FPGA Implementation of the Nintendo Entertainment System (NES)

*Team Name: Four People Generating A Nintendo Entertainment System*

*Eric Sullivan, Pavan Holla, Jonathan Ebert, Patrick Yang*

Micro-Architecture Document

University of Wisconsin-Madison

Spring 2017

**Table of Contents**

[**Introduction**](#_qm44511vwcs2) **3**

[**Top Level Block Diagram**](#_gjdgxs) **4**

[**CPU**](#_rdm0fxe7s5y9) **4**

[**Memory**](#_366k07gr393w) **4**

[**CPU Registers**](#_eftj2rjo9s1h) **5**

[**CPU ISA**](#_wvvuysbh6ili) **5**

[**CPU Addressing Modes**](#_9ie72e7qdon7) **5**

[**CPU Interrupts**](#_b7loafaij831) **6**

[**Opcode Matrix**](#_q77927r8c49e) **6**

[**CPU Block Diagram**](#_ki2ushz1f0a3) **8**

[**Top Level Interface**](#_er7yq56tef4v) **10**

[**Instruction Decode Interface**](#_lomim8yvajg6) **10**

[**MEM module**](#_b3lc7rd7ackr) **11**

[**ALU**](#_nexle64nrxxq) **11**

[**ALU Input Selector**](#_7m32oocsgxzi) **12**

[**Registers module**](#_5httww8vnv8g) **12**

[**Processor Control**](#_eiifcf5tv1l9) **13**

[**Enums**](#_3on9q42bqgrx) **14**

[**PPU**](#_cb54tv1epevk) **15**

[**PPU Memory Map**](#_dkksbep8besq) **16**

[**PPU Registers**](#_djradund7fk8) **16**

[**PPU CHAROM**](#_pa3j41hpauwc) **18**

[**PPU Rendering**](#_pg3as4rfwz9v) **18**

[**PPU Block Diagram**](#_qeyy04rw7qbm) **20**

[**PPU Memory Mapped Register Interface**](#_8bwwxukxxmnt) **20**

[**PPU Register Block Diagram**](#_h50m30h4jbfm) **21**

[**PPU Background Renderer**](#_t866hv20kwva) **22**

[**PPU Sprite Renderer**](#_o1sa0cowh2br) **22**

[**VRAM Interface**](#_xlgt5p96mvki) **23**

[**DMA**](#_7oqgkdlshdds) **24**

[**Memory Mappers**](#_7c1koopsou4c) **25**

[**APU**](#_8dlgb2qio3dw) **25**

[**APU Registers**](#_qk56hr19gphj) **26**

[**Controller (SPART)**](#_h6yc9vh2wntf) **27**

[**VGA**](#_xrfacxsmruiq) **28**

[**VGA Clock Gen**](#_w1ger0jiy8bk) **30**

[**VGA Timing Gen**](#_pw3svolg7ia) **30**

[**VGA Display Plane**](#_em4ktvbwfn8k) **30**

[**VGA RAM**](#_68btkfb1o5ru) **31**

[**VGA RAM Reader**](#_1cvttmqqni8n) **32**

[**Software**](#_zbj1aj20rigt) **33**

[**Controller Simulator**](#_ik7f7bcto722) **33**

[**PPU ROM’s Memory Map**](#_o4bj1b7uzlbc) **34**

[**CPU ROM’s Memory Map**](#_n5cypllda98c) **35**

[**Assembler**](#_t5xpa0jvlbyf) **35**

[**Opcode Table**](#_rp3i11uarg1) **36**

[**iNES ROM Converter**](#_ekqczej5ygn6) **38**

# Introduction

Following the video game crash in the early 1980s, Nintendo released their first video game console, the Nintendo Entertainment System (NES). Following a slow release and early recalls, the console began to gain momentum in a market that many thought had died out, and the NES is still appreciated by enthusiasts today. A majority of its early success was due to the relationship that Nintendo created with third-party software developers. Nintendo required that restricted developers from publishing games without a license distributed by Nintendo. This decision led to higher quality games and helped to sway the public opinion on video games, which had been plagued by poor games for other gaming consoles.

Our motivation is to better understand how the NES worked from a hardware perspective, as the NES was an extremely advanced console when it was released in 1985 (USA). The NES has been recreated multiple times in software emulators, but has rarely been done in a hardware design language, which makes this a unique project. Nintendo chose to use the 6502 processor, also used by Apple in the Apple II, and chose to include a picture processing unit to provide a memory efficient way to output video to the TV. Our main goal is to recreate the CPU and PPU in hardware, so that we can run games that were run on the original console. In order to exactly recreate the original console, we will also need to include memory mappers, an audio processing unit, a DMA unit, a VGA interface, and a way to use a controller for input. In addition, we will be writing our own assembler for the 6502 that will allow us to create simple programs to test our implementation.

Due to the complexity of the project, work will start on the CPU and PPU. A few weeks in we will consider how much progress has been made to the PPU, and if we don’t think that we will be able to finish it in a timely manner, we will instead use another implementation that we found online. By spending time creating our own PPU, we will gain valuable insight that will allow us to better integrate the whole console together. From here, we will begin working on other blocks of the project, including the controller, the memory mapper, and the APU (if time permits). Ultimately, the goal is to get an NES game running on our FPGA.

# Top Level Block Diagram



Figure 1: System level diagram.

# CPU

## Memory

The NES 6502 CPU has a 16 bit address bus and an 8 bit data bus. The memory latency is one cycle, i.e data follows

## 

## CPU Registers

The CPU of the NES is the MOS 6502. It is an accumulator plus index register machine. There are five primary registers on which operations are performed:

1. **PC**
2. **Accumulator(A)**
3. **X**
4. **Y**
5. **Stack pointer**
6. **Status Register**

## CPU ISA

The ISA may be classified into a few broad operations:

* Load into A,X,Y registers from memory
* Perform arithmetic operation on A,X or Y
* Move data from one register to another
* Program control instructions like Jump and Branch
* Stack operations
* Complex instructions that read, modify and write back memory.

## CPU Addressing Modes

Additionally, there are thirteen addressing modes which these operations can use. They are

* **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 content of sum of (the program counter and the byte in memory immediately following the instruction) is used.
* **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.
* **(Indirect, X)** – A combination of Indirect Addressing and Indexed Addressing
* **(Indirect), Y** - A combination of Indirect Addressing and Indexed Addressing

## CPU Interrupts

The 6502 supports three interrupts. The reset interrupt routine is called after a physical reset. The other two interrupts are the non\_maskable\_interrupt(NMI) and the general\_interrupt(IRQ). The general\_interrupt can be disabled by software whereas the others cannot.

## Opcode Matrix

The NES 6502 ISA is a CISC like ISA with 56 instructions. These 56 instructions can pair up with addressing modes to form various opcodes. The opcode is always 8 bits, however based on the addressing mode, upto 4 more memory location may need to be fetched.The memory is single cycle, i.e data[7:0] can be latched the cycle after address[15:0] is placed on the bus. The following tables summarize the instructions available and possible addressing modes:

|  |  |
| --- | --- |
| **Storage** | |
| LDA | Load A with M |
| LDX | Load X with M |
| LDY | Load Y with M |
| STA | Store A in M |
| STX | Store X in M |
| STY | Store Y in M |
| TAX | Transfer A to X |
| TAY | Transfer A to Y |
| TSX | Transfer Stack Pointer to X |
| TXA | Transfer X to A |
| TXS | Transfer X to Stack Pointer |
| TYA | Transfer Y to A |
| **Arithmetic** | |
| ADC | Add M to A with Carry |
| DEC | Decrement M by One |
| DEX | Decrement X by One |
| DEY | Decrement Y by One |
| INC | Increment M by One |
| INX | Increment X by One |
| INY | Increment Y by One |
| SBC | Subtract M from A with Borrow |
| **Bitwise** | |
| AND | AND M with A |
| ASL | Shift Left One Bit (M or A) |
| BIT | Test Bits in M with A |
| EOR | Exclusive-Or M with A |
| LSR | Shift Right One Bit (M or A) |
| ORA | OR M with A |
| ROL | Rotate One Bit Left (M or A) |
| ROR | Rotate One Bit Right (M or A) |
| **Branch** | |
| BCC | Branch on Carry Clear |
| BCS | Branch on Carry Set |
| BEQ | Branch on Result Zero |
| BMI | Branch on Result Minus |
| BNE | Branch on Result not Zero |
| BPL | Branch on Result Plus |
| BVC | Branch on Overflow Clear |
| BVS | Branch on Overflow Set |
| **Jump** | |
| JMP | Jump to Location |
| JSR | Jump to Location Save Return Address |
| RTI | Return from Interrupt |
| RTS | Return from Subroutine |
| **Status Flags** | |
| CLC | Clear Carry Flag |
| CLD | Clear Decimal Mode |
| CLI | Clear interrupt Disable Bit |
| CLV | Clear Overflow Flag |
| CMP | Compare M and A |
| CPX | Compare M and X |
| CPY | Compare M and Y |
| SEC | Set Carry Flag |
| SED | Set Decimal Mode |
| SEI | Set Interrupt Disable Status |
| **Stack** | |
| PHA | Push A on Stack |
| PHP | Push Processor Status on Stack |
| PLA | Pull A from Stack |
| PLP | Pull Processor Status from Stack |
| **System** | |
| BRK | Force Break |
| NOP | No Operation |

The specific opcode hex values are specified in the Assembler section [here](#_rp3i11uarg1).

For more information on the opcodes, please refer

<http://www.6502.org/tutorials/6502opcodes.html>

or

<http://www.thealmightyguru.com/Games/Hacking/Wiki/index.php/6502_Opcodes>

## CPU Block Diagram

|  |  |
| --- | --- |
| **Block** | **Primary Function** |
| Decode | Decode the current instruction. Classifies the opcode into an instruction\_type(arithmetic,ld etc) and addressing mode(immediate, indirect etc) |
| Processor Control | State machine that keeps track of current instruction stage, and generates signals to load registers. |
| ALU | Performs ALU ops and handles Status Flags |
| Registers | Contains all registers. Register values change according to signals from processor control. |

**Instruction flow**

The following table presents a high level overview of how each instruction is handled.

|  |  |  |
| --- | --- | --- |
| **Cycle Number** | **Blocks** | **Action** |
| 0 | Processor Control → Registers | Instruction Fetch |
| 1 | Register → Decode | Classify instruction and addressing mode |
| 1 | Decode → Processor Control | Init state machine for instruction type and addressing mode |
| 2-6 | Processor Control → Registers | Populate scratch registers based on addressing mode. |
| Last Cycle | Processor Control → ALU | Execute |
| Last Cycle | Processor Control → Registers | Instruction Fetch |

**State Machines**

Each {instruction\_type, addressing\_mode} triggers its own state machine. In brief, this state machine is responsible for signalling the Registers module to load/store addresses from memory or from the ALU.

State machine spec for each instruction type and addressing mode can be found at <https://docs.google.com/spreadsheets/d/16uGTSJEzrANUzr7dMmRNFAwA-_sEox-QsTjJSlt06lE/edit?usp=sharing>

Considering one of the simplest instructions ADC immediate,which takes two cycles, the state machine is as follows:

Instruction\_type=ARITHMETIC, addressing mode= IMMEDIATE

|  |  |  |
| --- | --- | --- |
| state=0 | state=1 | state=2 |
| ld\_sel=LD\_INSTR; //instr= memory\_data  pc\_sel=INC\_PC; //pc++  next\_state=state+1’b1 | ld\_sel=LD\_IMM;  //imm=memory\_data  pc\_sel=INC\_PC  next\_state=state+1’b1 | alu\_ctrl=DO\_OP\_ADC // execute  src1\_sel=SRC1\_A  src2\_sel=SRC2\_IMM  dest\_sel=DEST\_A  ld\_sel=LD\_INSTR//fetch next instruction  pc\_sel=INC\_PC  next\_state=1’b1 |

All instructions are classified into one of 55 state machines in the cpu specification sheet. The 6502 can take variable time for a single instructions based on certain conditions(page\_cross, branch\_taken etc). These corner case state transitions are also taken care of by processor control.

## Top Level Interface

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst | input |  | System active high reset |
| nmi | input | PPU | Non maskable interrupt from PPU. Executes BRK instruction in CPU |
| addr[15:0] | output | RAM | Address for R/W issued by CPU |
| dout[7:0] | input/  output | RAM | Data from the RAM in case of reads and and to the RAM in case of writes |
| memory\_read | output | RAM | read enable signal for RAM |
| memory\_write | output | RAM | write enable signal for RAM |

## Instruction Decode Interface

The decode module is responsible for classifying the instruction into one of the addressing modes and an instruction type. It also generates the signal that the ALU would eventually use if the instruction passed through the ALU.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| instruction\_register | input | Registers | Opcode of the current instruction |
| nmi | input | cpu\_top | Non maskable interrupt from PPU. Executes BRK instruction in CPU |
| instruction\_type | output | Processor Control | Type of instruction. Belongs to enum ITYPE. |
| addressing\_mode | output | Processor Control | Addressing mode of the opcode in instruction\_register. Belongs to enum AMODE. |
| alu\_sel | output | ALU | ALU operation expected to be performed by the opcode, eventually. Processor control chooses to use it at a cycle appropriate for the instruction. Belongs to enum DO\_OP. |

## MEM module

The MEM module is the interface between memory and CPU. It provides appropriate address and read/write signal for the memory. Controlled by the select signals

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| addr\_sel | input | Processor Control | Selects which input to use as address to memory. Enum of ADDR |
| int\_sel | input | Processor Control | Selects which interrupt address to jump to. Enum of INT\_TYPE |
| ld\_sel,st\_sel | input | Processor Control | Decides whether to read or write based on these signals |
| ad, ba, sp, irql, irqh, pc | input | Registers | Registers that are candidates of the address |
| addr | output | Memory | Address of the memory to read/write |
| read,write | output | Memory | Selects whether Memory should read or write |

## ALU

Performs arithmetic, logical operations and operations that involve status registers.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| in1, in2 | input | ALU Input Selector | Inputs to the ALU operations selected by ALU Input module. |
| alu\_sel | input | Processor Control | ALU operation expected to be performed by the opcode, eventually. Processor control chooses to use it at a cycle appropriate for the instruction. Belongs to enum DO\_OP. |
| clk, rst | input |  | System clock and active high reset |
| out | output | to all registers | Output of ALU operation. sent to all registers and registers decide whether to receive it or ignore it as its next value. |
| n, z, v, c, b, d, i | output |  | Status Register |

### ALU Input Selector

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| src1\_sel, src2\_sel | input | Processor Control | Control signal that determines which sources to take in as inputs to ALU according to the instruction and addressing mode |
| a, bal, bah, adl, pcl, pch, imm, adv, x, bav, y, offset | input | Registers | Registers that are candidates to the input to ALU |
| temp\_status | input | ALU | Sometimes status information is required but we don’t want it to affect the status register. So we directly receive temp\_status value from ALU |
| in1, in2 | output | ALU | Selected input for the ALU |

## Registers module

The following registers are present in the module

A - Accumulator

X,Y - Register to support indexed addressing

SP - Stack Pointer

PC - Program Counter

Status - 8 bit register where status[0:7] = { Carry flag, Zero flag, IRQ disable, 0, Break executed flag, 1, Overflow flag, Sign flag } .

ADH, ADL, BAH, BAL, ADV, BAV - Temporary registers for storing addresses. Used by the processor control as scratch registers.

IMM, Offset - registers that are used for calculations

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk, rst | input |  | System clk and rst |
| dest\_sel, pc\_sel, sp\_sel, ld\_sel, st\_sel | input | Processor Control | Selects which input to accept as new input. enum of DEST, PC, SP, LD, ST |
| clr\_adh, clr\_bah | input | Processor Control | Clears the high byte of ad, ba |
| alu\_out, next\_status | input | ALU | Output from ALU and next status value. alu\_out can be written to most of the registers |
| data | inout | Memory | Datapath to Memory. Either receives or sends data according to ld\_sel and st\_sel. |
| a, x, y, ir, imm, adv, bav, offset, sp, pc, ad, ba, n, z, v, c, b, d, i, status | output |  | Register outputs that can be used by different modules |

## Processor Control

The processor control module maintains the current state that the instruction is in and decides the control signals for the next state. Once the instruction type and addressing modes are decoded, the processor control block becomes aware of the number of cycles the instruction will take. Thereafter, at each clock cycle it generates the required control signals.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| instruction\_type | input | Decode | Type of instruction. Belongs to enum ITYPE. |
| addressing\_mode | input | Decode | Addressing mode of the opcode in instruction\_register. Belongs to enum AMODE. |
| alu\_ctrl | input | Decode | ALU operation expected to be performed by the opcode, eventually. Processor control chooses to use it at a cycle appropriate for the instruction. Belongs to enum DO\_OP. |
| reset\_adh | output | Registers | Resets ADH register |
| reset\_bah | output | Registers | Resets BAH register |
| set\_b | output | Registers | Sets the B flag |
| addr\_sel | output | Registers | Selects the value that needs to be set on the address bus. Belongs to enum ADDR |
| alu\_sel | output | ALU | Selects the operation to be performed by the ALU in the current cycle. Belongs to enum DO\_OP |
| dest\_sel | output | Registers | Selects the register that receives the value from ALU output.Belongs to enum DEST |
| ld\_sel | output | Registers | Selects the register that will receive the value from Memory Bus. Belongs to enum LD |
| pc\_sel | output | Registers | Selects the value that the PC will take next cycle. Belongs to enum PC |
| sp\_sel | output | Registers | Selects the value that the SP will take next cycle. Belongs to enum SP |
| src1\_sel | output | ALU | Selects src1 for ALU. Belongs to enum SRC1 |
| src2\_sel | output | ALU | Selects src2 for ALU. Belongs to enum SRC2 |
| st\_sel | output | Registers | Selects the register whose value will be placed on dout. Belongs to enum ST |

## Enums

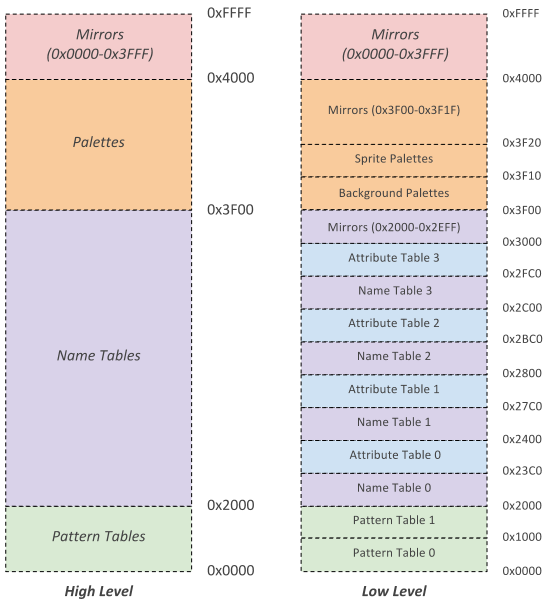
|  |  |
| --- | --- |
| Enum name | Legal Values |
| ITYPE | ARITHMETIC,BRANCH,BREAK,CMPLDX,CMPLDY,INTERRUPT,JSR,JUMP,OTHER,PULL,PUSH,RMW,RTI,RTS,STA,STX,STY |
| AMODE | ABSOLUTE,ABSOLUTE\_INDEX,ABSOLUTE\_INDEX\_Y,ACCUMULATOR,IMMEDIATE,IMPLIED,INDIRECT,INDIRECT\_X,INDIRECT\_Y,RELATIVE,SPECIAL,ZEROPAGE,ZEROPAGE\_INDEX,ZEROPAGE\_INDEX\_Y |
| DO\_OP | DO\_OP\_ADD,DO\_OP\_SUB,DO\_OP\_AND,DO\_OP\_OR,DO\_OP\_XOR,DO\_OP\_ASL,DO\_OP\_LSR,DO\_OP\_ROL,DO\_OP\_ROR,DO\_OP\_SRC2DO\_OP\_CLR\_C,DO\_OP\_CLR\_I,DO\_OP\_CLR\_V,DO\_OP\_SET\_C,DO\_OP\_SET\_I,DO\_OP\_SET\_V |
| ADDR | ADDR\_AD,ADDR\_PC,ADDR\_BA,ADDR\_SP,ADDR\_IRQL,ADDR\_IRQH |
| LD | LD\_INSTR,LD\_ADL,LD\_ADH,LD\_BAL,LD\_BAH,LD\_IMM,LD\_OFFSET,LD\_ADV,LD\_BAV,LD\_PCL,LD\_PCH |
| SRC1 | SRC1\_A,SRC1\_BAL,SRC1\_BAH,SRC1\_ADL,SRC1\_PCL,SRC1\_PCH,SRC1\_BAV,SRC1\_1 |
| SRC2 | SRC2\_DC,SRC2\_IMM,SRC2\_ADV,SRC2\_X,SRC2\_BAV,SRC2\_C,SRC2\_1,SRC2\_Y,SRC2\_OFFSET |
| DEST | DEST\_BAL,DEST\_BAH,DEST\_ADL,DEST\_A,DEST\_X,DEST\_Y,DEST\_PCL,DEST\_PCH,DEST\_NONE |
| PC | AD\_P\_TO\_PC,INC\_PC,KEEP\_PC |
| SP | INC\_SP,DEC\_SP |

## 

# PPU

The PPU is responsible to all of the drawing logic for video output. It contains data about both background tiles and sprite data. To draw this data to the screen the background and sprite data are fetched for every pixel and then a mux decides what data to send to the video out. Overall the logic of how the drawing is done is not too complex, but the protocols for obtaining this information from RAM/ROM is difficult because the PPU has to work between the two different clock domains of the CPU and memory system.

### PPU Memory Map



### PPU Registers

* + Control registers are mapped into the CPUs address space ($2000 - $2007)
  + The registers are repeated every eight bytes until address $3FFF
  + **PPUCTRL[7:0] (**$2000) WRITE
    - [1:0]: Base nametable address which is loaded at the start of a frame
      * 0: $2000
      * 1: $2400
      * 2: $2800
      * 3: $2C00
    - [2]: VRAM address increment per CPU read/write of PPUDATA
      * 0: Add 1 going across
      * 1: Add 32 going down
    - [3]: Sprite pattern table for 8x8 sprites
      * 0: $0000
      * 1: $1000
      * Ignored in 8x16 sprite mode
    - [4]: Background pattern table address
      * 0: $0000
      * 1: $1000
    - [5]: Sprite size
      * 0: 8x8
      * 1: 8x16
    - [6]: PPU master/slave select
      * 0: Read backdrop from EXT pins
      * 1: Output color on EXT pins
    - [7]: Generate NMI interrupt at the start of vertical blanking interval
      * 0: off
      * 1: on
  + **PPUMASK[7:0]** ($2001) WRITE
    - [0]: Use grayscale image
      * 0: Normal color
      * 1: Grayscale
    - [1]: Show left 8 pixels of background
      * 0: Hide
      * 1: Show background in leftmost 8 pixels of screen
    - [2]: Show left 8 piexels of sprites
      * 0: Hide
      * 1: Show sprites in leftmost 8 pixels of screen
    - [3]: Render the background
      * 0: Don’t show background
      * 1: Show background
    - [4]: Render the sprites
      * 0: Don’t show sprites
      * 1: Show sprites
    - [5]: Emphisize red
    - [6]: Emphisize green
    - [7]: Emphisize blue
  + **PPUSTATUS[7:0]** ($2002) READ
    - [4:0]: Nothing?
    - [5]: Set for sprite overflow which is when more than 8 sprites exist in one scanline (Is actually more complicated than this to do a hardware bug)
    - [6]: Sprite 0 hit. This bit gets set when a non zero part of sprite zero overlaps a non zero background pixel
    - [7]: Vertical blank status register
      * 0: Not in vertical blank
      * 1: Currently in vertical blank
  + **OAMADDR[7:0]** ($2003) WRITE
    - Address of the object attribute memory the program wants to access
  + **OAMDATA[7:0]** ($2004) READ/WRITE
    - ???
  + **PPUSCROLL[7:0]** ($2005) WRITE
    - Tells the PPU what pixel of the nametable selected in PPUCTRL should be in the top left hand corner of the screen
  + **PPUADDR[7:0]** ($2006) WRITE
    - Address the CPU wants to write to VRAM before writing a read of PPUSTATUS is required and then two bytes are written in first the high byte then the low byte
  + **PPUDATA[7:0]** ($2007) READ/WRITE
    - Writes/Reads data from VRAM for the CPU. The value in PPUADDR is then incremented by the value specified in PPUCTRL
  + **OAMDMA[7:0]** ($4014) WRITE
    - A write of $XX to this register will result in the CPU memory page at $XX00-$XXFF being written into the PPU object attribute memory

### PPU CHAROM

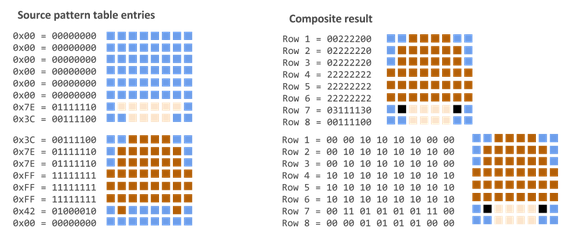
* + ROM from the cartridge is broken in two sections
    - Program ROM
      * Contains program code for the 6502
      * Is mapped into the CPU address space by the mapper
    - Character ROM
      * Contains sprite and background data for the PPU
      * Is mapped into the PPU address space by the mapper

### PPU Rendering

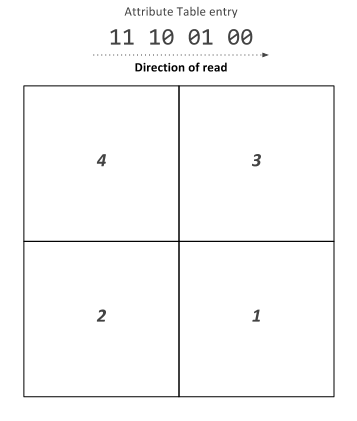
* + Pattern Tables
    - $0000-$2000 in VRAM
      * Pattern Table 0 ($0000-$0FFF)
      * Pattern Table 1 ($1000-$2000)
      * The program selects which one of these contains sprites and backgrounds
      * Each pattern table is 16 bytes long and represents 1 8x8 pixel tile
        + Each 8x1 row is 2 bytes long
        + Each bit in the byte represents a pixel and the corresponding bit for each byte is combined to create a 2 bit color.

Color\_pixel = {byte2[0], byte1[0]}

* + - * + So there can only be 4 colors in any given tile
        + Rightmost bit is leftmost pixel
    - Any pattern that has a value of 0 is transparent i.e. the background color



* + Name Tables
    - $2000-$2FFF in VRAM with $3000-$3EFF as a mirror
    - Laid out in memory in 32x30 fashion
      * Resulting in a resolution of 256x240
    - Although the PPU supports 4 name tables the NES only supplied enough VRAM for 2 this results in 2 of the 4 name tables being mirror
      * Vertically = horizontal movement
      * Horizontally = vertical movement
    - Each entry in the name table refers to one pattern table and is one byte. Since there are 32x30=960 entries each name table requires 960 bytes of space the left over 64 bytes are used for attribute tables
    - Attribute tables
      * 1 byte entries that contains the palette assignment for a 2x2 grid of tiles



* + Sprites
    - Just like backgrounds sprite tile data is contained in one of the pattern tables
    - But unlike backgrounds sprite information is not contained in name tables but in a special reserved 256 byte RAM called the object attribute memory (OAM)
  + Object Attribute Memory
    - 256 bytes of dedicated RAM
    - Each object is allocated 4 bytes of OAM so we can store data about 64 sprites at once
    - Each object has the following information stored in OAM
      * X Coordinate
      * Y Coordinate
      * Pattern Table Index
      * Palette Assignment
      * Horizontal/Vertical Flip
  + Palette Table
    - Located at $3F00-$3F20
      * $3F00-$3F0F is background palettes
      * $3F10-$3F1F is sprite palettes
    - Mirrored all the way to $4000
    - Each color takes one byte
    - Every background tile and sprite needs a color palette.
    - When the background or sprite is being rendered the the color for a specific table is looked up in the correct palette and sent to the draw select mux.
  + Rendering is broken into two parts which are done for each horizontal scanline
    - Background Rendering
      * The background enable register ($2001) controls if the default background color is rendered ($2001) or if background data from the background renderer.
      * The background data is obtained for every pixel
    - Sprite Rendering
      * The sprite renderer has room for 8 unique sprites on each scanline.
      * For each scanline the renderer looks through the OAM for sprites that need to be drawn on the scanline. If this is the case the sprite is loaded into the scanline local sprites
        + If this number exceeds 8 a flag is set and the behavior is undefined.
      * If a sprite should be drawn for a pixel instead of the background the sprite renderer sets the sprite priority line to a mux that decides what to send to the screen and the mux selects the sprite color data.

## 

## PPU Block Diagram

Untitled Diagram.png

## PPU Memory Mapped Register Interface

The PPU register interface is how the CPU changes how the PPU renders a 2d scene. It also allows the CPU to completely disable rendering for more time to write to VRAM.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| data[7:0] | inout | CPU | Bi directional data bus between the CPU/PPU |
| address[2:0] | input | CPU | Register select |
| rw | input | CPU | CPU read/write select |
| cs\_in | input | CPU | PPU chip select |
| irq | output | CPU | Signal PPU asserts to trigger CPU NMI |
| back\_pixel\_data[7:0] | output | VGA | Background pixel data to be sent to the display |
| sprte\_pixel\_data[7:0] | output | VGA | Sprite pixel data to be sent to the display |
| back\_pixel\_en | output | VGA | Tells you to write background pixel or default background pixel |
| sprite\_pixel\_en | output | VGA | Tells you to prioritize the sprite pixel over the background pixels |

## PPU Register Block Diagram

Untitled Diagram.png

## PPU Background Renderer

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| bg\_render\_en | input | PPU Register | Background render enable |
| x\_pos[9:0] | input | PPU Register | The current pixel for the active scanline |
| y\_pos[9:0] | input | PPU Register | The current scanline being rendered |
| vram\_data\_in[7:0] | input | PPU Register | The current data that has been read in from VRAM |
| bg\_pt\_sel | input | PPU Register | Selects the location of the background renderer pattern table |
| show\_bg\_left\_col | input | PPU Register | Determines if the background for the leftmost 8 pixels of each scanline will be drawn |
| fine\_x\_scroll[2:0] | input | PPU Register | Selects the pixel drawn on the left hand side of the screen |
| fine\_y\_scroll[2:0] | input | PPU Register | Selects the pixel drawn on the top of the screen |
| bg\_pal\_sel[3:0] | output | Pixel Mux | Selects the palette for the background pixel |
| vram\_addr\_out[13:0] | output | VRAM | The VRAM address the sprite renderer wants to read from |

## PPU Sprite Renderer

The PPU sprite renderer is used to render all of the sprite data for each scanline. The way the hardware was designed it only allows for 64 sprites to kept in object attribute memory at once, and only 8 sprites can be drawn for each scanline.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| spr\_render\_en | input | PPU Register | Sprite renderer enable signal |
| x\_pos[9:0] | input | PPU Register | The current pixel for the active scanline |
| y\_pos[9:0] | input | PPU Register | The current scanline being rendered |
| oam\_addr\_in[7:0] | input | PPU Register | The current OAM address being read/written |
| oam\_data\_in[7:0] | inout | PPU Register | The current data being read/written from OAM |
| vram\_data\_in | input | VRAM | The data the sprite renderer requested from VRAM |
| oam\_wr\_en | input | PPU Register | Selects if OAM is being read from or written to |
| spr\_pt\_sel | input | PPU Register | Determines the PPU pattern table address in VRAM |
| spr\_size\_sel | input | PPU Register | Determines the size of the sprites to be drawn |
| show\_spr\_left\_col | input | PPU Register | Determines if sprites on the leftmost 8 pixels of each scanline will be drawn |
| spr\_overflow | output | PPU Register | If more than 8 sprites fall on a single scanline this is set |
| spr\_zero\_hit | output | PPU Register | Set if sprite zero intersects with another sprite |
| oam\_data\_out[7:0] | output | PPU Register | OAM data is placed here on a read |
| vram\_addr\_out[13:0] | output | VRAM Mux | The VRAM address the sprite renderer wants to read from |
| spr\_vram\_req | output | VRAM Mux | Signals that the sprite renderer wants to read VRAM |
| spr\_pal\_out[3:0] | output | Pixel Mux | Signal that specifies the sprite palette data |
| spr\_pri\_out | output | Pixel Mux | Specifies the sprite to be drawn priority |

## VRAM Interface

The VRAM interface instantiates an Altera RAM IP core. Each read take 2 cycles one for the input and one for the output

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| vram\_addr[10:0] | input | PPU | Address from VRAM to read to or write from |
| vram\_data\_in[7:0] | input | PPU | The data to write to VRAM |
| vram\_en | input | PPU | The VRAM enable signal |
| vram\_rw | input | PPU | Selects if the current op is a read or write |
| vram\_data\_out[7:0] | output | PPU | The data that was read from VRAM on a read |

## DMA

The DMA is used to copy 256 bytes of data from the CPU address space into the OAM (PPU address space). The DMA is 4x faster than it would be to use str and ldr instructions to copy the data. While copying data, the CPU is paused.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| oamdma | input | PPU | When written to, the DMA will begin copying data to the OAM. If the value written here is XX then the data that will be copied begins at the address XX00 in the CPU RAM and goes until the address XXFF. Data will be copied to the OAM starting at the OAM address specified in the OAMADDR register of the OAM. |
| cpu\_ram\_q | input | CPU RAM | Data read in from CPU RAM will come here |
| dma\_done | output | CPU | Informs the CPU to pause while the DMA copies OAM data from the CPU RAM to the OAM section of the PPU RAM |
| cpu\_ram\_addr | output | CPU RAM | The address of the CPU RAM where we are reading data |
| cpu\_ram\_wr | output | CPU RAM | Read/write enable signal for CPU RAM |
| oam\_data | output | OAM | The data that will be written to the OAM at the address specified in OAMADDR |
| dma\_req | input | APU | High when the DMC wants to use the DMA |
| dma\_ack | output | APU | High when data on DMA |
| dma\_addr | input | APU | Address for DMA to read from \*\* CURRENTLY NOT USED \*\* |
| dma\_data | output | APU | Data from DMA to apu memory \*\* CURRENTLY NOT USED \*\* |

# Memory Mappers

Cartridges are a Read-Only Memory that contains necessary data to run games. However, it is some cases that a cartridge holds more data than the CPU can address to. In this case, memory mapper comes into play and changes the mapping as needed so that one address can point to multiple locations in a cartridge. For our case, the end goal is to get the game Super Mario Bros. running on our FPGA. This game does not use a memory mapper, so initially we will not be working on any memory mappers. If time permits, we might add support for the other memory mapping systems so that we can play other games.

These will be two ip catalog ROM blocks that are created using MIF files for Super Mario Bros. The will contain the information for the CPU and PPU RAM and VRAM respectively.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| addr | input | CPU/PPU | Address to read from |
| data | output | CPU/PPU | Data from the address |

# APU

The NES included an Audio Processing Unit (APU) to control all sound output. The APU contains five audio channels: two pulse wave modulation channels, a triangle wave channel, a noise channel (for random audio), and a delta modulation channel. Each channel is mapped to registers in the CPU’s address space. Each channel runs independently of each other. The outputs of all five channels are then combined using a non-linear mixing scheme. The APU also has a dedicated APU Status register. A write to this register can enable/disable any of the five channels. A read to this register can tell you if each channel still has a positive count on their respective timers. In addition, a read to this register will reveal any DMC or frame interrupts.

The APU still remains a stretch goal for us, however, the APU can still interrupt the DMA and the CPU so we will need to have a block for the APU that can generate these interrupts so that the software will still work correctly. We will send dma requests to the DMA for now, which will pause the DMA and CPU for the required amount of time if the APU was actually functioning. We will also send IRQ’s to the CPU as needed, but won’t process any audio.

## APU Registers

|  |  |  |  |
| --- | --- | --- | --- |
| $4000 | First pulse wave | DDLC VVVV | Duty, Envelope Loop, Constant Volume, Volume |
| $4001 | First pulse wave | EPPP NSSS | Enabled, Period, Negate, Shift |
| $4002 | First pulse wave | TTTT TTTT | Timer low |
| $4003 | First pulse wave | LLLL LTTT | Length counter load, Timer high |
| $4004 | Second pulse wave | DDLC VVVV | Duty, Envelope Loop, Constant Volume, Volume |
| $4005 | Second pulse wave | EPPP NSSS | Enabled, Period, Negate, Shift |
| $4006 | Second pulse wave | TTTT TTTT | Timer low |
| $4007 | Second pulse wave | LLLL LTTT | Length counter load, Timer high |
| $4008 | Triangle wave | CRRR RRRR | Length counter control, linear count load |
| $4009 | Triangle wave |  | Unused |
| $400A | Triangle wave | TTTT TTTT | Timer low |
| $400B | Triangle wave | LLLL LTTT | Length counter load, Timer high |
| $400C | Noise Channel | --LC VVVV | Envelope Loop, Constant Volume, Volume |
| $400D | Noise Channel |  | Unused |
| $400E | Noise Channel | L--- PPPP | Loop Noise, Noise Period |
| $400F | Noise Channel | LLLL L--- | Length counter load |
| $4010 | Delta modulation channel | IL-- FFFF | IRQ enable, Loop, Frequency |
| $4011 | Delta modulation channel | -LLL LLLL | Load counter |
| $4012 | Delta modulation channel | AAAA AAAA | Sample Address |
| $4013 | Delta modulation channel | LLLL LLLL | Sample Length |
| $4015 (write) | APU Status Register Writes | ---D NT21 | Enable DMC, Enable Noise, Enable Triangle, Enable Pulse 2/1 |
| $4015 (read) | APU Status Register Read | IF-D NT21 | DMC Interrupt, Frame Interrupt, DMC Active, Length Counter > 0 for Noise, Triangle, and Pulse Channels |
| $4017 | APU Frame Counter | MI-- ---- | Mode (0 = 4 step, 1 = 5 step), IRQ inhibit flag |

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| addr | input | CPU | Apu memory address |
| data | input | CPU | Data for APU |
| write | input | CPU | Write request to APU |
| dma\_req | output | PPU DMA | High when the DMC wants to use the DMA |
| dma\_ack | input | PPU DMA | High when data on DMA |
| dma\_addr | output | PPU DMA | Address for DMA to read from \*\* CURRENTLY NOT USED \*\* |
| dma\_data | input | PPU DMA | Data from DMA to apu memory \*\* CURRENTLY NOT USED \*\* |
| irq | output | CPU | interrupt form apu |

# Controller (SPART)

Controller is the input from the user to play the the game. We will be using the keyboard instead of the joypad as the controller. Keyboard will be communicating with FPGA, using SPART communication method. The input is then memory mapped on $4016 and $4017 for CPU to read every short time frame.

When writing high to address $4016 bit 0, the controllers are continuously, loaded with the states of each button. Once address $4016 bit 0 goes low, the data from the controllers can be read by reading from address $4016. The data will be read in serially on bit 0. The first read will return the state of button A, then B, Select, Start, Up, Down, Left, Right. It will read 1 if the button is pressed and 0 otherwise. Any read after the negedge of writing to $4016 bit 0 and after reading the first 8 button values, will be a 1.

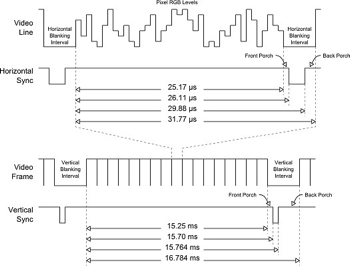
|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| TxD | output | UART | Transmit data line |
| RxD | input | UART | Receive data line |
| addr | input | CPU | Controller address 0 for $4016, 1 for $4017 |
| dout[7:0] | inout | CPU | Data from/to the CPU |
| cs | input | CPU | Chip select |
| wr | input | CPU | write enable signal |



# 

# VGA

The VGA interface consists of sending the pixel data to the screen one row at a time from left to right. In between each row it requires a special signal called horizontal sync (hsync) to be asserted at a specific time when only black pixels are being sent, called the blanking interval. This happens until the bottom of the screen is reached when another blanking interval begins where the interface is only sending black pixels, but instead of hsync being asserted the vertical sync signal is asserted.



The main difficulty with the VGA interface will be designing a system to take the PPU output (a 256x240 image) and converting it into a native resolution of 640x480 or 1280x960. Following is the interface description.



|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| V\_BLANK\_N | output |  | Syncing each pixel |
| VGA\_R[7:0] | output |  | Red pixel value |
| VGA\_G[7:0] | output |  | Green pixel value |
| VGA\_B[7:0] | output |  | Blue pixel value |
| VGA\_CLK | output |  | VGA clock |
| VGA\_HS | output |  | Horizontal line sync |
| VGA\_SYNC\_N | output |  | 0 |
| VGA\_VS | output |  | Vertical line sync |
| back\_pixel\_data[7:0] | input | PPU | Background pixel data to be sent to the display |
| sprte\_pixel\_data[7:0] | input | PPU | Sprite pixel data to be sent to the display |
| back\_pixel\_en | input | PPU | Tells you to write background pixel or default background pixel |
| sprite\_pixel\_en | input | PPU | Tells you to prioritize the sprite pixel over the background pixels |
| ppu\_clock | input | PPU | pixel data is updated every ppu clock cycle |

## VGA Clock Gen

This is the same module that was used in Lab 2 except with a different source clock.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| VGA\_CLK | output | VGA | Clock synced to VGA timing |
| locked | output |  | Locks VGA until clock is ready |

## VGA Timing Gen

This block is responsible for generating the timing signals for VGA with a screen resolution of 480x640. This includes the horizontal and vertical sync signals as well as the blank signal for each pixel.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| VGA\_CLK | input | Clock Gen | vga\_clk |
| rst\_n | input |  | System active low reset |
| V\_BLANK\_N | output | VGA, Ram Reader | Syncing each pixel |
| VGA\_HS | output | VGA | Horizontal line sync |
| VGA\_VS | output | VGA | Vertical line sync |

## 

## VGA Display Plane

The PPU will output sprite and background pixels to the VGA module, as well as enables for each. The display planes job is to use the enable signals to determine which pixel has priority on the screen. It will then update the RAM block at the appropriate address with the pixel data. In addition, the display plane will give our image black borders on the side and multiply the pixels to the 1280x960 format.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  | System clock |
| rst\_n | input |  | System active low reset |
| ppu\_clock | input | PPU | Clock speed that the pixels from the PPU come in |
| wr\_address | input | RAM | Address to write to |
| wr\_req | output | RAM | Write data to the RAM |
| data\_out[7:0] | output | RAM | The pixel data to store in RAM |
| back\_pixel\_data[7:0] | input | PPU | Background pixel data to be sent to the display |
| sprte\_pixel\_data[7:0] | input | PPU | Sprite pixel data to be sent to the display |
| back\_pixel\_en | input | PPU | Tells you to write background pixel or default background pixel |
| sprite\_pixel\_en | input | PPU | Tells you to prioritize the sprite pixel over the background pixels |

## 

## VGA RAM

This will be the 2-port RAM block from the Quartus IP catalog. It allows simultaneous read and writes from memory and will store the pixels that we print to the screen

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  |  |
| rst\_n | input |  | System active low reset |
| wr\_address | input | Display Plane | Address to write to |
| wr\_req | input | Display Plane | Request to write data |
| data\_in[7:0] | input | Display Plane | The data into the RAM |
| rd\_req | input | RAM Reader | Read data out from RAM |
| rd\_address | input | RAM Reader | Address to read from |
| data\_out[7:0] | output | RAM Reader | data out from RAM |

## 

## VGA RAM Reader

The RAM Reader is responsible for reading data from the correct address in the RAM block and outputting it as an RGB signal to the VGA. It will update the RGB signals every time the blank signal goes high. The NES supported a 256x240 image, which we will be converting to a 640x480 image. This means that the 256x240 image will be multiplied by 2, resulting in a 512x480 image. The remaining 128 pixels on the horizontal line will be filled with black pixels by this block. Lastly, this block will take use the pixel data from the PPU and the NES Palette RGB colors, to output the correct colors to the VGA.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Signal Type | Source/Dest | Description |
| clk | input |  |  |
| rst\_n | input |  | System active low reset |
| rd\_req | output | RAM | Read data out from RAM |
| rd\_address | output | RAM | Address to read from |
| data\_out[7:0] | input | RAM | data out from RAM |
| VGA\_R[7:0] | output |  | VGA Red pixel value |
| VGA\_G[7:0] | output |  | VGA Green pixel value |
| VGA\_B[7:0] | output |  | VGA Blue pixel value |
| VGA\_Blnk[7:0] | input | Time Gen | VGA Blank signal (high when we write each new pixel) |

# Software

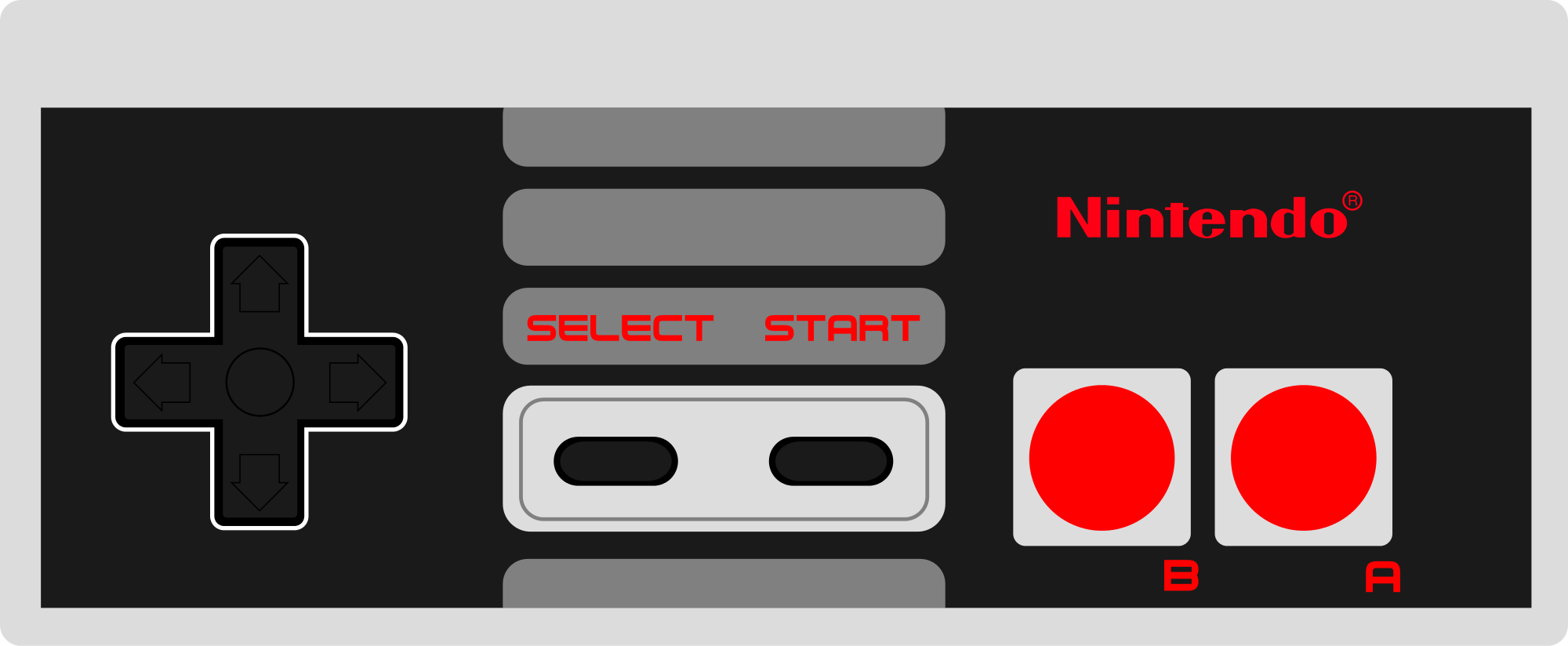
## Controller Simulator

In order to play games on the NES and provide input to our FPGA, we will have a java program that uses the javax.comm.\* library to read and write data serially using the SPART interface. When we receive a packet from the NES, it will indicate that we want to get the data from the controller. Then we will send a packet which indicates which buttons are being pressed.



|  |  |  |  |
| --- | --- | --- | --- |
| Packet name | Packet type | Packet Format | Description |
| Controller Data | output | ABST-UDLR | This packet indicates which buttons are being pressed. A 1 indicates pressed, a 0 indicates not pressed.  (A) A button, (B) B button, (S) Select button, (T) Start button, (U) Up, (D) Down, (L) Left, (R) Right |

The NES controller had a total of 8 buttons, as shown below.



The NES buttons will be mapped to specific keys on the keyboard. The keyboard information will be obtained using KeyListeners in the java.awt.\* library. The following table indicates how the buttons are mapped and their function in Super Mario Bros.

|  |  |  |
| --- | --- | --- |
| Keyboard button | NES Equivalent | Super Mario Bros. Function |
| X Key | A Button | Jump (Hold to jump higher) |
| Z Key | B Button | Sprint (Hold and use arrow keys) |
| Tab Key | Select Button | Pause Game |
| Enter Key | Start Button | Start Game |
| Up Arrow | Up on D-Pad | No function |
| Down Arrow | Down on D-Pad | Enter pipe (only works on some pipes) |
| Left Arrow | Left on D-Pad | Move left |
| Right Arrow | Right on D-Pad | Move right |

## PPU ROM’s Memory Map

This table shows how the PPU’s memory is laid out. The Registers are explained in greater detail in the Architecture Document.

|  |  |
| --- | --- |
| Address Range | Description |
| 0x0000 - 0x0FFF | Pattern Table 0 |
| 0x1000 - 0x1FFF | Pattern Table 1 |
| 0x2000 - 0x23BF | Name Table 0 |
| 0x23C0 - 0x23FF | Attribute Table 0 |
| 0x2400 - 0x27BF | Name Table 1 |
| 0x27C0 - 0x27FF | Attribute Table 1 |
| 0x2800 - 0x2BBF | Name Table 2 |
| 0x2BC0 - 0x2BFF | Attribute Table 2 |
| 0x2C00 - 0x2FBF | Name Table 3 |
| 0x2FC0 - 0x2FFF | Attribute Table 3 |
| 0x3000 - 0x3EFF | Mirrors 0x2000 - 0x2EFF |
| 0x3F00 - 0x3F0F | Background Palettes |
| 0x3F10 - 0x3F1F | Sprite Palettes |
| 0x3F20 - 0x3FFF | Mirrors 0x3F00 - 0x3F1F |
| 0x4000 - 0xFFFF | Mirrors 0x0000 - 0x3FFF |

## CPU ROM’s Memory Map

This table explains how the CPU’s memory is laid out. The Registers are explained in greater detail in the Architecture document.

|  |  |
| --- | --- |
| Address Range | Description |
| 0x0000 - 0x00FF | Zero Page |
| 0x0100 - 0x1FF | Stack |
| 0x0200 - 0x07FF | RAM |
| 0x0800 - 0x1FFF | Mirrors 0x0000 - 0x07FF |
| 0x2000 - 0x2007 | Registers |
| 0x2008 - 0x3FFF | Mirrors 0x2000 - 0x2007 |
| 0x4000 - 0x401F | I/O Registers |
| 0x4020 - 0x5FFF | Expansion ROM |
| 0x6000 - 0x7FFF | SRAM |
| 0x8000 - 0xBFFF | Program ROM Lower Bank |
| 0xC000 - 0xFFFF | Program ROM Upper Bank |

## Assembler

We will include an assembler that allows custom software to be developed for our console. This assembler will convert assembly code to machine code for the NES on .mif files that we can load into our FPGA. It will include support for labels and commenting.The ISA is specified in the table below:

### Opcode Table

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **Mode** | **Hex** | **Opcode** | **Mode** | **Hex** | **Opcode** | **Mode** | **Hex** |
| ADC | Immediate | 69 | DEC | Zero Page | C6 | ORA | Absolute | 0D |
| ADC | Zero Page | 65 | DEC | Zero Page, X | D6 | ORA | Absolute, X | 1D |
| ADC | Zero Page, X | 75 | DEC | Absolute | CE | ORA | Absolute, Y | 19 |
| ADC | Absolute | 6D | DEC | Absolute, X | DE | ORA | Indirect, X | 01 |
| ADC | Absolute, X | 7D | DEX | Implied | CA | ORA | Indirect, Y | 11 |
| ADC | Absolute, Y | 79 | DEY | Implied | 88 | PHA | Implied | 48 |
| ADC | Indirect, X | 61 | EOR | Immediate | 49 | PHP | Implied | 08 |
| ADC | Indirect, Y | 71 | EOR | Zero Page | 45 | PLA | Implied | 68 |
| AND | Immediate | 29 | EOR | Zero Page, X | 55 | PLP | Implied | 28 |
| AND | Zero Page | 25 | EOR | Absolute | 4D | ROL | Accumulator | 2A |
| AND | Zero Page, X | 35 | EOR | Absolute, X | 5D | ROL | Zero Page | 26 |
| AND | Absolute | 2D | EOR | Absolute, Y | 59 | ROL | Zero Page, X | 36 |
| AND | Absolute, X | 3D | EOR | Indirect, X | 41 | ROL | Absolute | 2E |
| AND | Absolute, Y | 39 | EOR | Indirect, Y | 51 | ROL | Absolute, X | 3E |
| AND | Indirect, X | 21 | INC | Zero Page | E6 | ROR | Accumulator | 6A |
| AND | Indirect, Y | 31 | INC | Zero Page, X | F6 | ROR | Zero Page | 66 |
| ASL | Accumulator | 0A | INC | Absolute | EE | ROR | Zero Page, X | 76 |
| ASL | Zero Page | 06 | INC | Absolute, X | FE | ROR | Absolute | 6E |
| ASL | Zero Page, X | 16 | INX | Implied | E8 | ROR | Absolute, X | 7E |
| ASL | Absolute | 0E | INY | Implied | C8 | RTI | Implied | 40 |
| ASL | Absolute, X | 1E | JMP | Indirect | 6C | RTS | Implied | 60 |
| BCC | Relative | 90 | JMP | Absolute | 4C | SBC | Immediate | E9 |
| BCS | Relative | B0 | JSR | Absolute | 20 | SBC | Zero Page | E5 |
| BEQ | Relative | F0 | LDA | Immediate | A9 | SBC | Zero Page, X | F5 |
| BIT | Zero Page | 24 | LDA | Zero Page | A5 | SBC | Absolute | ED |
| BIT | Absolute | 2C | LDA | Zero Page, X | B5 | SBC | Absolute, X | FD |
| BMI | Relative | 30 | LDA | Absolute | AD | SBC | Absolute, Y | F9 |
| BNE | Relative | D0 | LDA | Absolute, X | BD | SBC | Indirect, X | E1 |
| BPL | Relative | 10 | LDA | Absolute, Y | B9 | SBC | Indirect, Y | F1 |
| BRK | Implied | 00 | LDA | Indirect, X | A1 | SEC | Implied | 38 |
| BVC | Relative | 50 | LDA | Indirect, Y | B1 | SED | Implied | F8 |
| BVS | Relative | 70 | LDX | Immediate | A2 | SEI | Implied | 78 |
| CLC | Implied | 18 | LDX | Zero Page | A6 | STA | Zero Page | 85 |
| CLD | Implied | D8 | LDX | Zero Page, Y | B6 | STA | Zero Page, X | 95 |
| CLI | Implied | 58 | LDX | Absolute | AE | STA | Absolute | 8D |
| CLV | Implied | B8 | LDX | Absolute, Y | BE | STA | Absolute, X | 9D |
| CMP | Immediate | C9 | LDY | Immediate | A0 | STA | Absolute, Y | 99 |
| CMP | Zero Page | C5 | LDY | Zero Page | A4 | STA | Indirect, X | 81 |
| CMP | Zero Page, X | D5 | LDY | Zero Page, X | B4 | STA | Indirect, Y | 91 |
| CMP | Absolute | CD | LDY | Absolute | AC | STX | Zero Page | 86 |
| CMP | Absolute, X | DD | LDY | Absolute, X | BC | STX | Zero Page, Y | 96 |
| CMP | Absolute, Y | D9 | LSR | Accumulator | 4A | STX | Absolute | 8E |
| CMP | Indirect, X | C1 | LSR | Zero Page | 46 | STY | Zero Page | 84 |
| CMP | Indirect, Y | D1 | LSR | Zero Page, X | 56 | STY | Zero Page, X | 94 |
| CPX | Immediate | E0 | LSR | Absolute | 4E | STY | Absolute | 8C |
| CPX | Zero Page | E4 | LSR | Absolute, X | 5E | TAX | Implied | AA |
| CPX | Absolute | EC | NOP | Implied | EA | TAY | Implied | A8 |
| CPY | Immediate | C0 | ORA | Immediate | 09 | TSX | Implied | BA |
| CPY | Zero Page | C4 | ORA | Zero Page | 05 | TXA | Implied | 8A |
| CPY | Absolute | CC | ORA | Zero Page, X | 15 | TXS | Implied | 9A |
|  |  |  |  |  |  | TYA | Implied | 98 |

### NES Assembly Format

Our assembler will allow the following input format, each instruction/label will be on its own line. In addition unlimited whitespace is allowed:

|  |  |  |
| --- | --- | --- |
| Instruction Type | Format | Description |
| Label | Label\_Name: | Cannot be the same as an opcode name. Allows reference from branch opcodes. |
| Comment | ; Comment goes here | Anything after the ; will be ignored |
| CPU Start | \_CPU: | Signals the start of CPU memory |
| PPU Start | \_PPU: | Signals the start of PPU memory |
| Accumulator | <OPCODE> | Accumulator is value affected by Opcode |
| Implied | <OPCODE> | Operands implied by opcode. ie. TXA has X as source and Accumulator as destination |
| Immediate | <OPCODE> #<Immediate> | The decimal number will be converted to binary and used as operand |
| Absolute | <OPCODE> $<ADDR/LABEL> | The byte at the specified address is used as operand |
| Zero Page | <OPCODE> $<BYTE OFFSET> | The byte at address $00XX is used as operand. |
| Relative | <OPCODE> $<BYTE OFFSET/LABEL> | The byte at address PC +/- Offset is used as operand. Offset can range -128 to +127 |
| Absolute Index | <OPCODE> $<ADDR/LABEL>,<X or Y> | Absolute but value in register added to address. |
| Zero Page Index | <OPCODE> $<BYTE OFFSET>,<X or Y> | Zero page but value in register added to offset. |
| Zero Page X Indexed Indirect | <OPCODE> ($<BYTE OFFSET>,X) | Value in X added to offset. Address in $00XX (where XX is new offset) is used as the address for the operand. |
| Zero Page Y Indexed Indirect | <OPCODE> ($<BYTE OFFSET>),Y | The address in $00XX, where XX is byte offset, is added to the value in Y and is used as the address for the operand. |

### The following table shows the valid number format for the byte offsets, addresses, and immediates:

|  |  |  |
| --- | --- | --- |
| Immediate Decimal (Signed) | #<(-)DDD> | Max 127, Min -128 |
| Immediate Hexadecimal (Signed) | #$<HH> |  |
| Immediate Binary (Signed) | #%<BBBB.BBBB> | Allows ‘.’ in between bits |
| Address/Offset Hex | $<Addr/Offset> | 8 bits offset, 16 bits address |
| Address/Offset Binary | $%<Addr/Offset> | 8 bits offset, 16 bits address |
| Offset Decimal (Relative only) | #<(-)DDD> | Relative instructions can’t be Immediate, so this is allowed.  Max 127, Min -128 |

### 

### Invoking Assembler

|  |  |
| --- | --- |
| Usage | Description |
| java NESAssemble <input file> <cpuouput.mif> <ppuoutput.mif> | Reads the input file and outputs the CPU ROM to cpuoutput.mif and the PPU ROM to ppuoutput.mif |

## iNES ROM Converter

Most NES games are currently available online in files of the iNES format, a header format used by most software NES emulators. Our NES will not support this file format. Instead, we will write a java program that takes an iNES file as input and outputs two .mif files that contain the CPU RAM and the PPU VRAM. These files will be used to instantiate the ROM’s of the CPU and PPU in our FPGA.

|  |  |
| --- | --- |
| Usage | Description |
| java NEStoMIF <input.nes> <cpuouput.mif> <ppuoutput.mif> | Reads the input file and outputs the CPU RAM to cpuoutput.mif and the PPU VRAM to ppuoutput.mif |