboard+2
        sta     winner
        rts

check_diag_one:
        lda     gameboard+0
        sec
        sbc     gameboard+4
        bne     check_diag_two

        lda     gameboard+4
        sec
        sbc     gameboard+8
        bne     check_diag_two

        lda     gameboard+0
        beq     check_diag_two
```

```
        lda     gameboard+0
        sta     winner
        rts

check_diag_two:
        lda     gameboard+2
        sec
        sbc     gameboard+4
        bne     check_cat

        lda     gameboard+4
        sec
        sbc     gameboard+6
        bne     check_cat

        lda     gameboard+2
        beq     check_cat

        lda     gameboard+2
        sta     winner
        rts

check_cat:
        lda     gameboard+0
        beq     game_not_over
        lda     gameboard+1
        beq     game_not_over
        lda     gameboard+2
        beq     game_not_over
        lda     gameboard+3
        beq     game_not_over
        lda     gameboard+4
        beq     game_not_over
        lda     gameboard+5
        beq     game_not_over
        lda     gameboard+6
        beq     game_not_over
        lda     gameboard+7
        beq     game_not_over
        lda     gameboard+8
        beq     game_not_over

        lda     #$FF
        sta     winner
        rts

game_not_over:
        lda     #$00
        rts
;-------------------------------------
get_contents:
        lda     #$00
        clc
        ldx     cursory
        beq     get_x_cont
        adc     #$03
        dex
        beq     get_x_cont
        adc     #$03

get_x_cont:
        clc
        adc     cursorx
        tax
        lda     gameboard, X
        rts
;-------------------------------------
store_contents:
        tay
```

```
        lda     #$00
        clc
        ldx     cursory
        beq     get_x_cont2
        adc     #$03
        dex
        beq     get_x_cont2
        adc     #$03

get_x_cont2:
        adc     cursorx
        tax
        tya
        sta     gameboard, X
        rts
;-------------------------------------
get_ctrl1:
        lda     #$08
        tay
        ldx     #$01
        stx     $4016           ; latch data
        dex
        stx     $4016

get_bit:
        lda     $4016           ; rotate into carry
        ror     A

        txa                     ; rotate into MSB of x
        ror     A
        tax

        dey             ; next
        bne     get_bit

        rts
;-------------------------------------
get_ctrl2:
        lda     #$08
        tay
        ldx     #$01
        stx     $4016           ; latch data
        dex
        stx     $4016

get_bit2:
        lda     $4017
        ror     A
        txa
        ror     A
        tax
        dey
        bne     get_bit2

        rts
;-------------------------------------


;-------------------------------------
irq:
        php                     ; save context
        pha

        lda     #$55            ; reset timer to 60hz
        sta     timer

        lda     #$01            ; set flag
        sta     int_flag

        ldx     frame           ; select row
        lda     #$ff
```

```
        clc
sh_l_l: rol     A
        dex
        bne     sh_l_l
        sta     row_out


        ldx     frame           ; output column data for red and green
        dex
        lda     red_out, X
        asl     A
        asl     A
        asl     A
        sta     temp
        ldx     frame
        dex
        lda     green_out, X
        ora     temp
        and     #$3F
        sta     data_out


        dec     frame           ; only continue if in frame one
        bne     exit_irq

        lda     #$03            ; reload frame
        sta     frame

        jsr     get_ctrl1       ;get inputs
        sta     ctrl1
        jsr     get_ctrl2
        sta     ctrl2

exit_irq:
        pla
        plp
        rti
;-------------------------------------
nmi:    rti
;-------------------------------------


menu_red_row_1:
        db      %00000100
        db      %00000010
        db      %00000001
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000000
menu_red_row_2:
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000001
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000100
menu_red_row_3:
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000001
        db      %00000010
        db      %00000100
        db      %00000000
menu_grn_row_1:
        db      %00000000
```

```
        db      %00000000
        db      %00000100
        db      %00000010
        db      %00000001
        db      %00000000
        db      %00000000
        db      %00000000
menu_grn_row_2:
        db      %00000000
        db      %00000100
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000001
        db      %00000000
        db      %00000000
menu_grn_row_3:
        db      %00000100
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000000
        db      %00000001
        db      %00000010

        org     $FFFA
        dw      nmi
        dw      reset_v
        dw      irq
```