Worcester Polytechnic Institute
Department of Electrical and Computer Engineering

ECE-574: Project 3

Guilherme Meira
December 12, 2014

Contents

1 Introduction

This final project aims to implement the Chip-8 language[1]. Chip-8 was initially used in the mid-1970s to allow video-games to be more easily programmed and it gained a lot of popularity in the late-1980s, when some interpreters were developed for graphing calculators.

There are also two extensions to Chip-8. The most famous is the Super Chip (S-Chip, for short) that introduces a higher resolution display and some new instructions for drawing and interaction with the HP-48 calculator. This extension was implemented in this project, because the most interesting games available are for the S-Chip. There is also the MegaChip extension, which adds support for color, 8-bit sound, even higher resolution and some other features, but the only game I could find using this extension is a Pacman clone called MegaBlinky, so I considered that this extension is not worth the difficulty of implementing it.

This project makes use of most of the knowledge obtained during the ECE-574 course. All instructions are implemented in hardware, the interaction with the interpreter is done through the serial port that is controlled by the Microblaze and the game data is stored on the external SRAM. It also explores the use of the block RAM that is present in the FPGA, which was not explicitly covered during the course.

Verilog was the chosen HDL for this project, as it seems more productive to me. VHDL is unnecessarily verbose and has a complex type system. The Microblaze is programmed in C and also some Java is used during the project.

This report is divided in 12 sections. Section 2 summarizes every information I could find about the Chip-8 and the S-Chip. Sections 3 to 8 describe the implementation of each component of the project. Section 9 shows how the components are put together. Section 10 explains the approach used to simulate this project. Section 11 presents the resource usage and other information from the synthesis report. Section 12 concludes this report with a general overview of the main problems faced during implementation, some possible improvements and learned lessons.

2 Chip-8 and S-Chip

Although there is plenty of documentation about the Chip-8 available, there is no official document explaining it. Everything that is available online was made by third parties, probably by reverse engineering, because the source code for the original interpreter is not available. Thus, some information is conflicting. In this section, the Chip-8 and the S-Chip are explained and whenever there is a conflict between documentations, it will be

[1] http://en.wikipedia.org/wiki/CHIP-8

pointed out.

The two primary sources of information about the Chip-8 and the S-Chip are the Cowgod's Chip-8 Technical Reference [2] and the Erik Bryntse's S-Chip Technical Reference[3]. Other sources there were also used are David Winter's Emulator Documentation[4] and Peter Miller's Chip-8 Reference Manual[5]. Because of its simplicity, there is a lot of Chip-8/S-Chip implementations available online, so whenever needed, I went through the source code of some of those implementations.

2.1 Memory

Chip-8 can access up to 4KB of memory, from location `0x000` to `0xFFF`. The first 512 bytes of the memory are usually not used by the programs, because originally the interpreter was located in this space. In modern implementations this area is commonly used to store font sprites. The programs always begin after those bytes, at the address `0x200`, as shown on Figure 2.1.

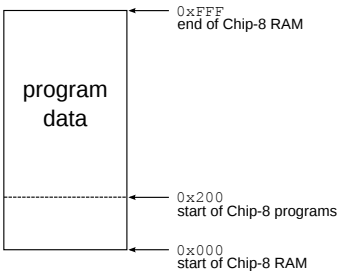

Figure 2.1: Chip-8 memory

2.2 Registers

There are 16 8-bit general purpose registers, usually referred to as `Vx`, being `x` a hexadecimal number from `0` to `F`. The `VF` register is used by some instructions as a flag register.

There is a 16-bit register called `I`, which is used to access memory. As Chip-8 has 4KB of RAM, only the 12 least significant bits should be used.

The program counter is a 16-bit register and holds the address of the current instruction. The programmer can set it by using jumps or function calls, but it can't be read.

There is also an implicit stack. The stack holds 16-bit addresses and is used to support nested function calls. The stack should support at least 16 levels of nesting and is not

[2] http://devernay.free.fr/hacks/chip8/C8TECH10.HTM
[3] http://devernay.free.fr/hacks/chip8/schip.txt
[4] http://vanbeveren.byethost13.com/stuff/CHIP8.pdf
[5] http://chip8.sourceforge.net/chip8-1.1.pdf

accessible by the programmer.

2.3 Keyboard

The original keyboard was a 16-key hexadecimal keypad, as shown on Figure 2.2, and the mapping of this keyboard to today's platforms is highly implementation-dependent. The approach used for this mapping in this project is discussed later.

1	2	3	C
4	5	6	D
7	8	9	E
A	0	B	F

Figure 2.2: Chip-8 keyboard

2.4 Display

The original Chip-8 has a 64x32-pixel monochrome display. The origin is on the top-left corner.

S-Chip introduces an extended video mode with 128x64 pixels. There are instructions that allow a program to switch between the video modes at any time, but the behavior of the screen contents is undefined. A program that uses the extended video mode should switch to that mode at the beggining of execution and stay in that mode. The video modes are compared in Figure 2.3.

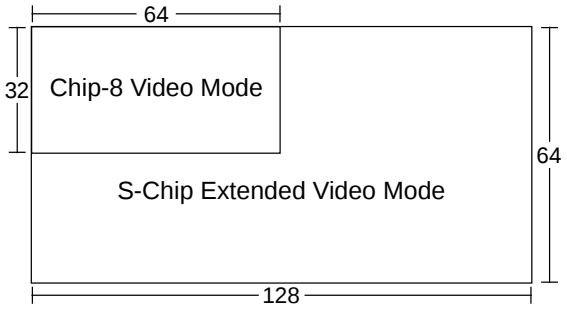

Figure 2.3: Chip-8/S-Chip video modes

The drawing is always done by using sprites. As the screen is monochromatic, sprites use 1 bit for each pixel. Chip-8 sprites can have from 1 to 15 bytes and each byte

represents 8 bits in a row, so the sprites can be from 8x1 to 8x15 pixels. S-Chip introduces a 16x16 sprite.

The interpreter must provide sprites for the hexadecimal digits from `0` to `F`. These sprites are 5-bytes large, but only half of the sprite is used, which gives us fonts with dimensions 4x5. S-Chip introduces 8x10 fonts for the digits from `0` to `9`. This font data is usually stored in the area that originally was reserved for the interpreter.

2.5 Timers and sound

Chip-8 provides two timers called delay and sound timers. Whenever a timer contains a value greater than zero it decrements at a rate of 60Hz until it reaches zero.

The delay timer is a general purpose timer that can be written and read by the programmer. The sound timer can only be written and whenever its value is different from zero a buzzer produces a sound. Chip-8 can produce only one tone and the frequency is implementation-dependent.

2.6 Instruction set

This subsection explains the opcodes available in the Chip-8 language and the ones added by the S-Chip extension. Every instruction is 2 bytes long and should be stored on even byte boundaries, even though some implementations do not require this. The opcodes are stored in big endian. There isn't a fixed clock speed for Chip-8 programs, modern implementations usually introduce an adjustable slow down to make the games playable. Game events with some timing requirement are implemented using the delay timer.

- `0`***nnn*** **- SYS *addr***
  Jump to a machine code routine at *nnn*. Used only on computers on which Chip-8 was originally implemented. It's ignored by modern interpreters.
- `00E0` **- CLS**
  Clear the display
- `00EE` **- RET**
  Return from a subroutine. The program counter is set to the address at the top of the stack and the stack pointer is decremented.
- `1`***nnn*** **- JP *addr***
  Jump to location *nnn*. The program counter is set to the specified address.
- `2`***nnn*** **- CALL *addr***
  Call subroutine at *nnn*. The current PC is put on the top of the stack and then it's set to *nnn*.

- `3xkk` - **SE V*x*,*byte***
  Skip next instruction if `Vx = kk`. Register `Vx` is compared to `kk` and if they are equal, the program counter is incremented by two instructions. Note that each instruction is 2 bytes long. To skip a instruction the program counter must be incremented by 4 bytes.

- `4xkk` - **SNE V*x*,*byte***
  Skip next instruction if `Vx != kk`. Register `Vx` is compared to `kk` and if they are different, the program counter is incremented by two instructions.

- `5xy0` - **SE V*x*,V*y***
  Skip next instruction if `Vx = Vy`. Registers `Vx` and `Vy` are compared and if they are equal, the program counter is incremented by two instructions.

- `6xkk` - **LD V*x*,*byte***
  The value `kk` is loaded into register `Vx`.

- `7xkk` - **ADD V*x*,*byte***
  The value `kk` is added to the current content of `Vx` and stored in `Vx`.

- `8xy0` - **LD V*x*,V*y***
  Store the current value of `Vy` into `Vx`.

- `8xy1` - **OR V*x*,V*y***
  Perform a bitwise OR between the values of `Vx` and `Vy` and store in `Vx`.

- `8xy2` - **AND V*x*,V*y***
  Perform a bitwise AND between the values of `Vx` and `Vy` and store in `Vx`.

- `8xy3` - **XOR V*x*,V*y***
  Perform a bitwise XOR between the values of `Vx` and `Vy` and store in `Vx`.

- `8xy4` - **ADD V*x*,V*y***
  Add the values of `Vx` and `Vy` and store the result in `Vx`. The register `VF` is used as a carry flag (it is set to `1` if the result is greater than 255 or `0` otherwise).

- `8xy5` - **SUB V*x*,V*y***
  Subtract `Vy` from `Vx` and store the result in `Vx` (`Vx` = `Vx-Vy`). The register `VF` is set to the negation of the the borrow, which means that it's set to `1` if `Vx >= Vy` and to `0` otherwise. There is some conflict about this instruction when the values are equal. David Winter's documentation says that `VF` should be set to `1` and Cowgod's documentation says it should be set to `0`. It makes more sense to me to set it to `1` in those cases (and that is what is done in this project), as when a number is subtracted from itself there is no borrow. The implementations I found also behave in this way and I also found one game (Ant in Search of Coke) that relies on this behavior.

- `8xy6` - **SHR V*x***
  The register `VF` is set to the value of the least significant bit of `Vx` and then `Vx` is shifted to the right by one place (divided by 2).

- `8xy7` - **SUBN V*x*,V*y***
  Perform the subtraction `Vx` = `Vy` - `Vx`. The same discussion about the borrow flag is valid here.

- `8xyE` - **SHL V*x***
  The register `VF` is set to the value of the most significant bit of `Vx` and then `Vx` is shifted to the left by one place (multiplied by 2).

- `9xy0` - **SNE V*x*,V*y***
  Skip next instruction if `Vx != Vy`. Registers `Vx` and `Vy` are compared and if they are different, the program counter is incremented by two instructions.

- `Annn` - **LD I,*addr***
  Set the value of the register `I` to *nnn*.

- `Bnnn` - **JP V0,*addr***
  Jump to location *nnn* + `V0`. The program counter is set to the address *nnn* added to the current value of `V0`.

- `Cxkk` - **RND V*addr*,*byte***
  A random value between 0 and 255 is generated and ANDed with `kk`. The result is stored in `Vx`.

- `Dxyn` - **DRW V*x*,V*y*,*nibble***
  Draw a `n`-byte sprite starting at the memory location pointed by `I` at coordinates (`Vx`,`Vy`). The drawing is done by XORing the sprite with the value that is already present on the screen (a bit `1` on the sprite flips the pixel on screen, a bit `0` leaves the pixel unchanged). If during the draw a pixel is erased, `VF` is set to `1`, otherwise it is set to `0`. It's possible for the registers to contain a value bigger than the screen size, so, in fact, the actual drawing position is obtained by taking the remainder of the division of the register value by the screen size. I couldn't find this behavior documented anywhere, but I found at least one implementation that does that and one game that relies on this behavior. Good news is that it's easy to do because the screen size is always a power of two, so all we have to do is discard some higher order bits. S-Chip extends this function in the following way: if `n` is zero and the interpreter is on extended video mode, a 16x16 sprite is drawn. There is a conflict here when a sprite is positioned in a way that part of the sprite is outside the screen. Cowgod's documentation says that it wraps around to the oposite side of the screen, while Peter Miller's documentation says that the part of the sprite outside the screen is simply ignored and David Winter's documentation does not mention this special case. There are interpreters implementing both behaviors and the games seem to not rely on this. So, for most games any behavior should work well. In this

project I implemented the wrapping behavior. Peter Miller's documentation also says that on low resolution mode this function delays until the start of a 60Hz clock cycle before drawing. I couldn't find this behavior in other implementations and it's also not implemented in this project.

- `Ex9E` - **SKP V*x***
  Skip next instruction if key with value corresponding to the current value of `Vx` is pressed.

- `ExA1` - **SKNP V*x***
  Skip next instruction if key with value corresponding to the current value of `Vx` is not pressed.

- `Fx07` - **LD V*x*,DT**
  The current value of the delay timer is stored in `Vx`.

- `Fx0A` - **LD V*x*,K**
  This is probably the most controversial opcode. According to Cowgod's and David Winter's documentations, the execution stops until a key is pressed, then the value of that key is stored in `Vx`. Peter Miller's documentation adds a lot of details: the value stored in `Vx` is the lowest value, in case of simultaneous key presses, execution is blocked until the key is released (or the execution is allowed to continue if this key press is not seen by any other call to input functions). According to his documentation, the buzzer should also sound while no key is being pressed. The interpreters I found do not implement those details. Games should not rely on that (or not use this function at all). In this project these details are also not implemented.

- `Fx15` - **LD DT,V*x***
  The delay timer is set to the current value of `Vx`.

- `Fx18` - **LD ST,V*x***
  The sound timer is set to the current value of `Vx`.

- `Fx1E` - **ADD I,V*x***
  The current value of `Vx` is added to `I` and the results stored in `I`.

- `Fx29` - **LD F,V*x***
  The register `I` is set to the location of the low resolution sprite corresponding to the current value of `Vx`.

- `Fx33` - **LD B,V*x***
  The BCD representation of `Vx` is stored in memory starting at location pointed by `I`. The hundreds digit is stored at `[I]`, the tens digit at `[I+1]` and the ones digit at `[I+2]`.

- `Fx55` - **LD [I],V*x***
  Store the values of the registers `V0` through `Vx` into memory, starting at the location pointed by `I`.
- `Fx15` - **LD V*x*,[I]**
  Read registers `V0` through `Vx` from memory starting at the location pointed by `I`.
- `00Cn` - **SCD *nibble***
  This inscruction was added by S-Chip. Scroll the screen down `n` rows.
- `00FB` - **SCR**
  This inscruction was added by S-Chip. Scroll the screen 4 columns to the right.
- `00FB` - **SCL**
  This inscruction was added by S-Chip. Scroll the screen 4 columns to the left.
- `00FD` - **EXIT**
  This instruction was added by S-Chip. Finish the program execution.
- `00FE` - **LOW**
  This instruction was added by S-Chip. Enter the low resolution mode.
- `00FF` - **HIGH**
  This instruction was added by S-Chip. Enter the high (extended) resolution mode.
- `Fx30` - **LD HF,V*x***
  This instruction was added by S-Chip. The register `I` is set to the location of the high resolution sprite corresponding to the current value of `Vx`.
- `Fx75` - **LD R,V*x***
  This instruction was added by S-Chip. Store the values from `V0` through `Vx` in the RPL user flags of the HP-48 calculator. The value of `x` must be less than or equals to 7. This is used by some games to save game state to recover it later. Some implementations create extra registers to mimic the RPL flags. In this project this instruction is a no-operation.
- `Fx65` - **LD V*x*,R**
  This instruction was added by S-Chip. Read the values from `V0` through `Vx` from the RPL user flags of the HP-48 calculator. The value of `x` must be less than or equals to 7. This is used by some games to restore game state. In this project this instruction is a no-operation.

3 Video controller

The video controller is the component responsible for generating the `red`, `green` and `blue` signals for the VGA display. It makes use of the VGA controller provided by Digilent that was explained in details on previous reports. This component also receives commands

from the Chip-8 core to perform drawing and shifting.

The current state of the screen is stored in the block memory of the FPGA. To access the block memory, the IP Core Generator was used to create a dual-port block RAM. The component generated by the Core Generator is specially useful because it allows the use of two independent read and write ports clocked at different speeds. In this project, port A is read and written by the circuit that interacts with the Chip-8 core and is clocked at 100MHz and port B is only read by the circuit that generates the VGA output and is clocked at 25MHz.

The generated block memory can store 64 entries of 128 bits each. As the block RAM that is available on the FPGA is only 16 bits wide, internally the generated core must combine four 32-bit block RAMs to generate a 128-bit memory. Although it is a big waste of resources, this approach makes the video logic much simpler.

The VGA mode used in this project is the 640x480, but the Chip-8 screen is 128x64 or 64x32, depending on the video mode, so the image needs to be scaled up to make a better use of the available space on the screen. To preserve the proportions, the Chip-8 image is scaled to 640x320 and positioned at the center of the screen, starting at line 80 on the VGA display, as shown on Figure 3.1. There are very powerful scaling algorithms available that can make the scaled image look smoother, but this project uses the same approach of all implementations I could find and simply makes the pixels bigger. Thus, on Chip-8 video mode, each pixel is 10x10 and on extended video mode each pixel is 5x5. A vertical and a horizontal counter are used to implement this simple scaling.

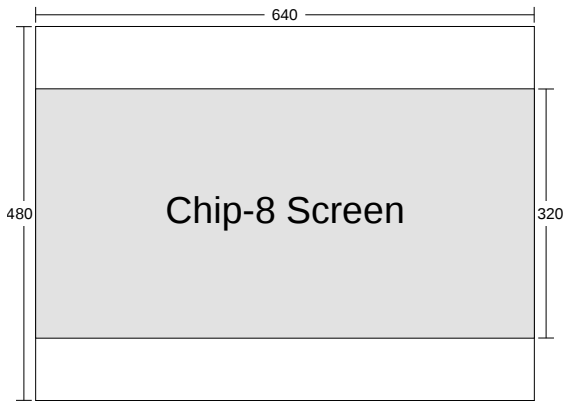

Figure 3.1: Chip-8 screen scaled to VGA screen

The updates to the screen state are controlled by the state machine shown on Figure 3.2.

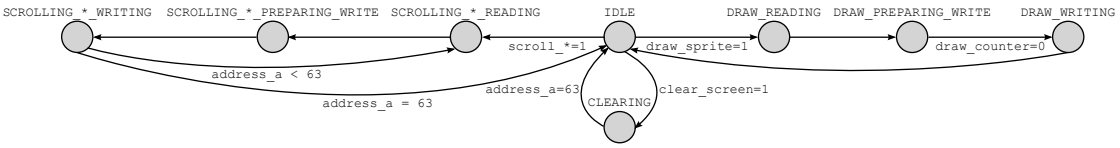

Figure 3.2: Video controller state machine

The screen clear is initiated by putting the signal `clear_screen` on `high`. Then, the address bus from RAM port A is used as a counter from zero to 63, storing zeros on every position of the RAM. After the process is finished, the state machine returns to `IDLE` state.

The the `draw_sprite` signal is `high`, a drawing starts. In the `DRAW_READING` state, the memory reads the current state of the area to be drawn. In the next clock cycle, the state machine enters the `DRAW_PREPARING_WRITE` state, in which it will shift the content to the correct position using a circular shift register, to ensure the wrapping behavior described above. In the last clock cycle in this state the current state is XORed with the new sprite and the resulting pattern is ready to be written in the `DRAW_WRITE` state. Also, the `colision` signal is generated here by ANDing the current state and the sprite. This process performs a drawing on one line of the screen. For multiple-line sprites it's up to the Chip-8 core to repeat the procedure several times.

For each screen scroll direction (left, right and down) there are three states. As they work in a very similar way, they are drawn only once in the picture. During the `SCROLLING_*_READ` state a row on the screen is read. On the `SCROLLING_*_PREPARING_WRITE` the data is shifted to the left or to the right if necessary and, finally, on the `SCROLLING_*_WRITING` the new content is written to the memory. During a scroll down the data is not shifted, it's only written to another row.

4 Memory controller

The external SRAM is used as the Chip-8 memory. Even though each entry in the SRAM available on the Nexys 3 board is 16 bits wide, only 8 bits are used, because the Chip-8 operates on 8-bit values.

The circuit used to interface with the SRAM is exactly the same memory controller developed for Project 2 and won't be detailed in this report.

5 Input controller

The input signals are generated by the Microblaze, so this component merely uses a decoder to select the pressed key with the lowest value. It's useful to implement the `Fx0A` opcode, that waits for a keypress.

Although this input controller is extremely simple, it's kept as a separate component to allow a simple replacement of the input system. It could be possible, for example, to implement an input controller that gets information from a keyboard connected to the USB Host port available on the Nexys 3.

6 Sound controller

Chip-8 can only generate one tone and the frequency is implementation-dependent. As the Nexys 3 does not provide any peripheral that can produce sound, it was implemented using the DAC used in the first project.

The component that interfaces with the DAC is very similar to the one used in Project 1. The component wasn't simply copied from Project 1 because it was written in VHDL, but the logic is the same and won't be discussed here.

Once we have a DAC Controller that can set the DAC output to an arbitrary value, all we have to do is sending the values to the DAC Controller at the correct time. The frequency chosen for this project is 440Hz. It's a note A, also known as A440, and it's usually used to tune instruments (in fact, because of this frequency choice, it's possible to write a Chip-8 tuner using only three instructions!).

Usually, sound is stored using a sample rate of 44.1kHz. Here, we will use 100 samples per cycle of our sound wave, giving us a sample rate of 44kHz. The wave to be produced is a sine, like in the Project 1, but this time it's not possible to list every sample in a table, due to the number of samples. The values were generated using an online generator[6].

This circuit is clocked at 100MHz, thus it needs to send a value to the DAC every $100 \times 10^6 \times \frac{1}{44000} \approx 2272$ clock cycles. It's done using a counter.

To check if the frequency is correct, I wrote a Chip-8 program to produce sound (the three-instruction tuner) and tested it using an online tuner[7]. As shown on Figure 6.1, the DAC produces the frequency we want it to produce with very good precision.

[6] http://www.daycounter.com/Calculators/Sine-Generator-Calculator2.phtml
[7] http://www.seventhstring.com/tuner/tuner.html

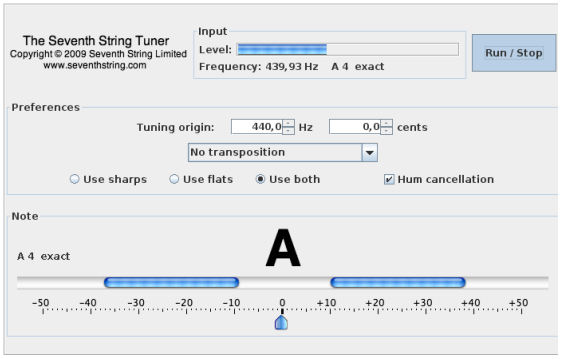

Figure 6.1: A440 produced using the DAC

7 Chip-8 Core

This component implements the Chip-8 instruction set, fetching instructions from the external memory, dealing with the input and drawing sprites to the screen. The implementation does not follow any processor design pattern and there isn't the concern of producing the most optimized implementation, because we don't have strong speed requirements here, so, whenever possible, the code legibility was favored over optimization.

The processor also has some debugging capabilities. When the `debug` signal is `high`, the processor halts after each instruction and waits for a rising edge of the `step` signal to proceed. When the processor is halted, it's possible to read it's internal state.

This circuit is controlled by two state machines. The main state machine controls the current status of the processor, while the communication state machine handles the generation of signals for the other components.

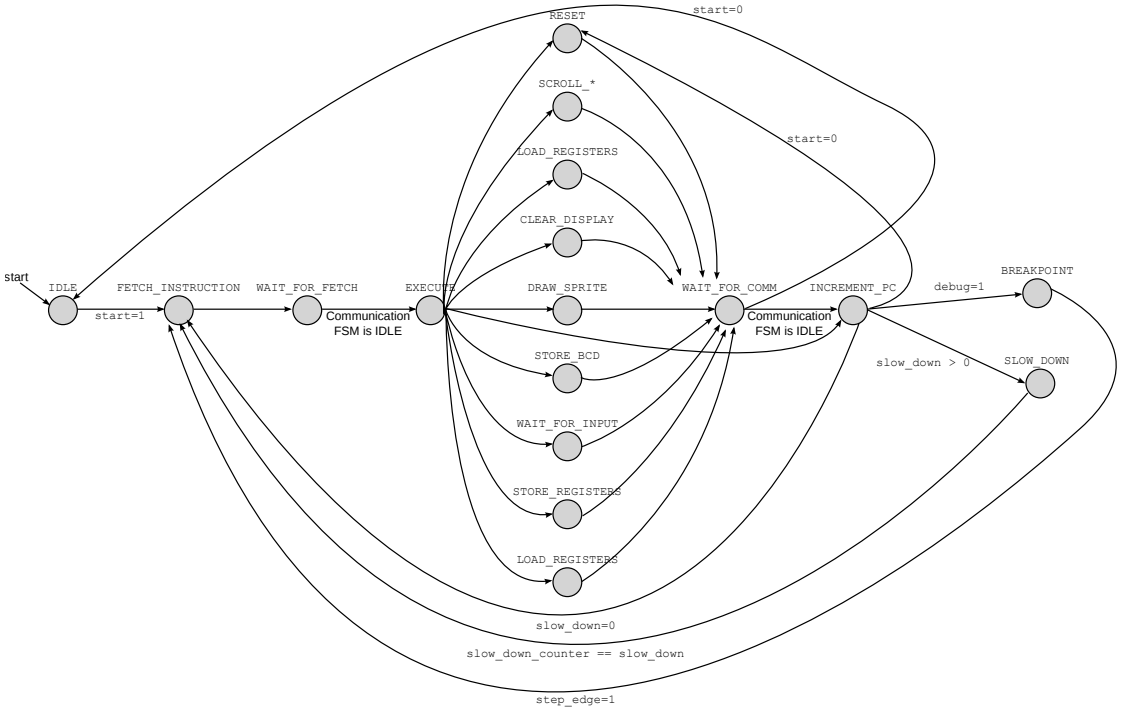

Figure 7.1: Chip-8 main state machine

The main state machine is shown on Figure 7.1. The processor starts on the `IDLE` state and waits for a `start` signal to begin execution. In the `FETCH_INSTRUCTION` state the fetch of a instruction starts and the processor waits for the end of the fetch on the `WAIT_FOR_FETCH` state. On the `EXECUTE` state the instruction is parsed and the state machine goes to the corresponding state. For instructions that can be executed in one clock cycle the state machine goes to the `INCREMENT_PC` state directly. Otherwise, it waits in the `WAIT_FOR_COMM` state for the communication state machine to performe the necessary operations. In the `INCREMENT_PC` state the program counter is incremented, sometimes by 2 and sometimes by 4 (when there is a skip). If the processor is in debug mode or if the slow down is enabled, the machine goes to the corresponding states and waits for the condition to proceed. If the `start` signal becomes `low`, the machine goes to the `RESET` state to restore the processor's initial state and allow a new program to be loaded, without the need to physically reset the board. The `SLOW_DOWN` state counts a certain number of clock cycles before proceeding to the next instruction. This number is determined by one of the inputs of the component and can have up to 16 bits. The `BREAKPOINT` state waits for a rising edge of the `step` signal. The edge is detected by the same technique used in previous projects, where the signal is connected to a two-bit shift-register and the filtered signal is high whenever the shift-register contains the value `10`.

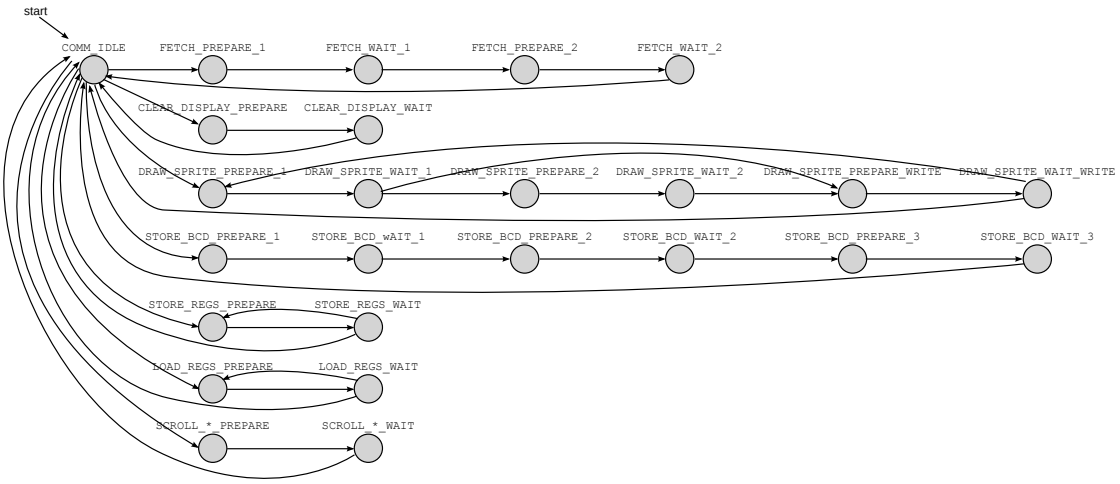

Figure 7.2: Communication state machine (the conditions for the transitions are not shown to keep the picture a little cleaner)

The communication state machine is shown on Figure 7.2. The procedure is very similar in all cases, with a prepare state to prepare the data and send to the video or memory controller and a wait state to wait fo the controller to finish its job. The fetch is composed by two groups of a prepare and a wait state, each group corresponding to one byte of the instruction, because each instruction is 16 bits large. The clear is composed by two states that simply ask the video controller to clear the screen and wait until it's done. The draw operation has two groups of prepare and wait states used to load sprites from memory. The second group of states is only used by the S-Chip 16x16 sprites to load the second half of the data. Then, there is a prepare and a wait states that ask the video controller to actually write the data to the screen. This procedure is repeated for every line of the sprite. The storage of a BCD number is done in three groups of a prepare and a waits state. Each group is responsible for the storage of one of the digits of the BCD number. Loading and storing registers use one prepare and one wait state to communicate with the memory controller and the procedure is repeated for each register that needs to be loaded. Finally, there are one prepare and one wait state for each direction of scroll (up, left and down) that simply ask the video controller to perform the scroll.

These two state machines perform most of the work inside the Chip-8 core, but there are also some other important subcomponents that are discussed in the following subsections.

7.1 Random generator

Chip-8 provides a function that generates random numbers. It's a known fact that computers and digital circuits in general are really bad at generating random data, so a pseudorandom number generator (PRNG) is commonly used when some randomness is necessary.

One of the most simple PRNGs available is the Linear Feedback Shift Register (LFSR). It's implemented as a shift register whose input bit is calculated by XORing some bits of the current state. This kind of PRNG can't be used in cases where good randomness is necessary, such as in criptography, because of its linearity, but for our purposes it's good enough.

The LFSR needs an initial value because whenever it reaches a state where all bits are zero, it will never leave this state. The LFSR used in this project and the initial value are shown on Figure 7.3.

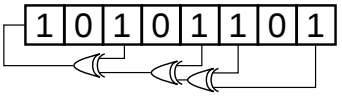

Figure 7.3: Linear Feedback Shift Register

7.2 Timers

Chip-8 has two timers known as delay and sound timers. Both timers decrement at a rate of 60Hz whenever their values are different from zero.

The timer component uses a counter that counts from zero to 1,666,666. As this circuit is clocked at 100MHz, this counter reaches its maximum value once every $\frac{1}{60}$ seconds and when this value is reached the counters are decremented if necessary. This is the technique explained in class for generating slower clocks.

7.3 BCD Encoder

This component is used by the `Fx33` opcode, that stores the BCD value of a register on memory. It's simple combinational logic that takes a binary input and generates three outputs, one for each decimal digit.

8 Microblaze

The Microblaze microcontroller is responsible for every interaction of the user with the system. The interaction is done through the serial port.

To allow the user to perform various actions, a very simple protocol was implemented. In this protocol, the user indicates the action he or she wants to perform by sending one byte with the action code. If the command requires some data from the user, it's sent after the action code. Then, the Microblaze performs the requested action, sends some data to the user when necessary and then sends back to the user the action code to indicate that everything worked as expected. In this protocol, every action is initiated by

the user, never by the Microblaze, so all the Microblaze has to do is to listen to the serial port and wait for some command.

The available commands are described below:

- **Echo (`0x00`):** the user sends five bytes to the Microblaze and it echoes back the same bytes. It can be used to check if the connection is working properly.
- **Load game (`0x01`):** the user sends two bytes indicating the size of the game in bytes and then sends the game data itself. The Microblaze loads that game into the SRAM starting from position `0x200` and also loads the font data to the lower positions of the memory.
- **Set debug mode (`0x02`):** the user sends one byte that can be `0` or `1` indicating if the debug mode should be enabled or disabled. The Microblaze sets the `debug` signal accordingly.
- **Start processor (`0x03`):** the Microblaze sets the `start` signal to `high`, causing the processor to begin execution.
- **Stop processor (`0x04`):** the Microblaze sets the `start` signal to `low`, causing the processor to stop execution and reset to its initial state.
- **Read registers (`0x05`):** the Microblaze sends 16 bytes containing the current value of the registers `V0` to `VF`.
- **Read program counter (`0x06`):** the Microblaze sends 2 bytes containing the current value of the program counter.
- **Read instruction register (`0x07`):** the Microblaze sends 2 bytes containing the current value of the instruction register (the last executed instruction).
- **Read I register (`0x08`):** the Microblaze sends 2 bytes containing the current value of the I register.
- **Key press (`0x09`):** the user sends one byte containing a value from `0x0` to `0xF` indicating what key should be pressed. The Microblaze holds this key pressed until a key release command is sent.
- **Key release (`0x0A`):** the user sends one byte containing a value from `0x0` to `0xF` indicating what key should be released.
- **Set slow down (`0x0B`):** the user sends two bytes containing the value that will be the maximum value for the slowdown counter.
- **Step (`0x0C`):** the Microblaze generates a rising edge on the `step` signal, causing the processor to proceed to the next instruction.

- **Wait for processor to halt (`0x0D`):** the Microblaze checks the `stopped` signal of the Chip-8 core some times to see if the processor halted. If after a certain number of tries the processor do not halt, the Microblaze sends a byte containing the value `0`, otherwise it sends the value `1`. This is useful to make sure the processor is not executing some instruction before asking for a register dump using the `0x05` command.

With this protocol and a reasonably good program for serial communication it's possible to load a game, start the processor and step instruction by instruction if necessary, but it's impossible to play any game sending press and release commands for every key. To provide a way of actually playing games, a Java application was written. The binary data of each ROM is stored in a ZIP file along with an icon, an image of the game and a JSON[8] file that contains some metadata about the game. The metadata consists of the name of the game, the author, a description, a set of instructions, the amount of slowdown the game needs to be playable and a key mapping that associates the keys on the hexadecimal Chip-8 keyboard to the keys in a standard computer keyboard. The application can read all ZIP files in a folder and show a list of the available games to the user. The user can, then, select which game he or she wants to play. The application uses the protocol described above to load the game into the system and then starts listening to the computer keyboard. By using the key mapping metadata, the application can send the correct press and release commands to the Microblaze whenever a mapped key is pressed. This per-game mapping is good because it allows various games to be mapped to the arrow keys, even when they use different Chip-8 keys as directional controls.

The application is shown on Figure 8.1. The implementation details are not discussed here, as it's not in the scope of the course.

[8] http://json.org/

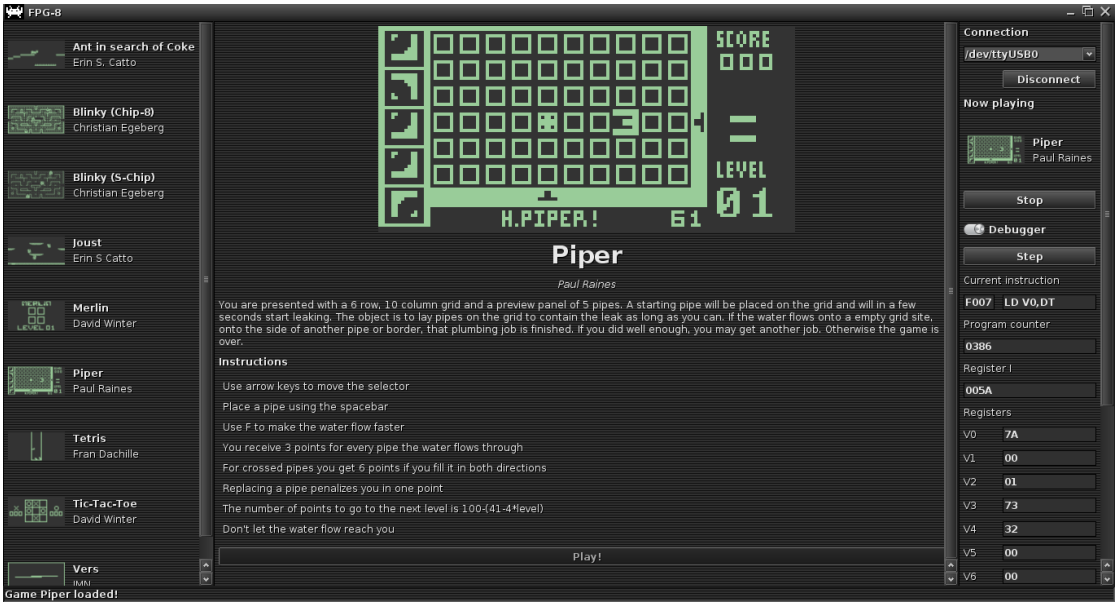

Figure 8.1: Java application to control the Chip-8

9 Combining all the modules

Figure 9.1 shows show the modules are connected.

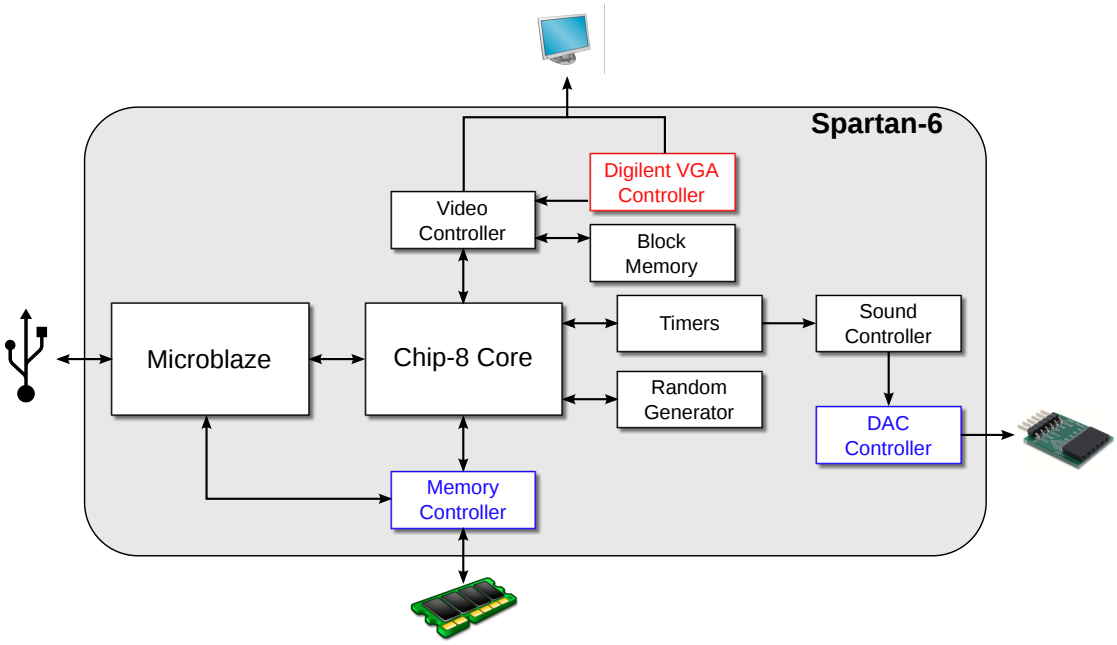

Figure 9.1: Top-level module of the project

Most components can be directly connected to each other. The only case that is worth mentioning is the connection to the memory controller, because we have two circuits writing to it.

To connect both the Microblaze and the Chip-8 core to the memory, a simple combinational circuit selects which input should be connected based on the current state of the processor. Whenever the processor is in the `IDLE` state the Microblaze drives the memory, otherwise the Chip-8 does that.

As in the Microblaze we don't have timing precision in the clock cycle level, we also have to make sure that the `write` signal generated by the Microblaze is not too long to avoid triggering more than one write operation. This is done as in other parts of the project, by connecting the `write` signal to a 2 bits shift-register. The `write` signal that is actually sent to the memory controller is obtained combinationally by checking when the value in the shift register is `01`, which makes sure that the signal is only one cycle long.

10 Simulation

Testing a processor, even a simple one like Chip-8, can be very tricky. Running programs (or games in this case) and visually comparing with other implementations is not enough to make sure everything is correct, because we can't be sure that the programs we are running make use of every available instruction.

However, in this project our concept of "correct" is slightly different from the previous projects. Executing a program consists basically of reading instructions and updating the current state of the processor accordingly, so we are not interested in every single waveform that is generated inside the circuit, as long as the state of the processor is correct after each instruction. So, to test this implementation and make use of the new techniques learned in class that use files within the test fixtures, we will only look at the state of the processor after each instruction and print messages on the screen, instead of analysing waveforms.

The nice thing about Chip-8 is that it's very easy to find open-source implementations for any platform. So, instead of generating the expected states by hand, I used one of those implementations to do that.

First, I downloaded the Chip-8 implementation in Java by Michael Arnauts[9] and then modified it to print to a file the current state of every register in the processor after each instruction. This file is generated in a format that can be easily read by the `readmemh` command in Verilog.

Then, I wrote a small Chip-8 code to test some functions, specially some corner cases like the subtraction function that most games probably wouldn't use. This code was assembled using the Mochi-8 Assembler[10] and ran on the Michael Arnauts' emulator.

After that, the assembled code was converted to a hexadecimal format that can be read by Verilog using the Linux command line utility `xxd`[11]. This hexadecimal version of the code was loaded into the very simple SRAM model provided by the professor and then the execution was simulated. After each instruction the current state of the processor is compared to the state that the Michael Arnauts' emulator was after the execution of the same instruction and a message is printed to the screen.

This procedure is shown on Figure 10.1 and the Chip-8 assembly code is attached to this report.

[9] https://github.com/michaelarnauts/chip8-java
[10] http://mochi8.weebly.com/
[11] http://linuxcommand.org/man_pages/xxd1.html

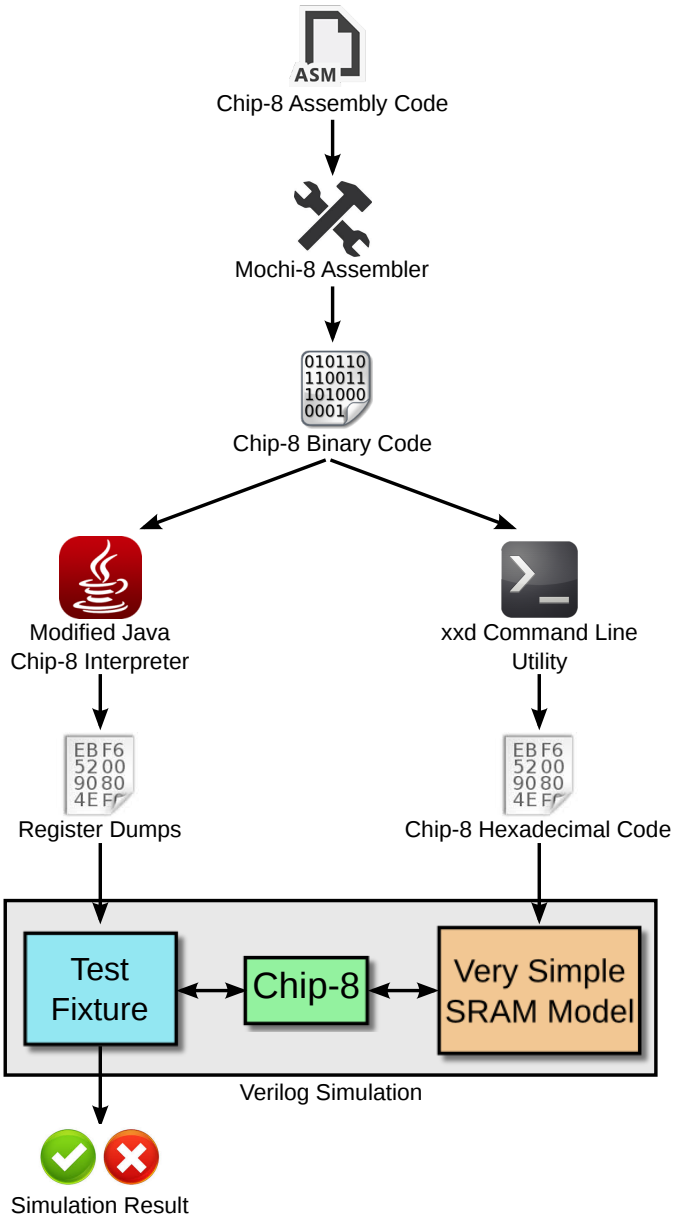

Figure 10.1: Procedure for testing the processor

Not every instruction was tested. The idea was only to try this testing procedure and see if it works while keeping things simple. The procedure proved to be efficient because once we have everything set up it's possible to test any arbitrary combination of instructions with little effort. Examples of instructions that weren't tested are functions that rely on the timers (it's difficult to keep any kind of timing accuracy between two totally different systems), input functions (some more work is necessary to build a convenient way to simulate the inputs on the Java and Verilog sides) and drawing functions.

11 Resource usage

Table 11.1 summarizes the relevant information abount the resource usage of this project.

Resource	Used	Available	Utilization
Slice registers	1625	18,224	8%
Lookup tables	3767	9,112	41%
Block RAM	20	32	62%

Table 11.1: Resource usage for this project

The usage of lookup tables increased significantly compared to the previous projects, which is understandable, because the amount of logic required for this project is much larger.

From the slice registers, 1018 were used in the circuits presented in this report and are explained on Table 11.2. The other flip-flops are used inside the IP cores (Microblaze and block RAM).

Signal	Number of flip-flops
Top level block	
shift_reg	2
Video Controller	
address_a	6
current_row	6
new_row	128
data_input_a	128
draw_counter	7
current_col	7
scroll_amount	4
pixel_counter_vertical	4
pixel_counter_horizontal	4
current_state	4
colision	1
write_enable_a	1
VGA Controller	
hcounter	11
vcounter	11
hs	1
vs	1
blank	1
Memory controller	
mem_address	26
counter	3
data_reg	16
current_state	2
Sound Controller	
current_value	7
counter	12
DAC Controller	
counter	4
shift_reg	16
current_state	2

Signal	Number of flip-flops
Chip-8 Core	
i_reg	16
mem_write_data	16
slow_down_counter	16
stack_pointer	4
scroll_down_amount	4
sprite_height	4
current_reg	4
stack	256
timer_data	8
comm_current_state	5
mem_address	26
draw_row	6
draw_col	7
shift_reg	2
current_state	5
program_counter	16
registers	128
set_delay_timer	1
set_sound_timer	1
mem_read	1
mem_write	1
instruction_register	16
sprite	16
draw_sprite	1
clear_screen	1
scroll_left	1
scroll_right	1
scroll_down	1
colision_detected	1
extended_video_mode	1
Random Generator	
register	1
Timers	
delay_timer	8
sound_timer	8
counter	21
Total	**1018**

Table 11.2: Flip-flops and explanations

The synthesis of this project produces 59 warnings. 46 of there warnings are due to the Microblaze and they are all in the following format:

```
NgdBuild:440 - FF primitive 'mcs_0/U0/microblaze_I/
   MicroBlaze_Core_I/Area.Data_Flow_I/Result_Mux_I/Using_FPGA.
   Result_Mux_Bits[15].Result_Mux_Bit_I/EX_Result_DFF' has
   unconnected output pin
```

There are also two warnings due to the DCM in the format:

```
HDLCompiler:1127 - "/path/to/Project3/ipcore_dir/dcm_25.v" Line
   132: Assignment to status_int ignored, since the identifier
   is never used
```

The block memory generates the following warning:

```
HDLCompiler:1499 - "/path/to/Project3/ipcore_dir/video_memory.v"
    Line 39: Empty module <video_memory> remains a black box.
```

Because we don't use the `busy` signal of the DAC controller, the sound controller generates a warning. It could be removed by removing the `busy` signal from the DAC controller, but it wasn't done because later this module can be reused and this signal can be useful.

```
HDLCompiler:1127 - "/path/to/Project3/sound_controller.v" Line
   44: Assignment to busy ignored, since the identifier is
   never used
```

Because we don't use the upper bits of the memory, we have this warning:

```
Xst:647 - Input <mem_read_data<15:8>> is never used. This port
   will be preserved and left unconnected if it belongs to a
   top-level block or it belongs to a sub-block and the
   hierarchy of this sub-block is preserved.
```

And also this warning:

```
Xst:1710 - FF/Latch <mem_write_data_8> (without init value) has
   a constant value of 0 in block <chip8_core>. This FF/Latch
   will be trimmed during the optimization process.
```

And 7 warnings in the following format:

```
Xst:1895 - Due to other FF/Latch trimming, FF/Latch <
   mem_write_data_9> (without init value) has a constant value
   of 0 in block <chip8_core>. This FF/Latch will be trimmed
   during the optimization process.
```

12 Conclusions

This was a very challenging project that involved most of the concepts that were introduced during the course.

After Project 2 I was more used to the Verilog syntax and it wasn't a big problem anymore. The challenging part here was the implementation of the Chip-8 core. The Video Controller was tricky, and required the use of block RAM, but each functionality could be easily tested and debugged individually. This wasn't true for the Core, as it wasn't possible to see if it could actually execute some code before blindly implementing more than 40 opcodes. Most opcodes were implemented correctly, but some of them required some debugging. At that time the use of files for testing wasn't introduced yet, so the solution was to implement a debugger in the Core, which is a very interesting extra feature that even some software implementations of Chip-8 do not provide.

The S-Chip extension also wasn't initially in the scope of the project, but it is worth the extra work, as most of the very good games rely on it.

This project can be improved in many ways. Some of them are listed below:

- The project needs a lot of optimizations. The one I consider the most important is the use of the block memory. The Core state machines could also be refactored to become simpler;
- Other input systems can be developed. The first idea was to use the USB Host port available on the Nexys 3 to interface with a keyboard, but the professor suggested the use of the serial port instead, for simplification purposes;
- Even though there are only a few games written for the MegaChip extension, implementing it would be a great challenge, as it adds colors and sound.

The full source code of the project was open-sourced and is available on GitHub[12]. This project required weeks of hard work to reach its current state, but in the end it's really rewarding to be able to execute real games written by other people. It's definitely something I'm proud of.

[12] https://github.com/guimeira/fpg8