# CHIP-8 Machine

Jacob Fustos, Cesar Avalos B.

May 11, 2017

## 1 INTRODUCTION

In the early 1970's Joseph A. Weisbecker, a microprocessor researcher and toy enthusiast designed the legendary 8-bit RCA 1802 microprocessor. A few years after RCA released the chip, they released the COSMAC VIP, a computer aimed at hobbyists, with the main purpose of playing video games. Joseph A. Weisback is also credited with creating an interpreted language included with the computer, this language is called CHIP-8. The design philosphy behind CHIP-8 is to create a very easy language to create video games.

This CHIP-8 language had a resurgence with the advent of more powerful graphing calculators, with the HP-48 and TI-89 as examples. Hobbyists regarded the CHIP-8 language as a first major test of their BASIC programming abilities. In the emulation development community nowadays CHIP-8 is again recommended as a first step towards more serious hardware emulation.

The idea of this project is to create an FPGA implementation of the CHIP-8, using only the opcodes and existing games as guides, and eventually run games on the board.

The final code is hosted here: https://github.com/cesar-avalos3/C8VHDL.

### 1.1 CHIP-8 SPECS

As previously stated, the CHIP-8 programming language was designed to be simple, as such its specs are modest. It's an 8-bit cpu, it has 16 8-bit data registers, named V0 to VF, an address register called I, involved in many opcode operations, and finally the stack register called the SP, to keep track of the memory addresses when calling subroutines. The machine makes use of 35 opcodes, all 2 bytes long and big-endian.

## 1.2 Opcode Table

| Opcode | Description |
|---|---|
| 0NNN | Execute subroutine at address NNN |
| 00E0 | Clear Screen |
| 00EE | Return from subroutine |
| 1NNN | Jump to address NNN |
| 2NNN | Execute subroutine starting at address NNN |
| 3XNN | If Vx == NN, skip the next instruction |
| 4XNN | If Vx != NN, skip the next instruction |
| 5XY0 | If Vx == Vy, skip the next instruction |
| 6XNN | Store NN in Vx |
| 7XNN | Add NN to Vx |
| 8XY0 | Store Vy in Vx |
| 8XY1 | Set Vx to Vx OR Vy |
| 8XY2 | Set Vx to Vx AND Vy |
| 8XY3 | Set Vx to Vx XOR Vy |
| 8XY4 | Store in Vx, Vx + Vy |
| 8XY5 | Subtract Vy from Vx, store in Vx |
| 8XY6 | Store in Vx, Vx » Vy |
| 8XY7 | Store in Vx, Vy - Vx |
| 8XYE | Store in Vx, Vx « Vy |
| 9XY0 | If Vx != Vy, skip the next instruction |
| ANNN | Store memory address NNN in Vi |
| BNNN | Jump to address NNN + V0 |
| CXNN | Store in Vx a random number |
| DXYN | Draw Sprite in at position Vx, Vy with n bytes of sprite data |
| EX9E | If key Vx is pressed, skip the next instruction |
| EXA1 | If key Vx is not pressed, skip the next instruction |
| FX07 | Store current delayTimer in Vx |
| FX0A | Wait for keypress, store result in Vx |
| FX15 | Set delayTimer to Vx |
| FX18 | Set soundTimer to Vx |
| FX1E | Add Vx to register I |
| FX29 | Set I to the memory address of sprite of hex digit in Vx (font) |
| FX33 | BCD |
| FX55 | Store V0 to Vx inclusive to memory location starting at address I |
| FX65 | Fill V0 to Vx inclusive from memory location starting at address I |

# 2 Operation

All modules share a common, more advanced, feature. This feature is the use of a return state register. In this way, certain states were able to be reused over and over again without the need for complicated logic. The example below shows a memory access. Many of the modules use this feature and use it in multiple places. As an example, the draw sprite instruction inside of the CPU needs to access memory 5 times per scan line. This is separate from the memory access that the fetchs do, and from the accesses that other instructions do. To save on states we created argument registers that a state may use to set up the memory access. It then sets a return state register with the state that it would like to return to. It then jumps into the memory access sequence. Once the sequence has completed control is returned to the desired state, and the result of the memory access is contained in the returned data register. Using this pattern, multiple state sequences are able to access memory without a complicated logic block for returning from memory.

## 2.1 Core

This implementation consists of 3 main components, first of which is the core. The core, as the name suggests, is the brains of the machine. Among its responsabilities:

- Keeping track of the registers.
- Telling the memory controller when and more importantly where to fetch information from
- Decoding the instructions from the previously fetched word. The decoding module is synthesized as a large mux.
- Executing the decoded instructions, from adding numbers to drawing sprites. Look the table in the previous page to see all the available operations.

## 2.2 Memory Controller

The memory module is at the heart of the system. It contains 2 inner memory modules, the RAM and ROM modules. The ROM module is at addresses 0x00000 through 0x0FFFF and contains the static information for each game. On reset, the memory module copies the ROM area corresponding to the game select lines into the RAM area. The RAM module contains a low area, 0x10000 through 0x10FFF, this area contains static information that is common to all games, and is also copied into the main memory on start-up. Addresses 0x11000 through 0x11FFF are the main memory. This section of RAM is what all other components in the system have access to through a 12 bit address line with the memory module. Once the initial copying is done, the memory module simply then acts as a 3 port arbiter to the memory for the other modules. Each module has its own hold line that it can set high. Once this is set that module will own memory until it sets its line low again. This feature is used by both the time keeper to do a read-modify-write access that needs to be atomic, and by the VGA controller, so that it can do its reads without being interrupted.

## 2.3 Audio

Audio was a funny thing, the Nexys-4 board doesn't actually have a whole lot of documentation on how to use its sound components, first we had no idea the board could generate sound, so the sound related opcode (FX18) was just turning on and off an LED. Once we figure the board could generate sound, simply switched the LED with the amp enable pin on the board, and sound was produced.

## 2.4 GAMES

The games are pretty simple for today's standard, we have games representing at least 4 decades, "Kaleidoscope" for instance is one of the games that shipped with the original CHIP-8 implementation in 1977, "Vers" from the late 80's,"Tetris" from the early 90's and "Brix" from the early 2000's. Some games require the player to keep the score and reset when done.
The whole list of games we include, and their respective authors is as follows:

- PONG by Paul Vervalin
- Tetris by Fran Dachille
- Blitz by David Winter
- Brix by Andre Gustafsson
- Cave by 199x
- Hidden by David Winter
- Kaleidoscope by Joseph Weisbecker (RCA)
- Merlin by David Winter
- Missile by David Winter
- Puzzle by Joseph Weisbecker (RCA)
- Tank by Joseph Weisbecker (RCA)
- Vers by JMN

# 3 Diagrams

## 3.1 Top Level - Diagram

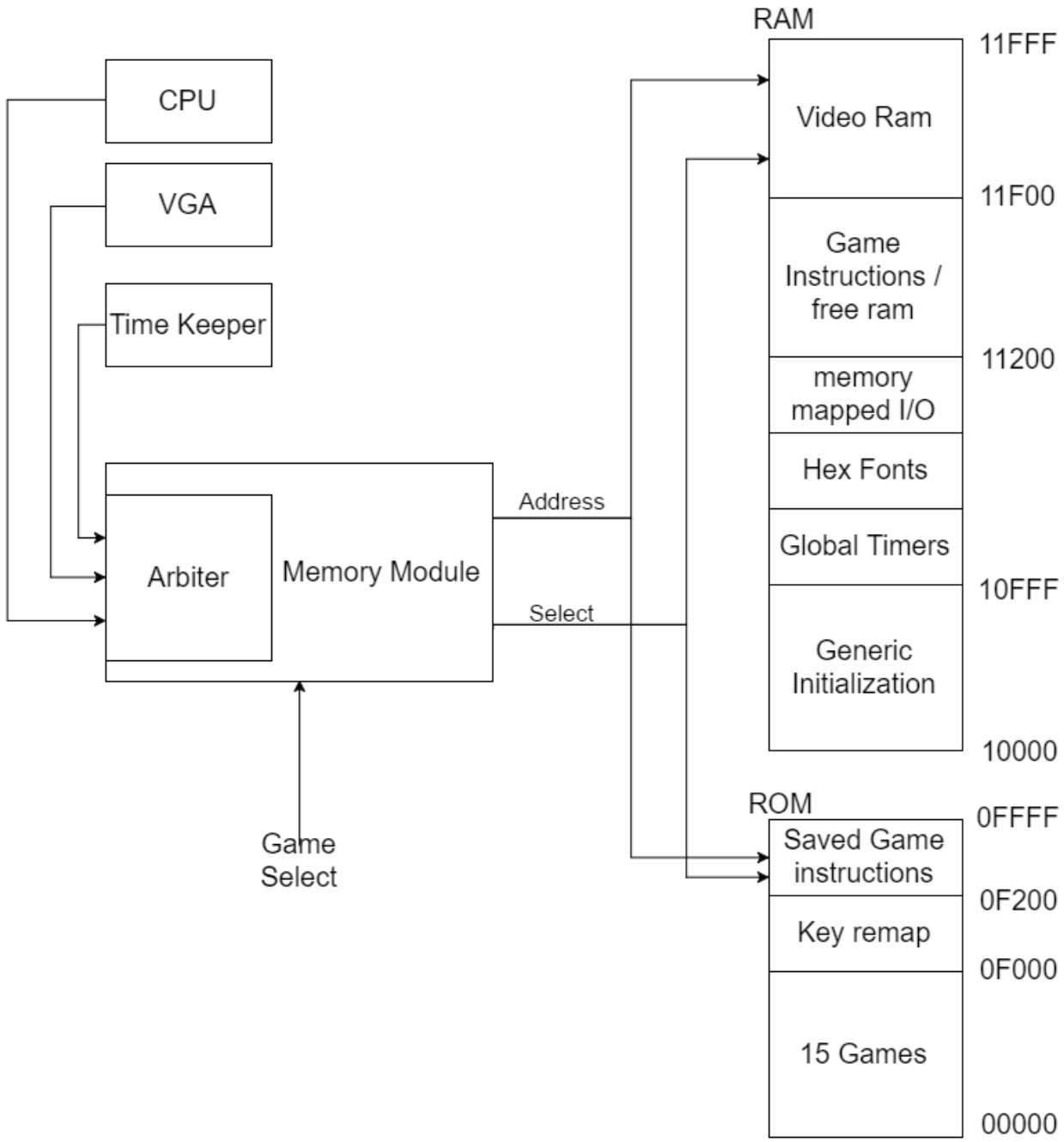


Figure 3.1: A top level diagram of all the components, includes memory mapping.

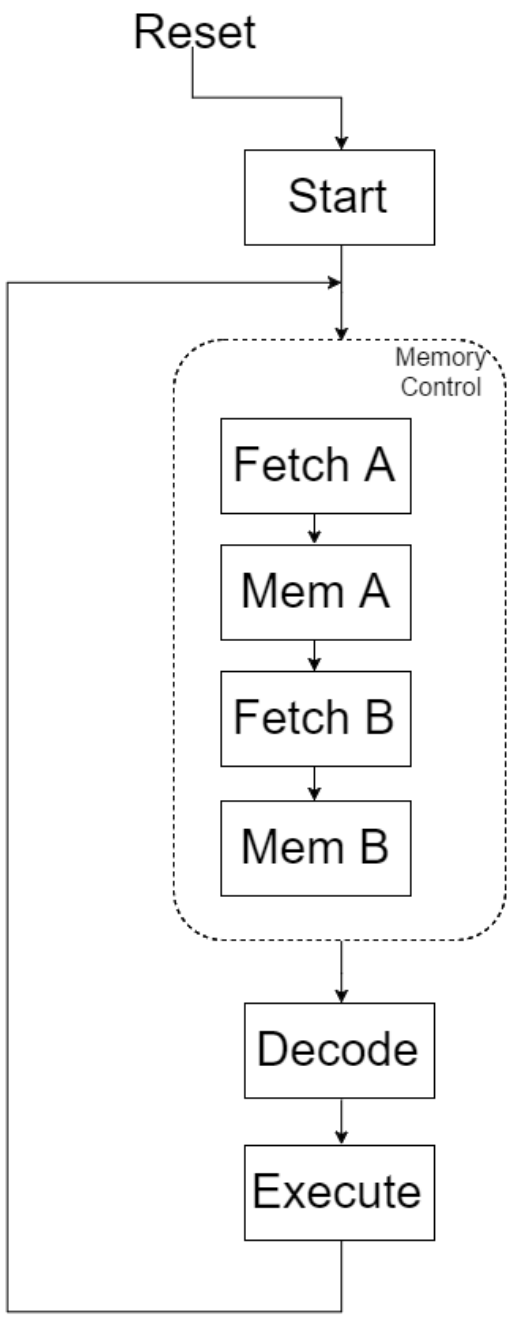


Figure 3.2: The core states diagram, this is a generic representation, because the execute state in this figure could actually be 35 different states, each opcode is represented as a separate state, adding them all to a diagram will be messy.

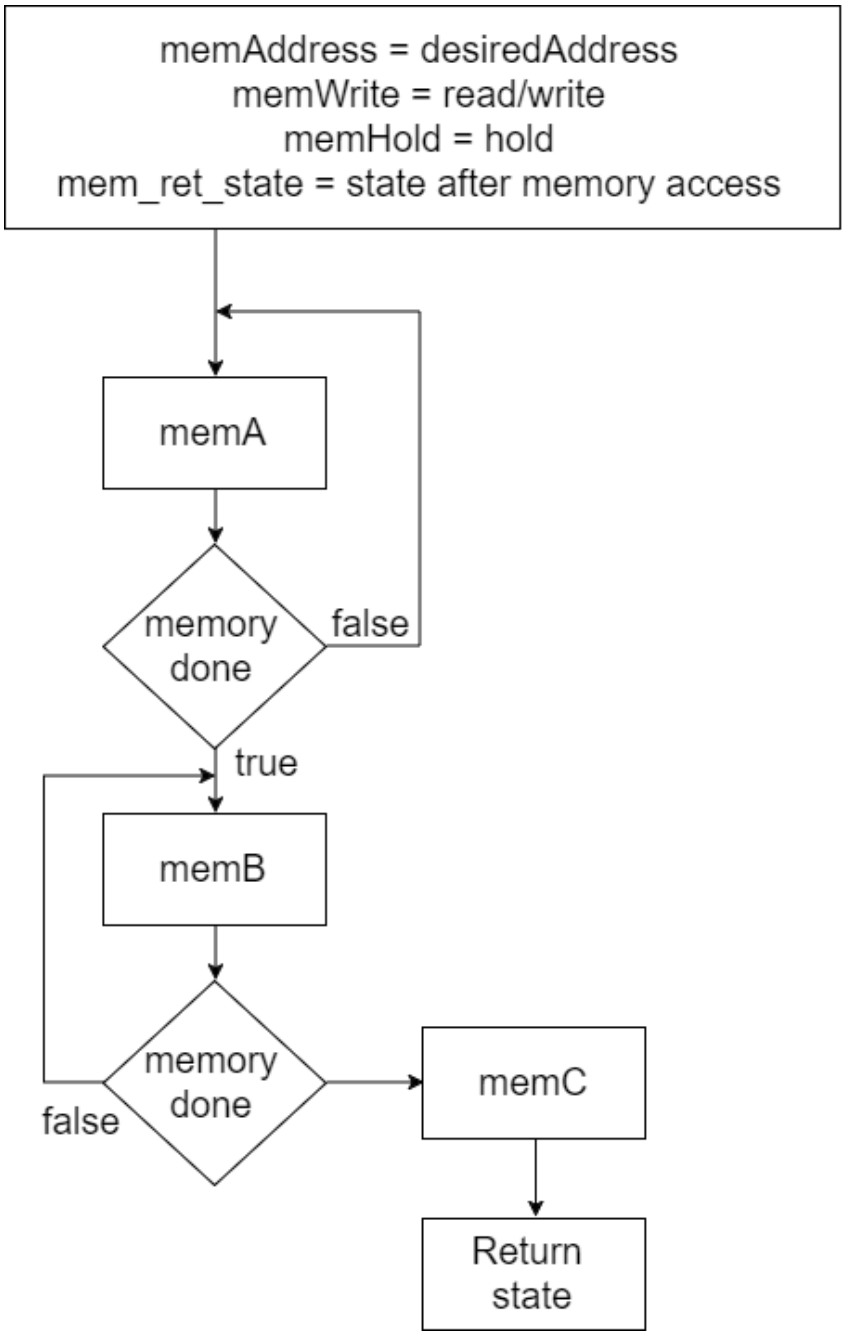


Figure 3.3: The memory controller state diagram

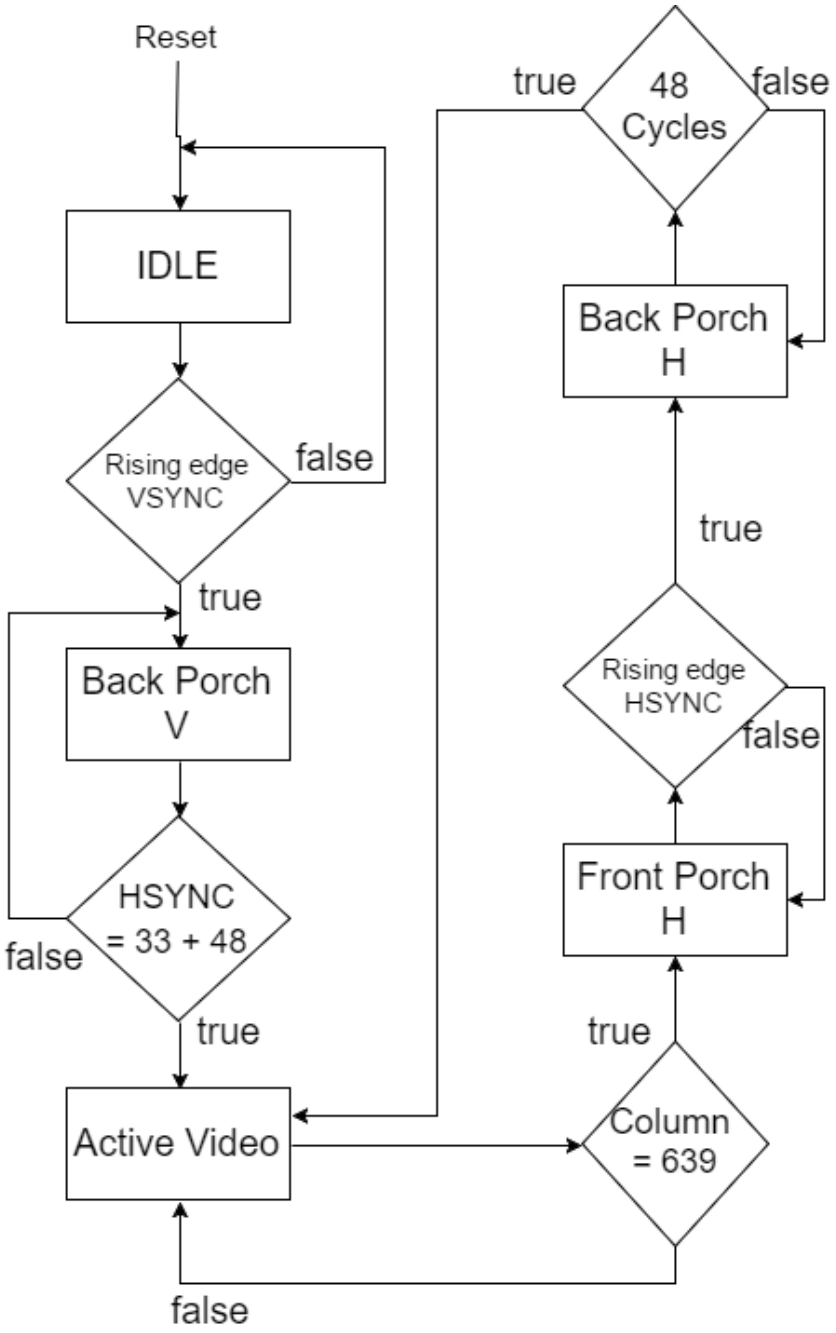


Figure 3.4: The video controller state diagram, notice only shown when the video is ready to draw

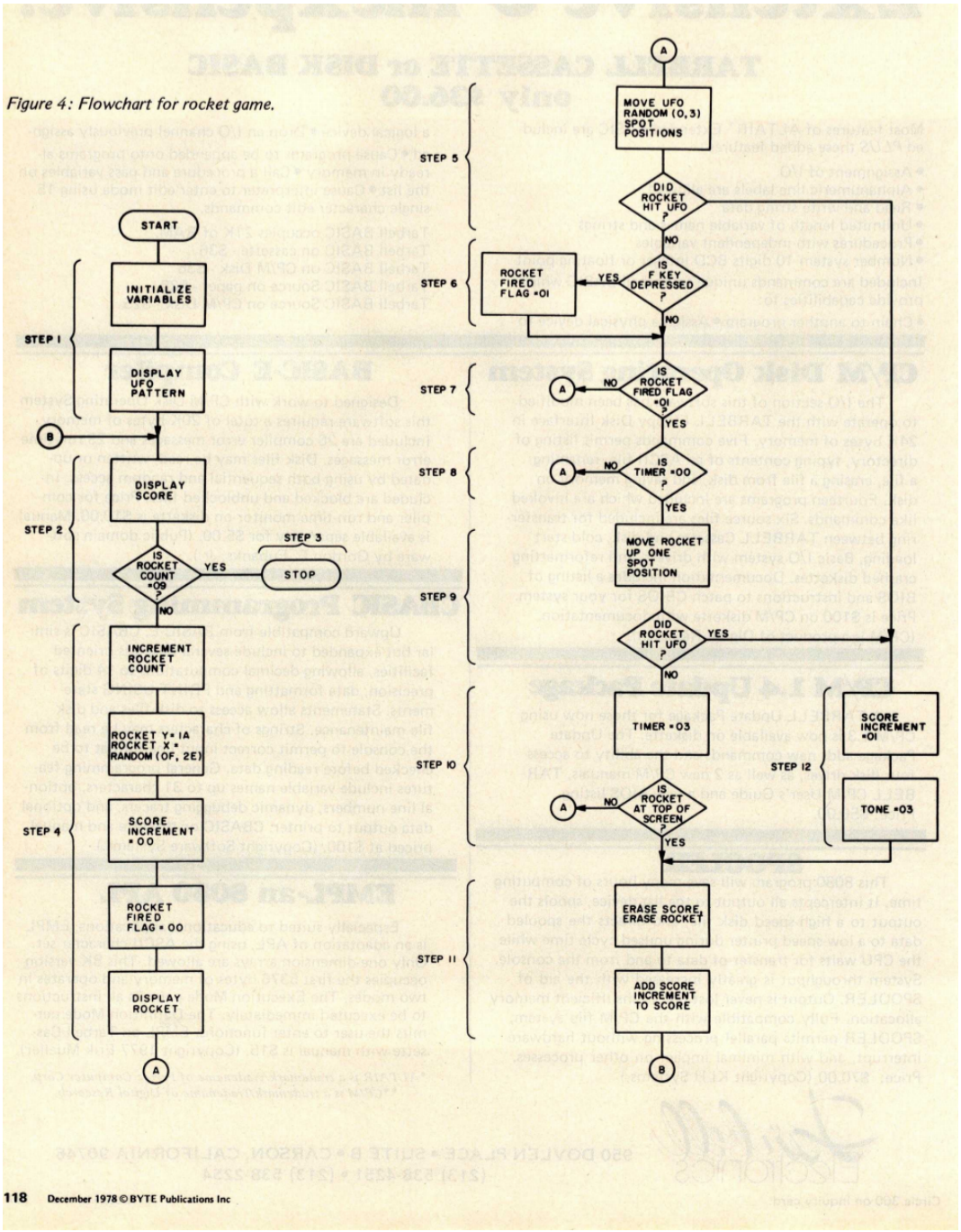


Figure 3.5: Flow chart of the CHIP-8 game rocket by Joseph Weisbacker. Published in BYTE magazine, December 1978 issue.


Figure 3.6: The Brix game, a breakout clone


Figure 3.7: The Merlin game, a simon says clone

# 4 Block Diagram

## 4.1 Top Level

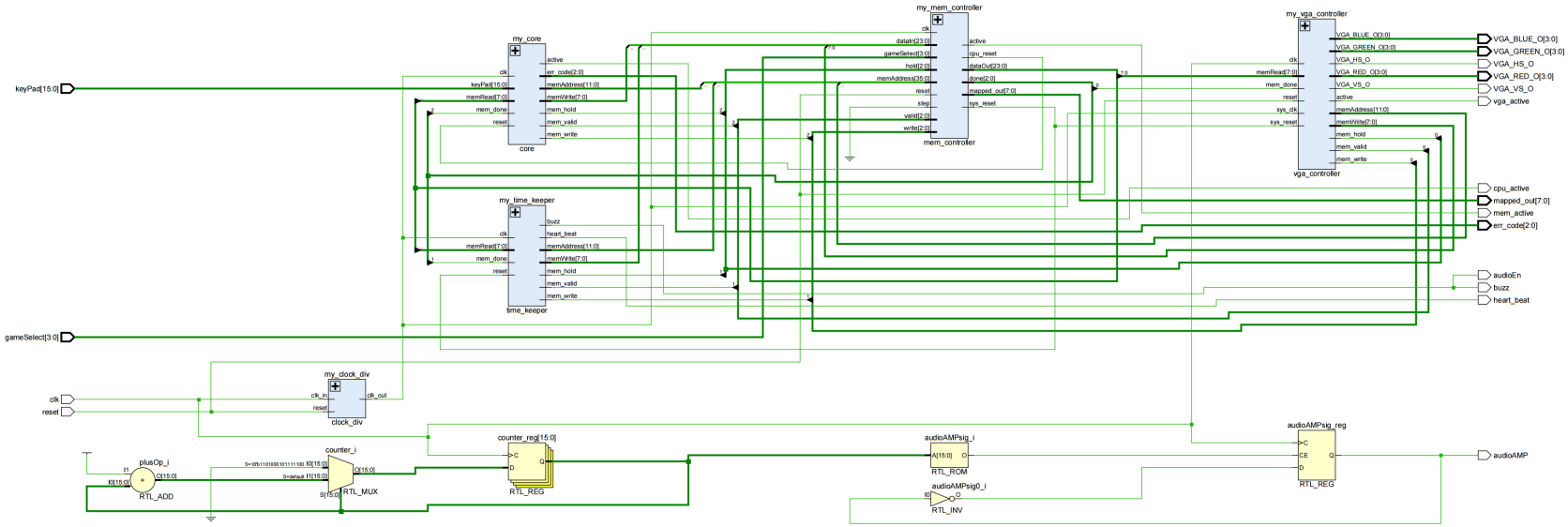


Figure 4.1: Block view of the top level component

## 4.2 Core

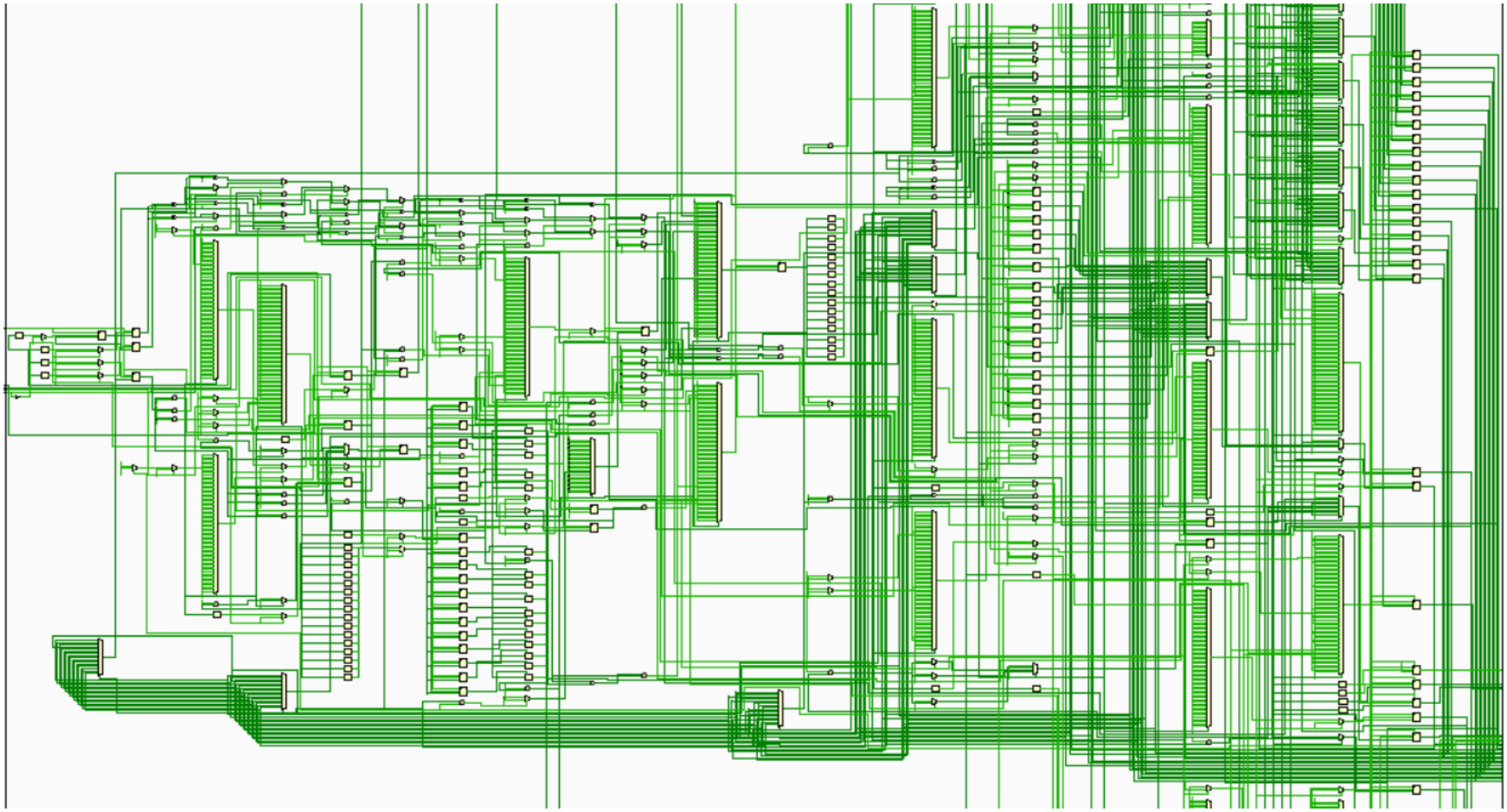

Figure 4.2: Partial Block view of the core component, this thing is monstruous, it's 5 times as tall as it is wide, which makes sense as it needs to have 16 muxes connecting the 16 registers.

## 4.3 Memory Controller

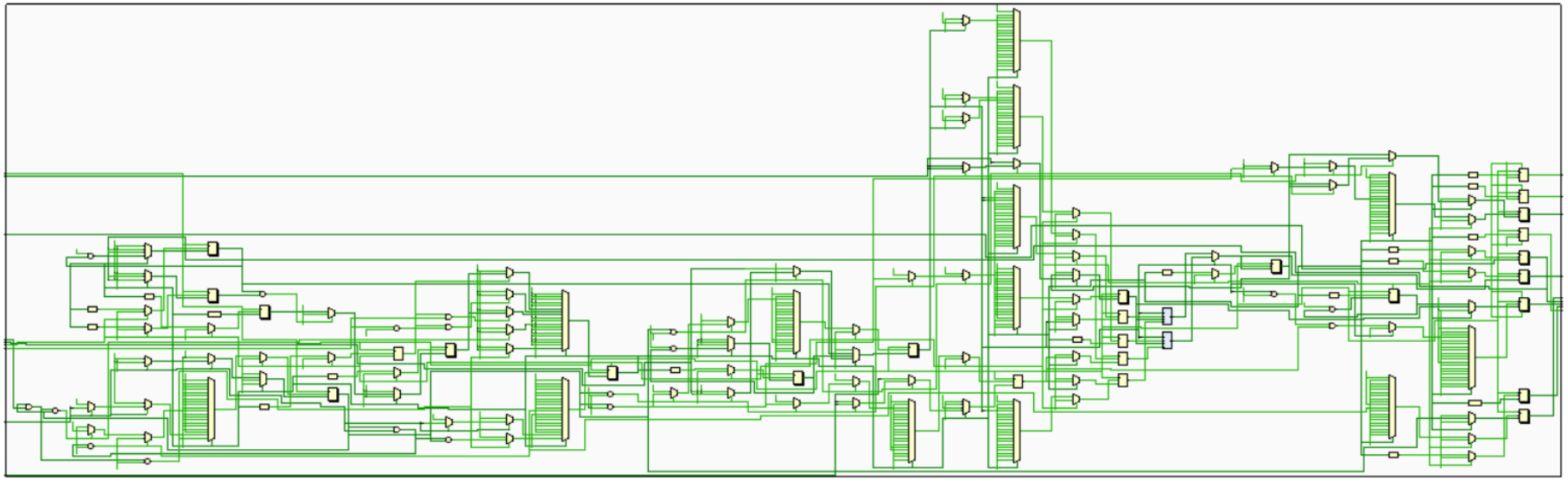

Figure 4.3: Block view of the memory controller

## 4.4 VGA Controller

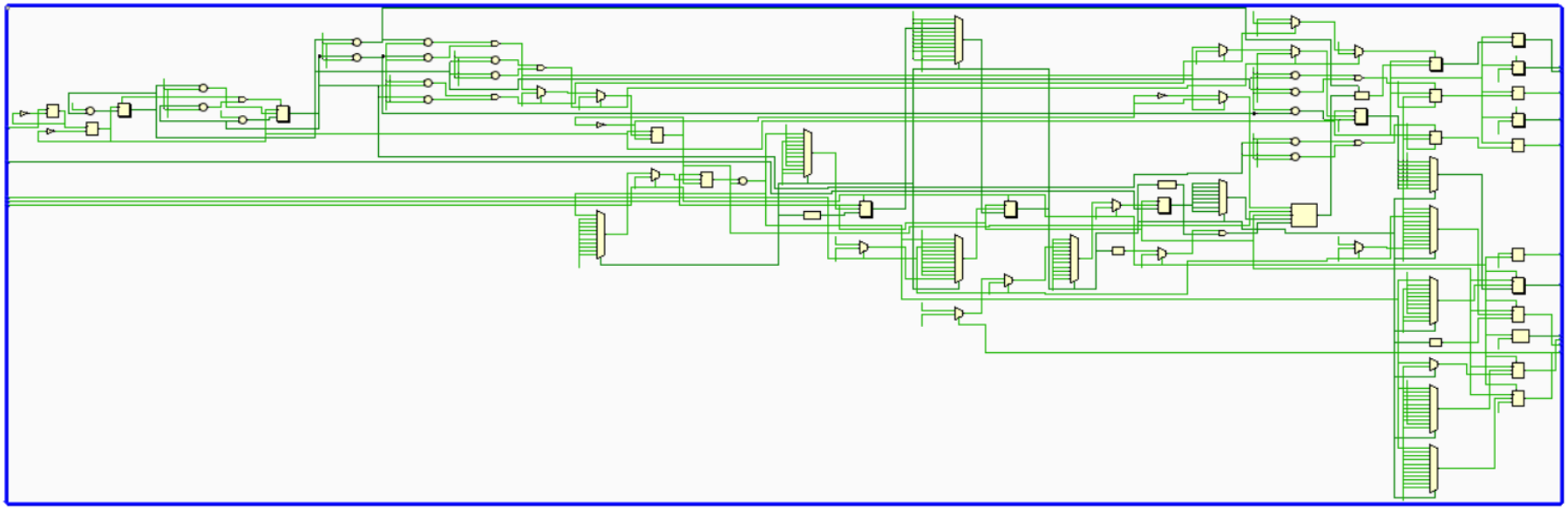

Figure 4.4: Block view of the VGA controller

# 5 Improvements

If we had more time, we would've like to implement a better 2 player solution. We were looking to perhaps add another dedicated controller or perhaps connect the keyboard via the PS/2 interface through the USB port, but there was no really quick way of implementing it, so we put it aside.

The original manual that came with the VIPer computer had a section dedicated to possible expansions one could introduce to the machine. Among those was a board, called the VP-595 Simple Sound Board, costed 30 USD in 1977 (around 125 USD nowadays) and provided a whooping 255 programmable tone frequencies, so one had to not only install the board, but add the opcode to allow the board to work, so this would've been a nice expansion to add to our project.

There was also overhauls to the original system, Super CHIP8 was briefly discussed for a moment in the group. Super CHIP8 allowed higher resolutions, had more memory available, meaning more sprites on screen, even though it's only a couple of opcodes, it would've still added unnecessary complexity to the project, so we decided not to pursue this.

Finally, perhaps simplify the code, there might be a lot of over-engineered aspects in our code that could be further refined, but as it currently is, it's not really taxing on the FPGA board, so it would be really down in our priority list.

# 6 Code

## 6.1 Top Level

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity chip_8A_board is
Port (  keyPad : in STD_LOGIC_VECTOR (15 downto 0);
        gameSelect : in STD_LOGIC_VECTOR (3 downto 0);
        clk, reset : in STD_LOGIC;

        err_code : out STD_LOGIC_VECTOR( 2 downto 0 );
        heart_beat, buzz : out STD_LOGIC;
        cpu_active, mem_active, vga_active : out STD_LOGIC;

        mapped_out : out STD_LOGIC_VECTOR( 7 downto 0 );
        -- step : in STD_LOGIC; -- for debuging memory

        VGA_HS_O : out  STD_LOGIC;
        VGA_VS_O : out  STD_LOGIC;
        VGA_RED_O    : out  STD_LOGIC_VECTOR (3 downto 0);
        VGA_GREEN_O  : out  STD_LOGIC_VECTOR (3 downto 0);
        VGA_BLUE_O   : out  STD_LOGIC_VECTOR (3 downto 0);

        audioEn : out STD_LOGIC;
        audioAMP: out STD_LOGIC


        );
end chip_8A_board;

architecture Behavioral of chip_8A_board is
    component core
        port ( memRead : in STD_LOGIC_VECTOR (7 downto 0);
               memWrite : out STD_LOGIC_VECTOR (7 downto 0);
               memAddress : out STD_LOGIC_VECTOR (11 downto 0);
               keyPad : in STD_LOGIC_VECTOR (15 downto 0);
               mem_valid : out STD_LOGIC;
               mem_write : out STD_LOGIC;
               mem_hold : out STD_LOGIC;
               mem_done : in STD_LOGIC;
               cpu_state : out STD_LOGIC_VECTOR( 7 downto 0 );
               t_PC : out STD_LOGIC_VECTOR ( 11 downto 0 );
               t_I : out STD_LOGIC_VECTOR ( 11 downto 0 );

               t_addr : out STD_LOGIC_VECTOR ( 11 downto 0 );
               t_n, t_x, t_y : out STD_LOGIC_VECTOR ( 3 downto 0 );
               t_kk : out STD_LOGIC_VECTOR ( 7 downto 0 );
```

```
47
48              t_SP : out STD_LOGIC_VECTOR ( 7 downto 0 );
49              t_STACK_0, t_STACK_1, t_STACK_2,  t_STACK_3,
50              t_STACK_4, t_STACK_5, t_STACK_6,  t_STACK_7,
51              t_STACK_8, t_STACK_9, t_STACK_A,  t_STACK_B,
52              t_STACK_C, t_STACK_D, t_STACK_E,  t_STACK_F
53                          : out STD_LOGIC_VECTOR (15 downto 0);
54
55              t_REG_0, t_REG_1, t_REG_2,  t_REG_3,
56              t_REG_4, t_REG_5, t_REG_6,  t_REG_7,
57              t_REG_8, t_REG_9, t_REG_A,  t_REG_B,
58              t_REG_C, t_REG_D, t_REG_E,  t_REG_F
59                          : out STD_LOGIC_VECTOR (7 downto 0);
60              clk : in STD_LOGIC;
61              active : out STD_LOGIC;
62              reset : in STD_LOGIC;
63              err_code : out STD_LOGIC_VECTOR( 2 downto 0 ));
64      end component;
65
66      component clock_div
67          port ( reset, clk_in : in STD_LOGIC;
68                 clk_out : out STD_LOGIC );
69      end component;
70
71      component mem_controller
72          port ( memAddress : in STD_LOGIC_VECTOR (35 downto 0);
73                 dataIn : in STD_LOGIC_VECTOR (23 downto 0);
74                 dataOut : out STD_LOGIC_VECTOR (23 downto 0);
75                 valid : in STD_LOGIC_VECTOR (2 downto 0);
76                 done : out STD_LOGIC_VECTOR (2 downto 0);
77                 write : in STD_LOGIC_VECTOR (2 downto 0);
78                 hold : in STD_LOGIC_VECTOR (2 downto 0);
79                 gameSelect : in STD_LOGIC_VECTOR (3 downto 0);
80                 gameSelected : out STD_LOGIC_VECTOR (3 downto 0);
81                 mapped_out : out STD_LOGIC_VECTOR( 7 downto 0 );
82                 debug_read_data : out STD_LOGIC_VECTOR( 7 downto 0 );
83                 mem_state : out STD_LOGIC_VECTOR( 7 downto 0 );
84                 sys_reset : out STD_LOGIC;
85                 cpu_reset : out STD_LOGIC;
86                 active : out STD_LOGIC;
87                 step : in STD_LOGIC;
88                 clk, reset : in STD_LOGIC);
89      end component;
90
91      component time_keeper
92          port ( memRead : in STD_LOGIC_VECTOR (7 downto 0);
93                 memWrite : out STD_LOGIC_VECTOR (7 downto 0);
94                 memAddress : out STD_LOGIC_VECTOR (11 downto 0);
95                 mem_valid : out STD_LOGIC;
96                 mem_write : out STD_LOGIC;
```

```
97                mem_hold : out STD_LOGIC;
98                mem_done : in STD_LOGIC;
99                clk : in STD_LOGIC;
100               reset : in STD_LOGIC;
101               heart_beat : out STD_LOGIC;
102               buzz : out STD_LOGIC);
103       end component;
104
105       component vga_controller
106           port ( memRead : in STD_LOGIC_VECTOR (7 downto 0);
107                  memWrite : out STD_LOGIC_VECTOR (7 downto 0);
108                  memAddress : out STD_LOGIC_VECTOR (11 downto 0);
109                  mem_valid : out STD_LOGIC;
110                  mem_write : out STD_LOGIC;
111                  mem_hold : out STD_LOGIC;
112                  mem_done : in STD_LOGIC;
113                  active : out STD_LOGIC;
114                  clk, sys_clk : in STD_LOGIC;
115                  reset, sys_reset : in STD_LOGIC;
116                  VGA_HS_O : out  STD_LOGIC;
117                  VGA_VS_O : out  STD_LOGIC;
118                  VGA_RED_O    : out  STD_LOGIC_VECTOR (3 downto 0);
119                  VGA_GREEN_O  : out  STD_LOGIC_VECTOR (3 downto 0);
120                  VGA_BLUE_O   : out  STD_LOGIC_VECTOR (3 downto 0));
121       end component;
122
123       signal s_CPU_memRead : STD_LOGIC_VECTOR (7 downto 0);
124       signal s_CPU_memWrite : STD_LOGIC_VECTOR (7 downto 0);
125       signal s_CPU_memAddress : STD_LOGIC_VECTOR (11 downto 0);
126       signal s_CPU_mem_valid : STD_LOGIC;
127       signal s_CPU_mem_write : STD_LOGIC;
128       signal s_CPU_mem_hold : STD_LOGIC;
129       signal s_CPU_mem_done : STD_LOGIC;
130       signal s_cpu_state : STD_LOGIC_VECTOR( 7 downto 0 );
131
132       signal s_TIME_memRead : STD_LOGIC_VECTOR (7 downto 0);
133       signal s_TIME_memWrite : STD_LOGIC_VECTOR (7 downto 0);
134       signal s_TIME_memAddress : STD_LOGIC_VECTOR (11 downto 0);
135       signal s_TIME_mem_valid : STD_LOGIC;
136       signal s_TIME_mem_write : STD_LOGIC;
137       signal s_TIME_mem_hold : STD_LOGIC;
138       signal s_TIME_mem_done : STD_LOGIC;
139
140       signal s_VGA_memRead : STD_LOGIC_VECTOR (7 downto 0);
141       signal s_VGA_memWrite : STD_LOGIC_VECTOR (7 downto 0);
142       signal s_VGA_memAddress : STD_LOGIC_VECTOR (11 downto 0);
143       signal s_VGA_mem_valid : STD_LOGIC;
144       signal s_VGA_mem_write : STD_LOGIC;
145       signal s_VGA_mem_hold : STD_LOGIC;
146       signal s_VGA_mem_done : STD_LOGIC;
```

```
147     signal s_gameSelected : STD_LOGIC_VECTOR (3 downto 0);
148 
149     signal s_memAddress : STD_LOGIC_VECTOR (35 downto 0);
150     signal s_dataIn : STD_LOGIC_VECTOR (23 downto 0);
151     signal s_dataOut : STD_LOGIC_VECTOR (23 downto 0);
152     signal s_valid : STD_LOGIC_VECTOR (2 downto 0);
153     signal s_done : STD_LOGIC_VECTOR (2 downto 0);
154     signal s_write : STD_LOGIC_VECTOR (2 downto 0);
155     signal s_hold : STD_LOGIC_VECTOR (2 downto 0);
156 
157     signal s_sys_reset : STD_LOGIC;
158     signal s_cpu_reset : STD_LOGIC;
159     signal s_mem_state : STD_LOGIC_VECTOR (7 downto 0);
160     signal s_debug_read_data : STD_LOGIC_VECTOR( 7 downto 0 );
161     signal s_step : STD_LOGIC;
162 
163     signal s_clock : std_logic;
164 
165     signal s_t_PC : STD_LOGIC_VECTOR ( 11 downto 0 );
166     signal s_t_I : STD_LOGIC_VECTOR ( 11 downto 0 );
167 
168     signal s_t_addr : STD_LOGIC_VECTOR ( 11 downto 0 );
169     signal s_t_n, s_t_x, s_t_y : STD_LOGIC_VECTOR ( 3 downto 0 );
170     signal s_t_kk : STD_LOGIC_VECTOR ( 7 downto 0 );
171 
172     signal s_t_SP : STD_LOGIC_VECTOR ( 7 downto 0 );
173     signal  s_t_STACK_0,  s_t_STACK_1,  s_t_STACK_2,  s_t_STACK_3,
174             s_t_STACK_4,  s_t_STACK_5,  s_t_STACK_6,  s_t_STACK_7,
175             s_t_STACK_8,  s_t_STACK_9,  s_t_STACK_A,  s_t_STACK_B,
176             s_t_STACK_C,  s_t_STACK_D,  s_t_STACK_E,  s_t_STACK_F
177                 : STD_LOGIC_VECTOR (15 downto 0);
178 
179     signal  s_t_REG_0,  s_t_REG_1,  s_t_REG_2,  s_t_REG_3,
180             s_t_REG_4,  s_t_REG_5,  s_t_REG_6,  s_t_REG_7,
181             s_t_REG_8,  s_t_REG_9,  s_t_REG_A,  s_t_REG_B,
182             s_t_REG_C,  s_t_REG_D,  s_t_REG_E,  s_t_REG_F
183                 : STD_LOGIC_VECTOR (7 downto 0);
184 
185     signal audioAMPsig : std_logic := '0';
186     signal counter: unsigned(15 DOWNTO 0) := x"0000";
187 
188     signal s_buzz : STD_LOGIC;
189 begin
190 
191     buzz <= s_buzz;
192     process(clk) begin
193         if rising_edge(clk) then
194             if(counter = 53628) then
195                 audioAMPsig <= not(audioAMPsig);
196                 counter <= x"0000";
```

```
197             else
198                 counter <= counter + 1;
199             end if;
200         end if;
201     end process;
202 
203     audioAMP <= audioAMPsig;
204     audioEn <= s_buzz;
205 
206     s_step <= '0';
207     -- s_step <= step; -- for debugging memory
208 
209     s_memAddress <= s_CPU_memAddress & s_TIME_memAddress & s_VGA_memAddress;
210     s_dataIn <= s_CPU_memWrite & s_TIME_memWrite & s_VGA_memWrite;
211     s_valid <= s_CPU_mem_valid & s_TIME_mem_valid & s_VGA_mem_valid;
212     s_write <= s_CPU_mem_write & s_TIME_mem_write & s_VGA_mem_write;
213     s_hold <= s_CPU_mem_hold & s_TIME_mem_hold & s_VGA_mem_hold;
214 
215     s_CPU_memRead <= s_dataOut( 23 downto 16 );
216     s_TIME_memRead <= s_dataOut( 15 downto 8 );
217     s_VGA_memRead <= s_dataOut( 7 downto 0 );
218 
219     s_CPU_mem_done <= s_done(2);
220     s_TIME_mem_done <= s_done(1);
221     s_VGA_mem_done <= s_done(0);
222 
223     my_core : core
224         port map ( memRead => s_CPU_memRead,
225                    memWrite => s_CPU_memWrite,
226                    memAddress => s_CPU_memAddress,
227                    keyPad => keyPad,
228                    mem_valid => s_CPU_mem_valid,
229                    mem_write => s_CPU_mem_write,
230                    mem_hold => s_CPU_mem_hold,
231                    mem_done => s_CPU_mem_done,
232                    cpu_state => s_cpu_state,
233                    t_PC => s_t_PC,
234                    t_I => s_t_I,
235 
236                    t_addr => s_t_addr,
237                    t_n => s_t_n, t_x => s_t_x, t_y => s_t_y,
238                    t_kk => s_t_kk,
239 
240                    t_SP => s_t_SP,
241                    t_STACK_0  => s_t_STACK_0,   t_STACK_1  => s_t_STACK_1,
242                    t_STACK_2  => s_t_STACK_2,   t_STACK_3  => s_t_STACK_3,
243                    t_STACK_4  => s_t_STACK_4,   t_STACK_5  => s_t_STACK_5,
244                    t_STACK_6  => s_t_STACK_6,   t_STACK_7  => s_t_STACK_7,
245                    t_STACK_8  => s_t_STACK_8,   t_STACK_9  => s_t_STACK_9,
246                    t_STACK_A  => s_t_STACK_A,   t_STACK_B  => s_t_STACK_B,
```

```
247                 t_STACK_C  => s_t_STACK_C,   t_STACK_D  => s_t_STACK_D,
248                 t_STACK_E  => s_t_STACK_E,   t_STACK_F  => s_t_STACK_F,
249
250                 t_REG_0  => s_t_REG_0,    t_REG_1  => s_t_REG_1,
251                 t_REG_2  => s_t_REG_2,   t_REG_3  => s_t_REG_3,
252                 t_REG_4  => s_t_REG_4,    t_REG_5  => s_t_REG_5,
253                 t_REG_6  => s_t_REG_6,   t_REG_7  => s_t_REG_7,
254                 t_REG_8  => s_t_REG_8,    t_REG_9  => s_t_REG_9,
255                 t_REG_A  => s_t_REG_A,   t_REG_B  => s_t_REG_B,
256                 t_REG_C  => s_t_REG_C,    t_REG_D  => s_t_REG_D,
257                 t_REG_E  => s_t_REG_E,   t_REG_F  => s_t_REG_F,
258
259                 clk => s_clock,
260                 reset => s_cpu_reset,
261                 active => cpu_active,
262                 err_code => err_code);
263
264     my_clock_div : clock_div
265         port map (  reset => reset,
266                     clk_in => clk,
267                     clk_out => s_clock );
268
269     my_mem_controller : mem_controller
270         port map ( memAddress => s_memAddress,
271                 dataIn => s_dataIn,
272                 dataOut => s_dataOut,
273                 valid => s_valid,
274                 done => s_done,
275                 write => s_write,
276                 hold => s_hold,
277                 gameSelect => gameSelect,
278                 gameSelected => s_gameSelected,
279                 mapped_out => mapped_out,
280                 debug_read_data => s_debug_read_data,
281                 mem_state => s_mem_state,
282                 sys_reset => s_sys_reset,
283                 cpu_reset => s_cpu_reset,
284                 step => s_step,
285                 clk => s_clock,
286                 active => mem_active,
287                 reset => reset );
288
289     my_time_keeper : time_keeper
290         port map ( memRead => s_TIME_memRead,
291                 memWrite => s_TIME_memWrite,
292                 memAddress => s_TIME_memAddress,
293                 mem_valid => s_TIME_mem_valid,
294                 mem_write => s_TIME_mem_write,
295                 mem_hold => s_TIME_mem_hold,
296                 mem_done => s_TIME_mem_done,
```

```
297                     clk => s_clock,
298                     reset => s_sys_reset,
299                     heart_beat => heart_beat,
300                     buzz => s_buzz );
301 
302     my_vga_controller : vga_controller
303         port map ( memRead => s_VGA_memRead,
304                     memWrite => s_VGA_memWrite,
305                     memAddress => s_VGA_memAddress,
306                     mem_valid => s_VGA_mem_valid,
307                     mem_write => s_VGA_mem_write,
308                     mem_hold => s_VGA_mem_hold,
309                     mem_done => s_VGA_mem_done,
310                     clk => clk,
311                     sys_clk => s_clock,
312                     active => vga_active,
313                     reset => reset,
314                     sys_reset => s_sys_reset,
315                     VGA_HS_O => VGA_HS_O,
316                     VGA_VS_O => VGA_VS_O,
317                     VGA_RED_O => VGA_RED_O,
318                     VGA_GREEN_O => VGA_GREEN_O,
319                     VGA_BLUE_O => VGA_BLUE_O );
320 end Behavioral;
```

## 6.2 Core

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity core is
    Port ( memRead : in STD_LOGIC_VECTOR (7 downto 0);
           memWrite : out STD_LOGIC_VECTOR (7 downto 0);
           memAddress : out STD_LOGIC_VECTOR (11 downto 0);
           keyPad : in STD_LOGIC_VECTOR (15 downto 0);
           mem_valid : out STD_LOGIC;
           mem_write : out STD_LOGIC;
           mem_hold : out STD_LOGIC;
           mem_done : in STD_LOGIC;
           cpu_state : out STD_LOGIC_VECTOR( 7 downto 0 );

           t_PC : out STD_LOGIC_VECTOR ( 11 downto 0 );
           t_I : out STD_LOGIC_VECTOR ( 11 downto 0 );

           t_addr : out STD_LOGIC_VECTOR ( 11 downto 0 );
           t_n, t_x, t_y : out STD_LOGIC_VECTOR ( 3 downto 0 );
           t_kk : out STD_LOGIC_VECTOR ( 7 downto 0 );

           t_SP : out STD_LOGIC_VECTOR ( 7 downto 0 );
           t_STACK_0, t_STACK_1, t_STACK_2,  t_STACK_3,
           t_STACK_4, t_STACK_5, t_STACK_6,  t_STACK_7,
           t_STACK_8, t_STACK_9, t_STACK_A, t_STACK_B,
           t_STACK_C, t_STACK_D, t_STACK_E, t_STACK_F
                       : out STD_LOGIC_VECTOR (15 downto 0);

           t_REG_0, t_REG_1, t_REG_2,  t_REG_3,
           t_REG_4, t_REG_5, t_REG_6,  t_REG_7,
           t_REG_8, t_REG_9, t_REG_A, t_REG_B,
           t_REG_C, t_REG_D, t_REG_E, t_REG_F
                       : out STD_LOGIC_VECTOR (7 downto 0);

           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           active : out STD_LOGIC;
           err_code : out STD_LOGIC_VECTOR( 2 downto 0 ) );
end core;

architecture Behavioral of core is
    constant DT_ADDRESS  : std_logic_vector(11 downto 0):= "000000000000";
    constant ST_ADDRESS  : std_logic_vector(11 downto 0):= "000000000001";

    constant INVALID_OP  : std_logic_vector(2 downto 0):= "001";
    constant STACK_OVER  : std_logic_vector(2 downto 0):= "010";
```

```
49      constant STACK_UNDER : std_logic_vector(2 downto 0):= "011";
50
51      type state is ( initA, error, fetchA, fetchB, delay,
52                      memA, memB, Decode,
53                      readKey0, readKey1, readKey2, readKey3,
54                      readKey4, readKey5, readKey6, readKey7,
55                      O_NOP, O_CLEAR, O_RET, O_JUMP, O_CALL,
56                      O_SNI_X_EQ_KK, O_SNI_X_NE_KK, O_SNI_X_EQ_Y, O_LD_X_KK, O_ADD_X_KK,
57                      O_LD_X_Y, O_OR_X_Y, O_AND_X_Y, O_XOR_X_Y, O_ADC_X_Y,
58                      O_SUB_X_Y, O_SHR_X_Y, O_SUBN_X_Y, O_SHL_X_Y, O_SNI_X_NE_Y,
59                      O_LD_I_ADDR, O_JMP_V0_ADDR, O_RND, O_DRW, O_SNI_KEY_X,
60                      O_SNI_KEY_NX, O_LD_X_DT, O_LD_X_KEY, O_LD_DT_X, O_LD_ST_X,
61                      O_ADD_I_X, O_LD_F_X, O_LD_B_X, O_LD_I_X, O_LD_X_I,
62                      CLEAR_A, getSprite, grab_graphicsA, grab_graphicsB,
63                      drawSprite, drawSpriteA, drawSpriteB, getDT,
64                      BCD_hundreds, BCD_tens, BCD_ones, pull_reg );
65      signal current_state : state;
66      signal tmp_err_code : STD_LOGIC_VECTOR( 2 downto 0 );
67
68      signal mem_ret_state : state;
69      signal tmp_mem_write : STD_LOGIC;
70      signal mem_ret_data : STD_LOGIC_VECTOR (7 downto 0);
71
72      signal PC : STD_LOGIC_VECTOR ( 11 downto 0 );
73      signal instruction_high : STD_LOGIC_VECTOR ( 7 downto 0 );
74
75      signal addr : STD_LOGIC_VECTOR ( 11 downto 0 );
76      signal n, x, y : STD_LOGIC_VECTOR ( 3 downto 0 );
77      signal kk : STD_LOGIC_VECTOR ( 7 downto 0 );
78
79      signal SP : STD_LOGIC_VECTOR ( 7 downto 0 );
80      type STACK is array( 15 downto 0 ) of STD_LOGIC_VECTOR (15 downto 0);
81      signal cpu_STACK : STACK;
82
83      type REG is array( 15 downto 0 ) of STD_LOGIC_VECTOR (7 downto 0);
84      signal cpu_REG : REG;
85
86      type POINTER_TABLE is array( 15 downto 0 ) of STD_LOGIC_VECTOR (11 downto 0);
87      signal hex_digits : POINTER_TABLE;
88
89      signal I : STD_LOGIC_VECTOR ( 11 downto 0 );
90
91      --- poor man's RNG, since a human will let up on the reset,
92      --- should start at a random point at least.
93      signal RAND : STD_LOGIC_VECTOR ( 15 downto 0 ) := "1010101010101010";
94
95      signal multi_address : STD_LOGIC_VECTOR ( 11 downto 0 );
96      signal multi_count : STD_LOGIC_VECTOR ( 8 downto 0 );
97
98      signal graphic_addressA : STD_LOGIC_VECTOR ( 11 downto 0 );
```

```
99      signal graphic_addressB : STD_LOGIC_VECTOR ( 11 downto 0 );
100     signal graphic_bufferA : STD_LOGIC_VECTOR ( 7 downto 0 );
101     signal graphic_bufferB : STD_LOGIC_VECTOR ( 7 downto 0 );
102     signal sprite_buffer    : STD_LOGIC_VECTOR ( 7 downto 0 );
103     signal sprite_buffer_flip    : STD_LOGIC_VECTOR ( 7 downto 0 );
104     signal graphic_offset : STD_LOGIC_VECTOR ( 2 downto 0 );
105     signal graphic_collision : STD_LOGIC;
106
107     signal key_counter : STD_LOGIC_VECTOR ( 3 downto 0 );
108     signal BCD_total : STD_LOGIC_VECTOR ( 7 downto 0 );
109     signal BCD_left : STD_LOGIC_VECTOR ( 7 downto 0 );
110     signal reg_copy_num : STD_LOGIC_VECTOR ( 3 downto 0 );
111
112     signal instruction_delay : STD_LOGIC_VECTOR ( 23 downto 0 );
113     signal current_delay : STD_LOGIC_VECTOR ( 23 downto 0 );
114
115     signal s_keyPad : STD_LOGIC_VECTOR (15 downto 0);
116     type KEY is array( 15 downto 0 ) of STD_LOGIC_VECTOR (3 downto 0);
117     signal map_KEY : KEY;
118
119 begin
120     mem_write <= tmp_mem_write;
121     err_code <= tmp_err_code;
122
123     s_keyPad( 0) <= keyPad(to_integer( unsigned (map_KEY(  0 ))) );
124     s_keyPad( 1) <= keyPad(to_integer( unsigned (map_KEY(  1 ))) );
125     s_keyPad( 2) <= keyPad(to_integer( unsigned (map_KEY(  2 ))) );
126     s_keyPad( 3) <= keyPad(to_integer( unsigned (map_KEY(  3 ))) );
127     s_keyPad( 4) <= keyPad(to_integer( unsigned (map_KEY(  4 ))) );
128     s_keyPad( 5) <= keyPad(to_integer( unsigned (map_KEY(  5 ))) );
129     s_keyPad( 6) <= keyPad(to_integer( unsigned (map_KEY(  6 ))) );
130     s_keyPad( 7) <= keyPad(to_integer( unsigned (map_KEY(  7 ))) );
131     s_keyPad( 8) <= keyPad(to_integer( unsigned (map_KEY(  8 ))) );
132     s_keyPad( 9) <= keyPad(to_integer( unsigned (map_KEY(  9 ))) );
133     s_keyPad(10) <= keyPad(to_integer( unsigned (map_KEY( 10 ))) );
134     s_keyPad(11) <= keyPad(to_integer( unsigned (map_KEY( 11 ))) );
135     s_keyPad(12) <= keyPad(to_integer( unsigned (map_KEY( 12 ))) );
136     s_keyPad(13) <= keyPad(to_integer( unsigned (map_KEY( 13 ))) );
137     s_keyPad(14) <= keyPad(to_integer( unsigned (map_KEY( 14 ))) );
138     s_keyPad(15) <= keyPad(to_integer( unsigned (map_KEY( 15 ))) );
139
140     t_PC <= PC;
141     t_I <= I;
142
143     t_addr <= addr;
144     t_n <= n;
145     t_x <= x;
146     t_y <= y;
147     t_kk <= kk;
148
```

```
149     t_SP <= SP;
150     t_STACK_0  <= cpu_STACK( 0);   t_STACK_1  <= cpu_STACK( 1);
151     t_STACK_2  <= cpu_STACK( 2);   t_STACK_3  <= cpu_STACK( 3);
152     t_STACK_4  <= cpu_STACK( 4);   t_STACK_5  <= cpu_STACK( 5);
153     t_STACK_6  <= cpu_STACK( 6);   t_STACK_7  <= cpu_STACK( 7);
154     t_STACK_8  <= cpu_STACK( 8);   t_STACK_9  <= cpu_STACK( 9);
155     t_STACK_A  <= cpu_STACK(10);   t_STACK_B  <= cpu_STACK(11);
156     t_STACK_C  <= cpu_STACK(12);   t_STACK_D  <= cpu_STACK(13);
157     t_STACK_E  <= cpu_STACK(14);   t_STACK_F  <= cpu_STACK(15);
158
159     t_REG_0  <= cpu_REG( 0);   t_REG_1  <= cpu_REG( 1);
160     t_REG_2  <= cpu_REG( 2);   t_REG_3  <= cpu_REG( 3);
161     t_REG_4  <= cpu_REG( 4);   t_REG_5  <= cpu_REG( 5);
162     t_REG_6  <= cpu_REG( 6);   t_REG_7  <= cpu_REG( 7);
163     t_REG_8  <= cpu_REG( 8);   t_REG_9  <= cpu_REG( 9);
164     t_REG_A  <= cpu_REG(10);   t_REG_B  <= cpu_REG(11);
165     t_REG_C  <= cpu_REG(12);   t_REG_D  <= cpu_REG(13);
166     t_REG_E  <= cpu_REG(14);   t_REG_F  <= cpu_REG(15);
167
168     process( clk, reset )
169         variable instruction : STD_LOGIC_VECTOR ( 15 downto 0 );
170         variable tmp_SP : STD_LOGIC_VECTOR ( 7 downto 0 );
171         variable tmp_8 : STD_LOGIC_VECTOR ( 7 downto 0 );
172         variable tmp_8x : STD_LOGIC_VECTOR ( 7 downto 0 );
173         variable tmp_8y : STD_LOGIC_VECTOR ( 7 downto 0 );
174         variable tmp_8A : STD_LOGIC_VECTOR ( 7 downto 0 );
175         variable tmp_8B : STD_LOGIC_VECTOR ( 7 downto 0 );
176         variable tmp_4 : STD_LOGIC_VECTOR ( 3 downto 0 );
177         variable tmp_12 :  STD_LOGIC_VECTOR ( 11 downto 0 );
178         variable tmp_12A : STD_LOGIC_VECTOR ( 11 downto 0 );
179         variable tmp_12B : STD_LOGIC_VECTOR ( 11 downto 0 );
180         variable key_num : STD_LOGIC_VECTOR ( 3 downto 0 );
181         variable key_mask : STD_LOGIC_VECTOR ( 15 downto 0 );
182         variable math_buf : STD_LOGIC_VECTOR ( 8 downto 0 );
183     begin
184         if( reset = '1' ) then
185             current_state <= initA;
186             memWrite <= "00000000";
187             memAddress <= x"000";
188             mem_valid <= '0';
189             tmp_mem_write <= '0';
190             mem_hold <= '0';
191             tmp_err_code <= "000";
192             cpu_state <= x"01";
193             active <= '0';
194             if ( rising_edge( clk ) ) then
195                 RAND <= ( RAND(0) xor RAND(2) xor RAND(3) xor RAND(5) ) &
196                                                         RAND( 15 downto 1);
197             end if;
198         elsif ( rising_edge( clk ) ) then
```

```
199     RAND <= ( RAND(0) xor RAND(2) xor RAND(3) xor RAND(5) ) &
200                                         RAND( 15 downto 1);
201     current_state <= current_state;
202     case current_state is
203         when error =>
204             cpu_state <= x"00";
205         when initA =>
206             cpu_state <= x"02";
207             active <= '1';
208             PC <= x"200";
209             I <= "000000000000";
210             SP <= "00000000";
211 
212             cpu_REG( 0) <= "00000000";  cpu_REG( 8) <= "00000000";
213             cpu_REG( 1) <= "00000000";  cpu_REG( 9) <= "00000000";
214             cpu_REG( 2) <= "00000000";  cpu_REG(10) <= "00000000";
215             cpu_REG( 3) <= "00000000";  cpu_REG(11) <= "00000000";
216             cpu_REG( 4) <= "00000000";  cpu_REG(12) <= "00000000";
217             cpu_REG( 5) <= "00000000";  cpu_REG(13) <= "00000000";
218             cpu_REG( 6) <= "00000000";  cpu_REG(14) <= "00000000";
219             cpu_REG( 7) <= "00000000";  cpu_REG(15) <= "00000000";
220 
221             key_counter <= "0000";
222 
223             hex_digits( 0) <= "000000000010";  hex_digits( 8) <= "000000101010";
224             hex_digits( 1) <= "000000000111";  hex_digits( 9) <= "000000101111";
225             hex_digits( 2) <= "000000001100";  hex_digits(10) <= "000000110100";
226             hex_digits( 3) <= "000000010001";  hex_digits(11) <= "000000111001";
227             hex_digits( 4) <= "000000010110";  hex_digits(12) <= "000000111110";
228             hex_digits( 5) <= "000000011011";  hex_digits(13) <= "000001000011";
229             hex_digits( 6) <= "000000100000";  hex_digits(14) <= "000001001000";
230             hex_digits( 7) <= "000000100101";  hex_digits(15) <= "000001001101";
231 
232             instruction_delay <= x"002000";
233 
234             memAddress <= x"080";
235             tmp_mem_write <= '0';
236             mem_hold <= '0';
237             mem_ret_state <= readKey0;
238             current_state <= memA;
239         when readKey0  =>
240             map_KEY(0) <= mem_ret_data( 7 downto 4 );
241             map_KEY(1) <= mem_ret_data( 3 downto 0 );
242             memAddress <= x"081";
243             mem_ret_state <= readKey1;
244             current_state <= memA;
245         when readKey1  =>
246             map_KEY(2) <= mem_ret_data( 7 downto 4 );
247             map_KEY(3) <= mem_ret_data( 3 downto 0 );
248             memAddress <= x"082";
```

```
249             mem_ret_state <= readKey2;
250             current_state <= memA;
251         when readKey2  =>
252             map_KEY(4) <= mem_ret_data( 7 downto 4 );
253             map_KEY(5) <= mem_ret_data( 3 downto 0 );
254             memAddress <= x"083";
255             mem_ret_state <= readKey3;
256             current_state <= memA;
257         when readKey3  =>
258             map_KEY(6) <= mem_ret_data( 7 downto 4 );
259             map_KEY(7) <= mem_ret_data( 3 downto 0 );
260             memAddress <= x"084";
261             mem_ret_state <= readKey4;
262             current_state <= memA;
263         when readKey4  =>
264             map_KEY(8) <= mem_ret_data( 7 downto 4 );
265             map_KEY(9) <= mem_ret_data( 3 downto 0 );
266             memAddress <= x"085";
267             mem_ret_state <= readKey5;
268             current_state <= memA;
269         when readKey5  =>
270             map_KEY(10) <= mem_ret_data( 7 downto 4 );
271             map_KEY(11) <= mem_ret_data( 3 downto 0 );
272             memAddress <= x"086";
273             mem_ret_state <= readKey6;
274             current_state <= memA;
275         when readKey6  =>
276             map_KEY(12) <= mem_ret_data( 7 downto 4 );
277             map_KEY(13) <= mem_ret_data( 3 downto 0 );
278             memAddress <= x"087";
279             mem_ret_state <= readKey7;
280             current_state <= memA;
281         when readKey7  =>
282             map_KEY(14) <= mem_ret_data( 7 downto 4 );
283             map_KEY(15) <= mem_ret_data( 3 downto 0 );
284             current_state <= fetchA;
285         when fetchA =>
286             cpu_state <= x"03";
287             memAddress <= PC;
288             PC <= PC + x"001";
289             tmp_mem_write <= '0';
290             mem_hold <= '0';
291             mem_ret_state <= delay;
292             current_delay <= x"000000";
293             current_state <= memA;
294         when delay =>
295             current_delay <= current_delay + x"000001";
296             if( current_delay = instruction_delay ) then
297                 current_state <= fetchB;
298             end if;
```

```
299         when fetchB =>
300             cpu_state <= x"04";
301             instruction_high <= mem_ret_data;
302             memAddress <= PC;
303             PC <= PC + x"001";
304             mem_ret_state <= Decode;
305             current_state <= memA;
306         when memA =>
307             cpu_state <= x"05";
308             if ( mem_done = '0' ) then
309                 mem_valid <= '1';
310                 current_state <= memB;
311             end if;
312         when memB =>
313             cpu_state <= x"06";
314             if( mem_done = '1' ) then
315                 if ( tmp_mem_write = '0' ) then
316                     mem_ret_data <= memRead;
317                 end if;
318 
319                 mem_valid <= '0';
320                 current_state <= mem_ret_state;
321             end if;
322         when Decode =>
323             cpu_state <= x"07";
324             instruction := instruction_high & mem_ret_data;
325             case instruction( 15 downto 12 ) is
326                 when "0000" =>
327                     if ( instruction = "0000000011100000" ) then
328                         current_state <= O_CLEAR;
329                     elsif ( instruction = "0000000011101110" ) then
330                         current_state <= O_RET;
331                     else
332                         current_state <= O_NOP;
333                     end if;
334                 when "0001" =>
335                     addr <= instruction( 11 downto 0 );
336                     current_state <= O_JUMP;
337                 when "0010" =>
338                     addr <= instruction( 11 downto 0 );
339                     current_state <= O_CALL;
340                 when "0011" =>
341                     x <= instruction( 11 downto 8 );
342                     kk <= instruction( 7 downto 0 );
343                     current_state <= O_SNI_X_EQ_KK;
344                 when "0100" =>
345                     x <= instruction( 11 downto 8 );
346                     kk <= instruction( 7 downto 0 );
347                     current_state <= O_SNI_X_NE_KK;
348                 when "0101" =>
```

```
349                         x <= instruction( 11 downto 8 );
350                         y <= instruction( 7 downto 4 );
351                         current_state <= O_SNI_X_EQ_Y;
352                     when "0110" =>
353                         x <= instruction( 11 downto 8 );
354                         kk <= instruction( 7 downto 0 );
355                         current_state <= O_LD_X_KK;
356                     when "0111" =>
357                         x <= instruction( 11 downto 8 );
358                         kk <= instruction( 7 downto 0 );
359                         current_state <= O_ADD_X_KK;
360                     when "1000" =>
361                         x <= instruction( 11 downto 8 );
362                         y <= instruction( 7 downto 4 );
363                         case instruction( 3 downto 0 ) is
364                             when "0000" =>
365                                 current_state <= O_LD_X_Y;
366                             when "0001" =>
367                                 current_state <= O_OR_X_Y;
368                             when "0010" =>
369                                 current_state <= O_AND_X_Y;
370                             when "0011" =>
371                                 current_state <= O_XOR_X_Y;
372                             when "0100" =>
373                                 current_state <= O_ADC_X_Y;
374                             when "0101" =>
375                                 current_state <= O_SUB_X_Y;
376                             when "0110" =>
377                                 current_state <= O_SHR_X_Y;
378                             when "0111" =>
379                                 current_state <= O_SUBN_X_Y;
380                             when "1110" =>
381                                 current_state <= O_SHL_X_Y;
382                             when others =>
383                                 tmp_err_code <= INVALID_OP;
384                                 current_state <= error;
385                         end case;
386                     when "1001" =>
387                         x <= instruction( 11 downto 8 );
388                         y <= instruction( 7 downto 4 );
389                         current_state <= O_SNI_X_NE_Y;
390                     when "1010" =>
391                         addr <= instruction( 11 downto 0 );
392                         current_state <= O_LD_I_ADDR;
393                     when "1011" =>
394                         addr <= instruction( 11 downto 0 );
395                         current_state <= O_JMP_V0_ADDR;
396                     when "1100" =>
397                         x <= instruction( 11 downto 8 );
398                         kk <= instruction( 7 downto 0 );
```

```
399                     current_state <= O_RND;
400                 when "1101" =>
401                     x <= instruction( 11 downto 8 );
402                     y <= instruction( 7 downto 4 );
403                     n <= instruction( 3 downto 0 );
404                     current_state <= O_DRW;
405                 when "1110" =>
406                     x <= instruction( 11 downto 8 );
407                     if ( instruction( 7 downto 0 ) = "10011110" ) then
408                         current_state <= O_SNI_KEY_X;
409                     elsif ( instruction( 7 downto 0 ) = "10100001" ) then
410                         current_state <= O_SNI_KEY_NX;
411                     else
412                         tmp_err_code <= INVALID_OP;
413                         current_state <= error;
414                     end if;
415                 when "1111" =>
416                     x <= instruction( 11 downto 8 );
417                     case instruction( 7 downto 0 ) is
418                         when x"07" =>
419                             current_state <= O_LD_X_DT;
420                         when x"0A" =>
421                             current_state <= O_LD_X_KEY;
422                         when x"15" =>
423                             current_state <= O_LD_DT_X;
424                         when x"18" =>
425                             current_state <= O_LD_ST_X;
426                         when x"1E" =>
427                             current_state <= O_ADD_I_X;
428                         when x"29" =>
429                             current_state <= O_LD_F_X;
430                         when x"33" =>
431                             current_state <= O_LD_B_X;
432                         when x"55" =>
433                             current_state <= O_LD_I_X;
434                         when x"65" =>
435                             current_state <= O_LD_X_I;
436                         when others =>
437                             tmp_err_code <= INVALID_OP;
438                             current_state <= error;
439                     end case;
440                 when others =>
441                     tmp_err_code <= INVALID_OP;
442                     current_state <= error;
443             end case;
444         when O_NOP =>
445             cpu_state <= x"08";
446             current_state <= fetchA;
447         when O_CLEAR =>
448             cpu_state <= x"09";
```

```
449             multi_address <= "111100000000"; -- 0xF00
450             multi_count <= "100000000"; -- 256
451             current_state <= CLEAR_A;
452         when CLEAR_A =>
453             cpu_state <= x"0A";
454             if ( multi_count = "000000000" ) then
455                 current_state <= fetchA;
456             else
457                 memAddress <= multi_address;
458                 tmp_mem_write <= '1';
459                 mem_hold <= '0';
460                 mem_ret_state <= CLEAR_A;
461                 memWrite <= "00000000";
462                 current_state <= memA;
463 
464                 multi_address <= multi_address + "000000000001";
465                 multi_count <= multi_count - "000000001";
466             end if;
467         when O_RET =>
468             cpu_state <= x"0B";
469             tmp_SP := SP;
470             tmp_SP := tmp_SP - "00000001";
471             if( tmp_SP = "11111111" ) then
472                 tmp_err_code <= STACK_UNDER;
473                 current_state <= error;
474             else
475                 PC <= cpu_STACK( to_integer( unsigned ( tmp_SP ) ))( 11 downto 0 );
476                 SP <= tmp_SP;
477                 current_state <= fetchA;
478             end if;
479         when O_JUMP =>
480             cpu_state <= x"0C";
481             PC <= addr;
482             current_state <= fetchA;
483         when O_CALL =>
484             cpu_state <= x"0D";
485             tmp_SP := SP;
486             if( tmp_SP = "00010000" ) then    -- 16 is to much
487                 tmp_err_code <= STACK_OVER;
488                 current_state <= error;
489             else
490                 cpu_STACK( to_integer( unsigned ( tmp_SP ) )) <= "0000" & PC;
491                 tmp_SP := tmp_SP + "00000001";
492                 SP <= tmp_SP;
493                 PC <= addr;
494                 current_state <= fetchA;
495             end if;
496         when O_SNI_X_EQ_KK =>
497             cpu_state <= x"0E";
498             if ( cpu_REG( to_integer( unsigned ( x ) ) ) = kk ) then
```

```
499                 PC <= PC + "000000000010";
500             end if;
501 
502             current_state <= fetchA;
503         when O_SNI_X_NE_KK =>
504             cpu_state <= x"0F";
505             if ( cpu_REG( to_integer( unsigned ( x ) ) ) /= kk ) then
506                 PC <= PC + "000000000010";
507             end if;
508 
509             current_state <= fetchA;
510         when O_SNI_X_EQ_Y =>
511              cpu_state <= x"10";
512              if ( cpu_REG( to_integer( unsigned ( x ) ) )
513                   =
514                   cpu_REG( to_integer( unsigned ( y ) ) ) ) then
515                 PC <= PC + "000000000010";
516              end if;
517 
518              current_state <= fetchA;
519         when O_LD_X_KK =>
520             cpu_state <= x"11";
521             cpu_REG(to_integer(unsigned( x ))) <= kk;
522             current_state <= fetchA;
523         when O_ADD_X_KK =>
524             cpu_state <= x"12";
525             cpu_REG(to_integer(unsigned( x ))) <=
526                     cpu_REG(to_integer(unsigned( x ))) + kk;
527             current_state <= fetchA;
528         when O_LD_X_Y =>
529             cpu_state <= x"13";
530             cpu_REG(to_integer(unsigned( x ))) <=
531                     cpu_REG(to_integer(unsigned( y )));
532             current_state <= fetchA;
533         when O_OR_X_Y =>
534             cpu_state <= x"14";
535             cpu_REG(to_integer(unsigned( x ))) <=
536                     cpu_REG(to_integer(unsigned( x )))
537                     or
538                     cpu_REG(to_integer(unsigned( y )));
539             current_state <= fetchA;
540         when O_AND_X_Y =>
541             cpu_state <= x"15";
542             cpu_REG(to_integer(unsigned( x ))) <=
543                     cpu_REG(to_integer(unsigned( x )))
544                     and
545                     cpu_REG(to_integer(unsigned( y )));
546             current_state <= fetchA;
547         when O_XOR_X_Y =>
548             cpu_state <= x"16";
```

```
549             cpu_REG(to_integer(unsigned( x ))) <=
550                    cpu_REG(to_integer(unsigned( x )))
551                    xor
552                    cpu_REG(to_integer(unsigned( y )));
553             current_state <= fetchA;
554         when O_ADC_X_Y =>
555             cpu_state <= x"17";
556             math_buf :=
557                     (  '0' & cpu_REG(to_integer(unsigned( x )))  )
558                     +
559                     (  '0' & cpu_REG(to_integer(unsigned( y )))  );
560             cpu_REG(to_integer(unsigned( x ))) <= math_buf( 7 downto 0 );
561             cpu_REG( 15 ) <= "0000000" & math_buf( 8 );
562             current_state <= fetchA;
563         when O_SUB_X_Y =>
564             cpu_state <= x"18";
565             cpu_REG(to_integer(unsigned( x ))) <=
566                    cpu_REG(to_integer(unsigned( x )))
567                    -
568                    cpu_REG(to_integer(unsigned( y )));
569             if ( cpu_REG(to_integer(unsigned( x )))
570                  >=
571                  cpu_REG(to_integer(unsigned( y )))  ) then
572                 cpu_REG( 15 ) <= "00000001";
573             else
574                 cpu_REG( 15 ) <= "00000000";
575             end if;
576             current_state <= fetchA;
577         when O_SHR_X_Y =>
578             cpu_state <= x"19";
579             cpu_REG(to_integer(unsigned( x ))) <= '0' &
580                         cpu_REG(to_integer(unsigned( y )))( 7 downto 1 );
581             cpu_REG( 15 ) <= "0000000" &
582                                 cpu_REG(to_integer(unsigned( y )))(0);
583             current_state <= fetchA;
584         when O_SUBN_X_Y =>
585             cpu_state <= x"1A";
586             cpu_REG(to_integer(unsigned( x ))) <=
587                     cpu_REG(to_integer(unsigned( y )))
588                     -
589                     cpu_REG(to_integer(unsigned( x )));
590             if ( cpu_REG(to_integer(unsigned( y )))
591                  >=
592                  cpu_REG(to_integer(unsigned( x )))  ) then
593                cpu_REG( 15 ) <= "00000001";
594             else
595                cpu_REG( 15 ) <= "00000000";
596             end if;
597             current_state <= fetchA;
598         when O_SHL_X_Y =>
```

```
599             cpu_state <= x"1B";
600             cpu_REG(to_integer(unsigned( x ))) <=
601                    cpu_REG(to_integer(unsigned( y )))( 6 downto 0 ) & '0';
602             cpu_REG( 15 ) <= "0000000" &
603                                 cpu_REG(to_integer(unsigned( y )))(7);
604             current_state <= fetchA;
605         when O_SNI_X_NE_Y =>
606             cpu_state <= x"1C";
607             if ( cpu_REG( to_integer( unsigned ( x ) ) )
608                  /=
609                  cpu_REG( to_integer( unsigned ( y ) ) ) ) then
610                 PC <= PC + "000000000010";
611             end if;
612 
613             current_state <= fetchA;
614         when O_LD_I_ADDR =>
615             cpu_state <= x"1D";
616             I <= addr;
617             current_state <= fetchA;
618         when O_JMP_V0_ADDR =>
619             cpu_state <= x"1E";
620             PC <= addr + ( "0000" & cpu_REG(0) );
621             current_state <= fetchA;
622         when O_RND =>
623             cpu_state <= x"1F";
624             tmp_8 := RAND( 7 downto 0 );
625             tmp_8 := tmp_8 and kk;
626             cpu_REG( to_integer( unsigned ( x ) ) ) <= tmp_8;
627             current_state <= fetchA;
628         when O_DRW =>
629             cpu_state <= x"20";
630             tmp_8x := "00" & cpu_REG( to_integer( unsigned ( x ) ) )(5 downto 0);
631             tmp_8y := "000" & cpu_REG( to_integer( unsigned ( y ) ) )( 4 downto 0);
632 
633             -- address = 0xF00 + ( y << 3 ) + ( x >> 3 )
634             tmp_12 := "1111" & tmp_8y( 4 downto 0 ) & tmp_8x( 5 downto 3);
635             graphic_addressA <= tmp_12;
636 
637             -- address + 1 = address + 1, but roll over on 8 byte boundary
638             tmp_12 := tmp_12( 11 downto 3 ) & ( tmp_12( 2 downto 0 ) + "001");
639             graphic_addressB <= tmp_12;
640 
641             graphic_offset <= tmp_8x( 2 downto 0 );
642             graphic_collision <= '0';
643             multi_address <= I;
644             current_state <= getSprite;
645         when getSprite =>
646             cpu_state <= x"21";
647             -- when n gets to 0 we are done, if there was a collision set VF
648             if( n = "0000" ) then
```

```
649                 if ( graphic_collision = '1' ) then
650                     cpu_REG( 15 ) <= "00000001";
651                 else
652                     cpu_REG( 15 ) <= "00000000";
653                 end if;
654                 current_state <= fetchA;
655             else
656                 -- we have more to do, n--
657                 n <= n - "0001";
658                 -- readAddress = spriteAddress++
659                 memAddress <= multi_address;
660                 multi_address <= multi_address + "000000000001";
661                 -- read, do not hold memory
662                 tmp_mem_write <= '0';
663                 mem_hold <= '0';
664                 mem_ret_state <= grab_graphicsA;
665                 current_state <= memA;
666             end if;
667         when grab_graphicsA =>
668             cpu_state <= x"22";
669             -- grab the value
670             sprite_buffer_flip <= mem_ret_data;
671             -- get old info from video buffer low byte
672             memAddress <= graphic_addressA;
673             mem_ret_state <= grab_graphicsB;
674             current_state <= memA;
675         when grab_graphicsB =>
676             cpu_state <= x"23";
677             -- flip the sprite bits because bit 7 drawn at 0, 0
678             --  should be pixel 0 if that makes sense?
679             sprite_buffer(0) <= sprite_buffer_flip(7);
680             sprite_buffer(1) <= sprite_buffer_flip(6);
681             sprite_buffer(2) <= sprite_buffer_flip(5);
682             sprite_buffer(3) <= sprite_buffer_flip(4);
683             sprite_buffer(4) <= sprite_buffer_flip(3);
684             sprite_buffer(5) <= sprite_buffer_flip(2);
685             sprite_buffer(6) <= sprite_buffer_flip(1);
686             sprite_buffer(7) <= sprite_buffer_flip(0);
687             -- grab the value
688             graphic_bufferA <= mem_ret_data;
689             -- get old info from video buffer high byte
690             memAddress <= graphic_addressB;
691             mem_ret_state <= drawSprite;
692             current_state <= memA;
693         when drawSprite =>
694             cpu_state <= x"24";
695             -- xor the new data into highByte::lowByte at the corret offset
696             tmp_8A := graphic_bufferA;
697             tmp_8B := mem_ret_data;
698
```

```
699             if( graphic_offset = "000" ) then
700                 tmp_8A := tmp_8A xor sprite_buffer;
701             elsif ( graphic_offset = "001" ) then
702                 tmp_8B := tmp_8B xor ( "0000000" & sprite_buffer(7) );
703                 tmp_8A := tmp_8A xor ( sprite_buffer( 6 downto 0 ) & "0");
704             elsif ( graphic_offset = "010" ) then
705                 tmp_8B := tmp_8B xor ( "000000" & sprite_buffer(7 downto 6) );
706                 tmp_8A := tmp_8A xor ( sprite_buffer( 5 downto 0 ) & "00");
707             elsif ( graphic_offset = "011" ) then
708                 tmp_8B := tmp_8B xor ( "00000" & sprite_buffer(7 downto 5) );
709                 tmp_8A := tmp_8A xor ( sprite_buffer( 4 downto 0 ) & "000");
710             elsif ( graphic_offset = "100" ) then
711                 tmp_8B := tmp_8B xor ( "0000" & sprite_buffer(7 downto 4) );
712                 tmp_8A := tmp_8A xor ( sprite_buffer( 3 downto 0 ) & "0000");
713             elsif ( graphic_offset = "101" ) then
714                 tmp_8B := tmp_8B xor ( "000" & sprite_buffer(7 downto 3) );
715                 tmp_8A := tmp_8A xor ( sprite_buffer( 2 downto 0 ) & "00000");
716             elsif ( graphic_offset = "110" ) then
717                 tmp_8B := tmp_8B xor ( "00" & sprite_buffer(7 downto 2) );
718                 tmp_8A := tmp_8A xor ( sprite_buffer( 1 downto 0 ) & "000000");
719             elsif ( graphic_offset = "111" ) then
720                 tmp_8B := tmp_8B xor ( "0" & sprite_buffer(7 downto 1) );
721                 tmp_8A := tmp_8A xor ( sprite_buffer( 0 ) & "0000000");
722             end if;
723 
724             -- collision if the old bits are gone
725             if( ( graphic_bufferA and tmp_8A ) /= graphic_bufferA ) then
726                 graphic_collision <= '1';
727             elsif ( ( mem_ret_data and tmp_8B ) /= mem_ret_data ) then
728                 graphic_collision <= '1';
729             end if;
730 
731             -- store this here to remember we cannot use it this state, whoops!!
732             graphic_bufferB <= tmp_8B;
733             -- writeAddress = videoLowByte++8 to increment to the next row.
734             memAddress <= graphic_addressA;
735             graphic_addressA <= graphic_addressA + "00000001000";
736             -- we are writing the new xored video data here
737             tmp_mem_write <= '1';
738             memWrite <= tmp_8A;
739             mem_ret_state <= drawSpriteA;
740             current_state <= memA;
741         when drawSpriteA =>
742             cpu_state <= x"25";
743             -- writeAddress = videoHighByte++8 to increment to the next row.
744             memAddress <= graphic_addressB;
745             graphic_addressB <= graphic_addressB + "00000001000";
746             -- write the rest of the video update
747             memWrite <= graphic_bufferB;
748             mem_ret_state <= drawSpriteB;
```

```
749             current_state <= memA;
750         when drawSpriteB =>
751             cpu_state <= x"26";
752             -- if the addresses rolled off the bottom of the screen,
753             --  then put them back up at the top of the buffer
754             --  instead of top of memory
755             if( graphic_addressA( 11 downto 8 ) = "0000" ) then
756                 graphic_addressA <= graphic_addressA or x"F00";
757                 graphic_addressB <= graphic_addressB or x"F00";
758             end if;
759 
760             -- get the next sprite line.
761             current_state <= getSprite;
762         when O_SNI_KEY_X =>
763             cpu_state <= x"27";
764             key_num := cpu_REG( to_integer( unsigned ( x ) ) )( 3 downto 0 );
765             key_mask := "0000000000000001";
766 
767             key_mask := std_logic_vector(  shift_left( unsigned( key_mask ),
768                                 to_integer( unsigned ( key_num ) ) )  );
769             if( ( key_mask and s_keyPad ) = key_mask ) then
770                 PC <= PC + "000000000010";
771             end if;
772 
773             current_state <= fetchA;
774         when O_SNI_KEY_NX =>
775             cpu_state <= x"28";
776             key_num := cpu_REG( to_integer( unsigned ( x ) ) )( 3 downto 0 );
777             key_mask := "0000000000000001";
778 
779             key_mask := std_logic_vector(  shift_left( unsigned( key_mask ),
780                                 to_integer( unsigned ( key_num ) ) )  );
781             if( ( key_mask and s_keyPad ) /= key_mask ) then
782                 PC <= PC + "000000000010";
783             end if;
784 
785             current_state <= fetchA;
786         when O_LD_X_DT =>
787             cpu_state <= x"29";
788             memAddress <= DT_ADDRESS;
789             tmp_mem_write <= '0';
790             mem_hold <= '0';
791             mem_ret_state <= getDT;
792             current_state <= memA;
793         when getDT =>
794             cpu_state <= x"2A";
795             cpu_REG( to_integer( unsigned ( x ) ) ) <= mem_ret_data;
796             current_state <= fetchA;
797         when O_LD_X_KEY =>
798             cpu_state <= x"2B";
```

```
799             -- keep rolling through all 15 keys until we find one that is down.
800             key_num := key_counter;
801             key_mask := "0000000000000001";
802 
803             key_counter <= key_counter + 1;
804             key_mask := std_logic_vector(  shift_left( unsigned( key_mask ),
805                         to_integer( unsigned ( key_num ) ) )  );
806 
807             if( ( key_mask and s_keyPad ) = key_mask ) then
808                 cpu_REG( to_integer( unsigned ( x ) ) ) <= "0000" & key_num;
809                 current_state <= fetchA;
810             end if;
811         when O_LD_DT_X =>
812             cpu_state <= x"2C";
813             memAddress <= DT_ADDRESS;
814             tmp_mem_write <= '1';
815             mem_hold <= '0';
816             mem_ret_state <= fetchA;
817             memWrite <= cpu_REG( to_integer( unsigned ( x ) ) );
818             current_state <= memA;
819         when O_LD_ST_X =>
820             cpu_state <= x"2D";
821             memAddress <= ST_ADDRESS;
822             tmp_mem_write <= '1';
823             mem_hold <= '0';
824             mem_ret_state <= fetchA;
825             memWrite <= cpu_REG( to_integer( unsigned ( x ) ) );
826             current_state <= memA;
827         when O_ADD_I_X =>
828             cpu_state <= x"2E";
829             I <= I +  ( "0000" & cpu_REG(to_integer(unsigned( x ))) );
830             current_state <= fetchA;
831         when O_LD_F_X =>
832             cpu_state <= x"2F";
833             tmp_4 := cpu_REG(to_integer(unsigned( x )))( 3 downto 0 );
834             I <= hex_digits( to_integer(unsigned( tmp_4 )));
835             current_state <= fetchA;
836         when O_LD_B_X =>
837             cpu_state <= x"30";
838             BCD_left <= cpu_REG(to_integer(unsigned( x )));
839             BCD_total <= "00000000";
840             current_state <= BCD_hundreds;
841         when BCD_hundreds =>
842             cpu_state <= x"31";
843             if ( BCD_left < "01100100" ) then -- if < 100
844                 memAddress <= I;
845                 tmp_mem_write <= '1';
846                 mem_hold <= '0';
847                 mem_ret_state <= BCD_tens;
848                 memWrite <= BCD_total;
```

```
849                 BCD_total <= "00000000";
850                 current_state <= memA;
851             else
852                 BCD_left <= BCD_left - "01100100"; -- BCD_left -= 100
853                 BCD_total <= BCD_total + "00000001"; -- total++
854             end if;
855         when BCD_tens =>
856             cpu_state <= x"32";
857             if ( BCD_left < "00001010" ) then -- if < 10
858                 memAddress <= I + "000000000001";
859                 mem_ret_state <= BCD_ones;
860                 memWrite <= BCD_total;
861                 BCD_total <= "00000000";
862                 current_state <= memA;
863             else
864                 BCD_left <= BCD_left - "00001010"; -- BCD_left -= 10
865                 BCD_total <= BCD_total + "00000001"; -- total++
866             end if;
867         when BCD_ones =>
868             cpu_state <= x"33";
869             if ( BCD_left = "00000000" ) then
870                 memAddress <= I + "000000000010";
871                 mem_ret_state <= fetchA;
872                 memWrite <= BCD_total;
873                 BCD_total <= "00000000";
874                 current_state <= memA;
875             else
876                 BCD_left <= BCD_left - "00000001"; -- BCD_left -= 1
877                 BCD_total <= BCD_total + "00000001"; -- total++
878             end if;
879         when O_LD_I_X =>
880             cpu_state <= x"34";
881             memAddress <= I + ( "00000000" & x );
882             tmp_mem_write <= '1';
883             mem_hold <= '0';
884             memWrite <= cpu_REG(to_integer(unsigned( x )));
885             x <= x - "0001";
886             current_state <= memA;
887             if( x = "0000" ) then
888                 mem_ret_state <= fetchA;
889             else
890                 mem_ret_state <= O_LD_I_X;
891             end if;
892         when O_LD_X_I =>
893             cpu_state <= x"35";
894             reg_copy_num <= "0000";
895             multi_address <= I + "000000000001";
896             memAddress <= I;
897             tmp_mem_write <= '0';
898             mem_hold <= '0';
```

```
899             mem_ret_state <= pull_reg;
900             current_state <= memA;
901         when pull_reg =>
902             cpu_state <= x"36";
903             cpu_REG( to_integer( unsigned ( reg_copy_num ) ) ) <= mem_ret_data;
904             memAddress <= multi_address;
905             multi_address <= multi_address + "000000000001";
906             reg_copy_num <= reg_copy_num + "0001";
907             if ( reg_copy_num = x ) then
908                 current_state <= fetchA;
909             else
910                 mem_ret_state <= pull_reg;
911                 current_state <= memA;
912             end if;
913         when others =>
914             cpu_state <= x"37";
915             tmp_err_code <= INVALID_OP;
916                     current_state <= error;
917             end case;
918         end if;
919     end process;
920 end Behavioral;
```

## 6.3 Memory Controller

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity mem_controller is
    Port ( memAddress : in STD_LOGIC_VECTOR (35 downto 0);
           dataIn : in STD_LOGIC_VECTOR (23 downto 0);
           dataOut : out STD_LOGIC_VECTOR (23 downto 0);
           valid : in STD_LOGIC_VECTOR (2 downto 0);
           done : out STD_LOGIC_VECTOR (2 downto 0);
           write : in STD_LOGIC_VECTOR (2 downto 0);
           hold : in STD_LOGIC_VECTOR (2 downto 0);
           gameSelect : in STD_LOGIC_VECTOR (3 downto 0);
           gameSelected : out STD_LOGIC_VECTOR (3 downto 0);
           mapped_out : out STD_LOGIC_VECTOR( 7 downto 0 );
           mem_state : out STD_LOGIC_VECTOR( 7 downto 0 );
           debug_read_data : out STD_LOGIC_VECTOR( 7 downto 0 );
           sys_reset : out STD_LOGIC;
           cpu_reset : out STD_LOGIC;
           active : out STD_LOGIC;
           step : in STD_LOGIC;
           clk, reset : in STD_LOGIC );
end mem_controller;

architecture Behavioral of mem_controller is
    signal s_address : STD_LOGIC_VECTOR (16 downto 0);
    signal s_clock : STD_LOGIC;
    signal s_we : STD_LOGIC;
    signal s_dataIn : STD_LOGIC_VECTOR (7 downto 0);
    signal s_dataOut : STD_LOGIC_VECTOR (7 downto 0);

    component mem_mod
        port( address : in STD_LOGIC_VECTOR (12 downto 0);
              clock : in STD_LOGIC;
              we : in STD_LOGIC;
              dataIn : in STD_LOGIC_VECTOR (7 downto 0);
              dataOut : out STD_LOGIC_VECTOR (7 downto 0));
    end component;

    signal s_mem_clock : STD_LOGIC;
    signal s_mem_we : STD_LOGIC;
    signal s_mem_dataOut : STD_LOGIC_VECTOR (7 downto 0);

    component game_rom
        port( address : in STD_LOGIC_VECTOR (15 downto 0);
              clock : in STD_LOGIC;
              we : in STD_LOGIC;
```

```
49              dataIn : in STD_LOGIC_VECTOR (7 downto 0);
50              dataOut : out STD_LOGIC_VECTOR (7 downto 0));
51     end component;
52
53     signal s_rom_clock : STD_LOGIC;
54     signal s_rom_we : STD_LOGIC;
55     signal s_rom_dataOut : STD_LOGIC_VECTOR (7 downto 0);
56
57     type memState is ( initA, initB, initC, waiting, ownedWaiting, copy_key_map,
58                        copy_init_low, copy_init_high, copy_read, copy_write,
59                        normMemAccess, memAccess, memClock, memGet,
60                        normMemGet, raiseDone, waitFinish, pad1, pad2 );
61
62     signal currentGame : STD_LOGIC_VECTOR (3 downto 0);
63     signal owner : STD_LOGIC_VECTOR (2 downto 0);
64     signal current_state, memReturn, copy_return : memState;
65
66     signal copy_start : STD_LOGIC_VECTOR (16 downto 0);
67     signal copy_end : STD_LOGIC_VECTOR (16 downto 0);
68     signal copy_to : STD_LOGIC_VECTOR (16 downto 0);
69
70     signal step_active : STD_LOGIC;
71     signal init_times : STD_LOGIC_VECTOR (7 downto 0);
72 begin
73
74     MEM : mem_mod
75         port map ( address => s_address( 12 downto 0 ),
76                    clock => s_mem_clock,
77                    we => s_mem_we,
78                    dataIn => s_dataIn,
79                    dataOut => s_mem_dataOut);
80
81     ROM : game_rom
82        port map ( address => s_address( 15 downto 0 ),
83                   clock => s_rom_clock,
84                   we => s_rom_we,
85                   dataIn => s_dataIn,
86                   dataOut => s_rom_dataOut);
87
88     gameSelected <= currentGame;
89
90     process( clk, reset )
91         variable cur_address : STD_LOGIC_VECTOR (11 downto 0);
92         variable data : STD_LOGIC_VECTOR (7 downto 0);
93         variable dataA : STD_LOGIC_VECTOR (7 downto 0);
94     begin
95         if ( reset = '1' ) then
96             current_state <= initA;
97             sys_reset <= '1';
98             -- cpu_reset <= '1'; -- simulation
```

```
99              cpu_reset <= '0'; -- board
100             init_times <= x"00";
101             mapped_out <= x"AA";
102             active <= '0';
103         elsif ( rising_edge( clk ) ) then
104             current_state <= current_state;
105             case current_state is
106                 when initA =>
107                     mem_state <= x"01";
108                     active <= '1';
109                     done <= "000";
110                     current_state <= initB;
111                     s_address <= "0000000000000000";
112                     s_clock <= '0';
113                     s_we <= '0';
114                     s_dataIn <= "00000000";
115
116                     s_mem_clock <= '0';
117                     s_mem_we <= '0';
118
119                     s_rom_clock <= '0';
120                     s_rom_we <= '0';
121
122                     step_active <= '0';
123
124                     init_times <= init_times + x"01";
125                     mapped_out <= init_times;
126                 when initB =>
127                     mem_state <= x"02";
128                     currentGame <= gameSelect;
129                     current_state <= copy_init_low;
130                     dataOut <= x"000000";
131                 when copy_init_low =>
132                     mem_state <= x"03";
133                     copy_start <= '1' & x"0000";
134                     copy_end <= '1' & x"01FF";
135                     copy_to <= '1' & x"1000";
136                     copy_return <= copy_key_map;
137                     current_state <= copy_read;
138                     --step_active <= '1'; -- for debugging memory early
139                 when copy_key_map =>
140                     copy_start <= '0' & currentGame & x"080";
141                     copy_end <= '0' & currentGame & x"087";
142                     copy_to <= '1' & x"1080";
143                     copy_return <= copy_init_high;
144                     current_state <= copy_read;
145                 when copy_init_high =>
146                     mem_state <= x"04";
147                     copy_start <= '0' & currentGame & x"200";
148                     copy_end <= '0' & currentGame & x"FFF";
```

```
149                     copy_to <= '1' & x"1200";
150                     copy_return <= initC;
151                     current_state <= copy_read;
152                     --step_active <= '1'; -- for debugging rom early
153                 when copy_read =>
154                     mem_state <= x"05";
155                     s_address <= copy_start;
156                     s_we <= '0';
157                     memReturn <= copy_write;
158                     current_state <= memAccess;
159                 when copy_write =>
160                     mem_state <= x"06";
161                     s_address <= copy_to;
162                     s_dataIn <= s_dataOut;
163                     s_we <= '1';
164 
165                     copy_start <= copy_start + ( '0' & x"0001");
166                     copy_to <= copy_to + ( '0' & x"0001");
167 
168                     current_state <= memAccess;
169                     if( copy_start = copy_end) then
170                         memReturn <= copy_return;
171                     else
172                         memReturn <= copy_read;
173                     end if;
174                 when initC =>
175                     mem_state <= x"07";
176                     owner <= "000";
177                     current_state <= waiting;
178                     sys_reset <= '0';
179                     -- cpu_reset <= '0'; -- simulation
180                     cpu_reset <= '1'; -- board
181                     -- step_active <= '1'; -- for debugging normal memory
182                 when waiting =>
183                     mem_state <= x"08";
184                     if ( valid(0) = '1' ) then
185                         owner <= "001";
186                         current_state <= normMemAccess;
187                     elsif ( valid(1) = '1' ) then
188                         owner <= "010";
189                         current_state <= normMemAccess;
190                     elsif ( valid(2) = '1' ) then
191                         owner <= "100";
192                         current_state <= normMemAccess;
193                     end if;
194                 when ownedWaiting =>
195                     mem_state <= x"09";
196                     if ( ( owner and hold ) = "000" ) then
197                         owner <= "000";
198                         current_state <= waiting;
```

```
199                     elsif ( ( owner and valid ) = owner ) then
200                         current_state <= normMemAccess;
201                     end if;
202                 when normMemAccess =>
203                     mem_state <= x"0A";
204                     if ( owner = "001" ) then
205                         s_address <= '1' & "0001" & memAddress( 11 downto 0 );
206                         s_dataIn <= dataIn( 7 downto 0 );
207                         s_we <= write(0);
208                     elsif ( owner = "010" ) then
209                         s_address <= '1' & "0001" & memAddress( 23 downto 12 );
210                         s_dataIn <= dataIn( 15 downto 8 );
211                         s_we <= write(1);
212                     elsif ( owner = "100" ) then
213                         s_address <= '1' & "0001" & memAddress( 35 downto 24 );
214                         s_dataIn <= dataIn( 23 downto 16 );
215                         s_we <= write(2);
216                     end if;
217
218                     memReturn <= normMemGet;
219                     current_state <= memAccess;
220                 when memAccess =>
221                     mem_state <= x"0B";
222                     if( s_address(16) = '1' ) then
223                         s_mem_clock <= '0';
224                         s_mem_we <= s_we;
225                         if( ( s_address( 15 downto 0 ) = x"11FF" )
226                                         and ( s_we = '1' ) ) then
227                             mapped_out <= s_dataIn;
228                         end if;
229                     else
230                         s_rom_clock <= '0';
231                         s_rom_we <= s_we;
232                     end if;
233
234                     current_state <= memClock;
235                 when memClock =>
236                     mem_state <= x"0C";
237                     if( s_address(16) = '1' ) then
238                         s_mem_clock <= '1';
239                     else
240                         s_rom_clock <= '1';
241                     end if;
242
243                     current_state <= memGet;
244                 when memGet =>
245                     mem_state <= x"0D";
246                     if( s_address(16) = '1' ) then
247                         s_dataOut <= s_mem_dataOut;
248                         debug_read_data <= s_mem_dataOut;
```

```
249                     else
250                         s_dataOut <= s_rom_dataOut;
251                         debug_read_data <= s_rom_dataOut;
252                     end if;
253 
254                     if( step_active = '1' ) then
255                         current_state <= pad1;
256                     else
257                         current_state <= memReturn;
258                     end if;
259                 when pad1 =>
260                     if( step_active = '1' and step = '0' ) then
261                         current_state <= pad1;
262                     else
263                         current_state <= pad2;
264                     end if;
265                 when pad2 =>
266                     if( step_active = '1' and step = '1' ) then
267                         current_state <= pad2;
268                     else
269                         current_state <= memReturn;
270                     end if;
271                 when normMemGet =>
272                     mem_state <= x"0E";
273                     if ( owner = "001" ) then
274                         dataOut( 7 downto 0 ) <= s_dataOut;
275                     elsif ( owner = "010" ) then
276                         dataOut( 15 downto 8 ) <= s_dataOut;
277                     elsif ( owner = "100" ) then
278                         dataOut( 23 downto 16 ) <= s_dataOut;
279                     end if;
280 
281                     current_state <= raiseDone;
282                 when raiseDone =>
283                     mem_state <= x"0F";
284                     done <= owner;
285                     current_state <= waitFinish;
286                 when waitFinish =>
287                     mem_state <= x"10";
288                     if ( ( owner and valid ) = "000") then
289                         done <= "000";
290                         if ( ( owner and hold ) = owner ) then
291                             current_state <= ownedWaiting;
292                         else
293                             owner <= "000";
294                             current_state <= waiting;
295                         end if;
296                     end if;
297                 when others =>
298                     mapped_out <= x"0F";
```

```
299             end case;
300         end if;
301     end process;
302 
303 end Behavioral;
```

## 6.4 VGA Controller

```
-- Adapted from Albert Fazakas who adapted from Alec Wyen and Mihaita Nagy
-- VGA controller sample demo
-- Copyright 2014 Digilent, Inc.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vga_controller is
    Port ( memRead : in STD_LOGIC_VECTOR (7 downto 0);
           memWrite : out STD_LOGIC_VECTOR (7 downto 0);
           memAddress : out STD_LOGIC_VECTOR (11 downto 0);
           mem_valid : out STD_LOGIC;
           mem_write : out STD_LOGIC;
           mem_hold : out STD_LOGIC;
           mem_done : in STD_LOGIC;
           active : out STD_LOGIC;
           clk : in STD_LOGIC;
           sys_clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           sys_reset : in STD_LOGIC;

           VGA_HS_O : out  STD_LOGIC;
           VGA_VS_O : out  STD_LOGIC;
           VGA_RED_O    : out  STD_LOGIC_VECTOR (3 downto 0);
           VGA_GREEN_O  : out  STD_LOGIC_VECTOR (3 downto 0);
           VGA_BLUE_O   : out  STD_LOGIC_VECTOR (3 downto 0)

           );
end vga_controller;

architecture Behavioral of vga_controller is

signal requestLine : STD_LOGIC_VECTOR (7 downto 0);
signal previousRequestBuffer : STD_LOGIC_VECTOR( 0 downto 0 );
signal requestBuffer : STD_LOGIC_VECTOR( 0 downto 0 ) := "0";

type VBUFF is array( 1 downto 0 ) of STD_LOGIC_VECTOR (63 downto 0);
signal vga_VBUFF : VBUFF := (others => '0');

----------------------------------------------------------------

-- Constants for various VGA Resolutions

----------------------------------------------------------------

--***640x480@60Hz***--
```

```
49 constant FRAME_WIDTH : natural := 640;
50 constant FRAME_HEIGHT : natural := 480;
51
52 constant H_FP : natural := 16; --H front porch width (pixels)
53 constant H_PW : natural := 96; --H sync pulse width (pixels)
54 constant H_MAX : natural := 800; --H total period (pixels)
55 --
56 constant V_FP : natural := 10; --V front porch width (lines)
57 constant V_PW : natural := 2; --V sync pulse width (lines)
58 constant V_MAX : natural := 525; --V total period (lines)
59
60 constant H_POL : std_logic := '0';
61 constant V_POL : std_logic := '0';
62
63
64 ------------------------------------------------------------------------
65
66 -- Signal Declarations
67
68 ------------------------------------------------------------------------
69
70
71 ------------------------------------------------------------------------
72
73 -- VGA Controller specific signals: Counters, Sync, R, G, B
74
75 ------------------------------------------------------------------------
76 -- Pixel clock, in this case 25 MHz
77 signal pxl_clk : std_logic := '0';
78
79 -- Horizontal and Vertical counters
80 signal h_cntr_reg : std_logic_vector(11 downto 0) := (others =>'0');
81 signal v_cntr_reg : std_logic_vector(11 downto 0) := (others =>'0');
82
83 -- Pipe Horizontal and Vertical Counters
84 signal h_cntr_reg_dly   : std_logic_vector(11 downto 0) := (others => '0');
85 signal v_cntr_reg_dly   : std_logic_vector(11 downto 0) := (others => '0');
86
87 -- Horizontal and Vertical Sync
88 signal h_sync_reg : std_logic := not(H_POL);
89 signal v_sync_reg : std_logic := not(V_POL);
90 -- Pipe Horizontal and Vertical Sync
91 signal h_sync_reg_dly : std_logic := not(H_POL);
92 signal v_sync_reg_dly : std_logic :=  not(V_POL);
93
94 -- VGA R, G and B signals coming from the main multiplexers
95 signal vga_red_cmb   : std_logic_vector(3 downto 0);
96 signal vga_green_cmb : std_logic_vector(3 downto 0);
97 signal vga_blue_cmb  : std_logic_vector(3 downto 0);
98 --The main VGA R, G and B signals, validated by active
```

```
signal vga_red    : std_logic_vector(3 downto 0);
signal vga_green  : std_logic_vector(3 downto 0);
signal vga_blue   : std_logic_vector(3 downto 0);

signal vga_red_reg    : std_logic_vector(3 downto 0);
signal vga_green_reg  : std_logic_vector(3 downto 0);
signal vga_blue_reg   : std_logic_vector(3 downto 0);

signal tmp_mem_write : std_logic;
signal mem_ret_data : std_logic_vector(7 downto 0);

signal mhz50 : std_logic := '0';

type state is ( waiting, get0, get1, get2, get3,
                get4, get5, get6, get7, memA, memB );
signal mem_ret_state, current_state : state;

begin

mem_write <= tmp_mem_write;

    process( sys_clk, sys_reset )
    begin
        if( sys_reset = '1' ) then
            active <= '0';
            memWrite <= ( others => '0' );
            memAddress <= ( others => '0' );
            mem_valid <= '0';
            tmp_mem_write <= '0';
            mem_hold <= '0';
            current_state <= waiting;
            mem_ret_state <= waiting;
        elsif ( rising_edge( sys_clk ) ) then
          current_state <= current_state;
          case current_state is
          when waiting =>
          if( previousRequestBuffer /= requestBuffer ) then
                previousRequestBuffer <= requestBuffer;
                memAddress <= x"F" & requestLine( 4 downto 0 ) & "000";
                tmp_mem_write <= '0';
                mem_hold <= '1';
                mem_ret_state <= get0;
                current_state <= memA;
            end if;
         when get0 =>
            vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 7 downto 0 )
                                                              <= mem_ret_data;
            memAddress <= x"F" & requestLine( 4 downto 0 ) & "001";
            mem_ret_state <= get1;
            current_state <= memA;
```

```
149             when get1 =>
150                 vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 15 downto 8 )
151                                                                     <= mem_ret_data;
152                 memAddress <= x"F" & requestLine( 4 downto 0 ) & "010";
153                 mem_ret_state <= get2;
154                 current_state <= memA;
155             when get2 =>
156                 vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 23 downto 16 )
157                                                                       <= mem_ret_data;
158                 memAddress <= x"F" & requestLine( 4 downto 0 ) & "011";
159                 mem_ret_state <= get3;
160                 current_state <= memA;
161             when get3 =>
162                 vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 31 downto 24 )
163                                                                       <= mem_ret_data;
164                 memAddress <= x"F" & requestLine( 4 downto 0 ) & "100";
165                 mem_ret_state <= get4;
166                 current_state <= memA;
167             when get4 =>
168                 vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 39 downto 32 )
169                                                                       <= mem_ret_data;
170                 memAddress <= x"F" & requestLine( 4 downto 0 ) & "101";
171                 mem_ret_state <= get5;
172                 current_state <= memA;
173             when get5 =>
174                 vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 47 downto 40 )
175                                                                       <= mem_ret_data;
176                 memAddress <= x"F" & requestLine( 4 downto 0 ) & "110";
177                 mem_ret_state <= get6;
178                 current_state <= memA;
179             when get6 =>
180                 vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 55 downto 48 )
181                                                                     <= mem_ret_data;
182                 memAddress <= x"F" & requestLine( 4 downto 0 ) & "111";
183                 mem_ret_state <= get7;
184                 current_state <= memA;
185             when get7 =>
186                 vga_VBUFF(to_integer( unsigned ( requestBuffer ) ))( 63 downto 56 )
187                                                                     <= mem_ret_data;
188                 mem_hold <= '0';
189                 current_state <= waiting;
190             when memA =>
191                 if ( mem_done = '0' ) then
192                     mem_valid <= '1';
193                     current_state <= memB;
194                 end if;
195             when memB =>
196                 if( mem_done = '1' ) then
197                     if ( tmp_mem_write = '0' ) then
198                         mem_ret_data <= memRead;
```

```
                end if;

                mem_valid <= '0';
                current_state <= mem_ret_state;
            end if;
      nd case;
        end if;
    end process;


process( clk )
begin
    if( rising_edge( clk )) then
        mhz50 <= not mhz50;
    end if;
end process;

process( mhz50 )
begin
    if( rising_edge( mhz50 )) then
        pxl_clk <= not pxl_clk;
    end if;
end process;


-----------------------------------------------------------------

-- Generate Horizontal, Vertical counters and the Sync signals

-----------------------------------------------------------------
  -- Horizontal counter
  process (pxl_clk)
  begin
    if (rising_edge(pxl_clk)) then
      if (h_cntr_reg = (H_MAX - 1)) then
        h_cntr_reg <= (others =>'0');
      else
        h_cntr_reg <= h_cntr_reg + 1;
      end if;
    end if;
  end process;
  -- Vertical counter
  process (pxl_clk)
  begin
    if (rising_edge(pxl_clk)) then
      if ((h_cntr_reg = (H_MAX - 1)) and (v_cntr_reg = (V_MAX - 1))) then
        v_cntr_reg <= (others =>'0');
      elsif (h_cntr_reg = (H_MAX - 1)) then
        v_cntr_reg <= v_cntr_reg + 1;
      end if;
```

```
249       end if;
250     end process;
251     -- Horizontal sync
252     process (pxl_clk)
253     begin
254       if (rising_edge(pxl_clk)) then
255         if (h_cntr_reg >= (H_FP + FRAME_WIDTH - 1)) and
256             (h_cntr_reg < (H_FP + FRAME_WIDTH + H_PW - 1)) then
257           h_sync_reg <= H_POL;
258         else
259           h_sync_reg <= not(H_POL);
260         end if;
261       end if;
262     end process;
263     -- Vertical sync
264     process (pxl_clk)
265     begin
266       if (rising_edge(pxl_clk)) then
267         if (v_cntr_reg >= (V_FP + FRAME_HEIGHT - 1)) and
268           (v_cntr_reg < (V_FP + FRAME_HEIGHT + V_PW - 1)) then
269           v_sync_reg <= V_POL;
270         else
271           v_sync_reg <= not(V_POL);
272         end if;
273       end if;
274     end process;
275
276
277     process( pxl_clk )
278       variable xspot : std_logic_vector(11 downto 0);
279       variable yspot : std_logic_vector(11 downto 0);
280       variable which_buf : std_logic_vector( 0 downto 0 );
281     begin
282       which_buf := not requestBuffer;
283       vga_red <= "0000";
284       vga_blue <= "0000";
285       vga_green <= "0000";
286       if( rising_edge(pxl_clk) ) then
287           if ( h_cntr_reg >= (H_FP + 64)) and
288                       (h_cntr_reg < (H_FP + FRAME_WIDTH - 64) ) then
289               if( v_cntr_reg >= (V_FP + 112)) and
290                       (v_cntr_reg < (V_FP + FRAME_HEIGHT - 112)) then
291                   xspot := h_cntr_reg - ( H_FP + 64 );
292                   yspot := v_cntr_reg - ( V_FP + 112 );
293                   if( ( xspot = x"000" ) and ( yspot( 2 downto 0 ) = "000") ) then
294                       requestBuffer <= which_buf;
295                       which_buf := not which_buf;
296                       yspot := "0000000" & yspot( 7 downto 3 );
297                       yspot := yspot + 1;
298                       requestLine <= "000" & yspot( 4 downto 0 );
```

```
299                 end if;
300
301                 if( vga_VBUFF( to_integer( unsigned ( which_buf ) ))
302                   ( to_integer( unsigned ( xspot( 11 downto 3 ) ) )) = '1' ) then
303                     vga_green <= "1111";
304                 end if;
305             end if;
306         end if;
307     end if;
308   end process;
309
310  vga_red_cmb <= vga_red;
311  vga_green_cmb <= vga_green;
312  vga_blue_cmb <= vga_blue;
313
314
315  -- Register Outputs
316   process (pxl_clk)
317   begin
318     if (rising_edge(pxl_clk)) then
319
320       v_sync_reg_dly <= v_sync_reg;
321       h_sync_reg_dly <= h_sync_reg;
322       vga_red_reg    <= vga_red_cmb;
323       vga_green_reg  <= vga_green_cmb;
324       vga_blue_reg   <= vga_blue_cmb;
325     end if;
326   end process;
327
328   -- Assign outputs
329   VGA_HS_O     <= h_sync_reg_dly;
330   VGA_VS_O     <= v_sync_reg_dly;
331   VGA_RED_O    <= vga_red_reg;
332   VGA_GREEN_O  <= vga_green_reg;
333   VGA_BLUE_O   <= vga_blue_reg;
334
335 end Behavioral;
```

## 6.5 Time Keeper

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity time_keeper is
    Port ( memRead : in STD_LOGIC_VECTOR (7 downto 0);
           memWrite : out STD_LOGIC_VECTOR (7 downto 0);
           memAddress : out STD_LOGIC_VECTOR (11 downto 0);
           mem_valid : out STD_LOGIC;
           mem_write : out STD_LOGIC;
           mem_hold : out STD_LOGIC;
           mem_done : in STD_LOGIC;
           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           heart_beat : out STD_LOGIC;
           buzz : out STD_LOGIC );
end time_keeper;

architecture Behavioral of time_keeper is
    constant DT_ADDRESS  : std_logic_vector(11 downto 0):= "000000000000";
    constant ST_ADDRESS  : std_logic_vector(11 downto 0):= "000000000001";

    type state is ( init, count, update, writeDT, readST, writeST, memA, memB );
    signal current_state : state;
    signal counter60hz : std_logic_vector( 23 downto 0 );
    -- close enough, only lets us know the timer is running about
    --      at the correct speed.
    signal counter1hz  : std_logic_vector( 6 downto 0 );

    signal mem_ret_state : state;
    signal tmp_mem_write : STD_LOGIC;
    signal mem_ret_data : STD_LOGIC_VECTOR (7 downto 0);
begin
    heart_beat <= counter1hz(6); -- for chip
    -- heart_beat <= counter1hz(0); -- for simulation
    mem_write <= tmp_mem_write;

    process( clk, reset )
        variable tmp_count : std_logic_vector( 23 downto 0 );
        variable tmp_8 : std_logic_vector( 7 downto 0 );
    begin
        if ( reset = '1' ) then
            current_state <= init;
            memWrite <= ( others => '0' );
            memAddress <= ( others => '0' );
            mem_valid <= '0';
            tmp_mem_write <= '0';
            mem_hold <= '0';
```

```
49              buzz <= '0';
50              counter1hz <= "1000000";
51          elsif ( rising_edge( clk ) ) then
52              current_state <= current_state;
53              case current_state is
54                  when init =>
55                      counter60hz <= ( others => '0' );
56                      current_state <= count;
57                  when count =>
58                      mem_hold <= '0';
59                      tmp_count := counter60hz;
60                      tmp_count := tmp_count + 1;
61                      if( tmp_count = "00000001101010101010101010" ) then -- for board divider
62                      --if( tmp_count = "00011010101010101010101010" ) then -- for board
63                      --if( tmp_count = "00000000000000010000000" ) then -- for test bench
64                          current_state <= update;
65                          tmp_count := ( others => '0' );
66                          counter1hz <= counter1hz + 1;
67                      end if;
68
69                      counter60hz <= tmp_count;
70                  when update =>
71                      memAddress <= DT_ADDRESS;
72                      tmp_mem_write <= '0';
73                      mem_hold <= '1';
74                      mem_ret_state <= writeDT;
75                      current_state <= memA;
76                  when writeDT =>
77                      tmp_8 := mem_ret_data;
78                      if ( tmp_8 /= "00000000" ) then
79                          tmp_8 := tmp_8 - "00000001";
80                      end if;
81
82                      memAddress <= DT_ADDRESS;
83                      tmp_mem_write <= '1';
84                      mem_ret_state <= readST;
85                      memWrite <= tmp_8;
86                      current_state <= memA;
87                  when readST =>
88                      memAddress <= ST_ADDRESS;
89                      tmp_mem_write <= '0';
90                      mem_ret_state <= writeST;
91                      current_state <= memA;
92                  when writeST =>
93                      tmp_8 := mem_ret_data;
94                      if ( tmp_8 /= "00000000" ) then
95                          tmp_8 := tmp_8 - "00000001";
96                          buzz <= '1';
97                      else
98                          buzz <= '0';
```

```
99                      end if;
100
101                     memAddress <= ST_ADDRESS;
102                     tmp_mem_write <= '1';
103                     mem_ret_state <= count;
104                     memWrite <= tmp_8;
105                     current_state <= memA;
106                 when memA =>
107                     if ( mem_done = '0' ) then
108                         mem_valid <= '1';
109                         current_state <= memB;
110                     end if;
111                 when memB =>
112                     if( mem_done = '1' ) then
113                         if ( tmp_mem_write = '0' ) then
114                             mem_ret_data <= memRead;
115                         end if;
116
117                         mem_valid <= '0';
118                         current_state <= mem_ret_state;
119                     end if;
120             end case;
121         end if;
122     end process;
123 end Behavioral;
```

## 6.6 RAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mem_mod is
    Port ( address : in STD_LOGIC_VECTOR (12 downto 0);
           clock : in STD_LOGIC;
           we : in STD_LOGIC;
           dataIn : in STD_LOGIC_VECTOR (7 downto 0);
           dataOut : out STD_LOGIC_VECTOR (7 downto 0));
end mem_mod;

architecture Behavioral of mem_mod is
    type RAM is array ( ( 2 * 4096 ) - 1 downto 0 ) of std_logic_vector( 7 downto 0 );

    signal sys_RAM : RAM := (
                                0 => x"00", 1 => x"00", -- set DT and ST to 0
                                511 => "10101010",           -- memory mapped port
                                 2 => "11110000",  22 => "10010000",  42 => "11110000",  62
                                 3 => "10010000",  23 => "10010000",  43 => "10010000",  63
                                 4 => "10010000",  24 => "11110000",  44 => "11110000",  64
                                 5 => "10010000",  25 => "00010000",  45 => "10010000",  65
                                 6 => "11110000",  26 => "00010000",  46 => "11110000",  66
                                 7 => "00100000",  27 => "11110000",  47 => "11110000",  67
                                 8 => "01100000",  28 => "10000000",  48 => "10010000",  68
                                 9 => "00100000",  29 => "11110000",  49 => "11110000",  69
                                10 => "00100000",  30 => "00010000",  50 => "00010000",  70
                                11 => "01110000",  31 => "11110000",  51 => "11110000",  71
                                12 => "11110000",  32 => "11110000",  52 => "11110000",  72
                                13 => "00010000",  33 => "10000000",  53 => "10010000",  73
                                14 => "11110000",  34 => "11110000",  54 => "11110000",  74
                                15 => "10000000",  35 => "10010000",  55 => "10010000",  75
                                16 => "11110000",  36 => "11110000",  56 => "10010000",  76
                                17 => "11110000",  37 => "11110000",  57 => "11100000",  77
                                18 => "00010000",  38 => "00010000",  58 => "10010000",  78
                                19 => "11110000",  39 => "00100000",  59 => "11100000",  79
                                20 => "00010000",  40 => "01000000",  60 => "10010000",  80
                                21 => "11110000",  41 => "01000000",  61 => "11100000",  81

                                others => ( others => '0')
                            );
    signal read_address : std_logic_vector( 12 downto 0 );
begin

    process ( clock )
    begin
        if ( rising_edge( clock ) ) then
            if( we = '1' ) then
```

```
49                  sys_RAM( to_integer( unsigned( address ))) <= dataIn;
50              end if;
51
52              read_address <= address;
53          end if;
54      end process;
55
56      dataOut <= sys_RAM( to_integer( unsigned( read_address )));
57  end Behavioral;
```

## 6.7 Rom

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity game_rom is
    Port ( address : in STD_LOGIC_VECTOR (15 downto 0);
           clock : in STD_LOGIC;
           we : in STD_LOGIC;
           dataIn : in STD_LOGIC_VECTOR (7 downto 0);
           dataOut : out STD_LOGIC_VECTOR (7 downto 0));
end game_rom;

architecture Behavioral of game_rom is
    type RAM is array ( ( 16 * 4096 ) - 1 downto 0 ) of std_logic_vector( 7 downto 0 );

    signal sys_RAM : RAM := (
            -- light travels left
            (0 + 136) => x"00", (0 + 137) => x"00", -- delay
            (0 + 138) => x"00",
            512 => x"66", 513 => x"01",  -- LD V6, 0x01
            514 => x"61", 515 => x"3C",  -- LD V1, 0x3C
            516 => x"F1", 517 => x"15",  -- LD DT, V1;   ; loop
            518 => x"86", 519 => x"6E",  -- SHL V6, V6
            520 => x"4F", 521 => x"01",  -- SNE VF, 0x01
            522 => x"66", 523 => x"01",  -- LD V6, 0x01
            524 => x"80", 525 => x"60",  -- LD V0, V6
            526 => x"A1", 527 => x"FF",  -- LD I, 0x1FF
            528 => x"F0", 529 => x"55",  -- LD [I], V0
            530 => x"F2", 531 => x"07",  -- LD V2, DT   ; wait
            532 => x"32", 533 => x"00",  -- SE V2, 0x00
            534 => x"12", 535 => x"12",  -- JP wait
            536 => x"12", 537 => x"04",  -- JP loop

            -- light travels right
            (4096 + 136) => x"00", (4096 + 137) => x"00", -- delay
            (4096 + 138) => x"00",
            (4096 + 512) => x"66", (4096 + 513) => x"80",  -- LD V6, 0x80
            (4096 + 514) => x"61", (4096 + 515) => x"3C",  -- LD V1, 0x3C
            (4096 + 516) => x"F1", (4096 + 517) => x"15",  -- LD DT, V1;   ; loop
            (4096 + 518) => x"86", (4096 + 519) => x"66",  -- SHR V6, V6
            (4096 + 520) => x"4F", (4096 + 521) => x"01",  -- SNE VF, 0x01
            (4096 + 522) => x"66", (4096 + 523) => x"80",  -- LD V6, 0x80
            (4096 + 524) => x"80", (4096 + 525) => x"60",  -- LD V0, V6
            (4096 + 526) => x"A1", (4096 + 527) => x"FF",  -- LD I, 0x1FF
            (4096 + 528) => x"F0", (4096 + 529) => x"55",  -- LD [I], V0
            (4096 + 530) => x"F2", (4096 + 531) => x"07",  -- LD V2, DT   ; wait
            (4096 + 532) => x"32", (4096 + 533) => x"00",  -- SE V2, 0x00
            (4096 + 534) => x"12", (4096 + 535) => x"12",  -- JP wait
```

```
        (4096 + 536) => x"12", (4096 + 537) => x"04",  -- JP loop

        -- random lights
        (8192 + 136) => x"00", (8192 + 137) => x"00", -- delay
        (8192 + 138) => x"00",
        (8192 + 512) => x"61", (8192 + 513) => x"3C",  -- LD V1, 0x3C
        (8192 + 514) => x"F1", (8192 + 515) => x"15",  -- LD DT, V1;   ; loop
        (8192 + 516) => x"C0", (8192 + 517) => x"FF",  -- RND V0, FF
        (8192 + 518) => x"A1", (8192 + 519) => x"FF",  -- LD I, 0x1FF
        (8192 + 520) => x"F0", (8192 + 521) => x"55",  -- LD [I], V0
        (8192 + 522) => x"F2", (8192 + 523) => x"07",  -- LD V2, DT   ; wait
        (8192 + 524) => x"32", (8192 + 525) => x"00",  -- SE V2, 0x00
        (8192 + 526) => x"12", (8192 + 527) => x"0A",  -- JP wait
        (8192 + 528) => x"12", (8192 + 529) => x"02",  -- JP loop

        -- Code adapted to show structure not content
        -- Would be too much to print in a report
        -- PONG by Paul Vervalin
        (12288 + 128) => x"0C", (12288 + 129) => x"00",
        . . .
        . . .

        -- Tetris by Fran Dachille
        (16384 + 128) => x"00", (16384 + 129) => x"F0",
        . . .
        . . .

        -- Blitz by David Winter
        (20480 + 128) => x"00", (20480 + 129) => x"00",
        . . .
        . . .

        -- Brix by Andre Gustafsson
        (24576 + 128) => x"00", (24576 + 129) => x"00",
        . . .
        . . .

        -- Cave by 199x
        (28672 + 128) => x"00", (28672 + 129) => x"B0",
        . . .
        . . .

        -- Hidden by David Winter
        (32768 + 128) => x"00", (32768 + 129) => x"B0",
        . . .
        . . .

        -- Kaleid by Joseph Weisbecker (RCA)
        (36864 + 128) => x"00", (36864 + 129) => x"B0",
        . . .
```

```
99              . . .
100
101             -- Merlin by David Winter
102             (40960 + 128) => x"00", (40960 + 129) => x"00",
103             . . .
104             . . .
105             -- Missile by David Winter
106             (45056 + 128) => x"0F", (45056 + 129) => x"00",
107             . . .
108             . . .
109
110             -- Puzzle by Joseph Weisbecker (RCA)
111             (49152 + 128) => x"00", (49152 + 129) => x"B0",
112             . . .
113             . . .
114
115             -- Tank by Joseph Weisbecker (RCA)
116             (53248 + 128) => x"00", (53248 + 129) => x"B0",
117             . . .
118             . . .
119
120             -- Vers by JMN
121             (57344 + 128) => x"03", (57344 + 129) => x"00",
122             . . .
123             . . .
124             others => ( others => '0')
125         );
126     signal read_address : std_logic_vector( 15 downto 0 );
127 begin
128
129     process ( clock )
130     begin
131         if ( rising_edge( clock ) ) then
132             if( we = '1' ) then
133                 sys_RAM( to_integer( unsigned( address ))) <= dataIn;
134             end if;
135
136             read_address <= address;
137         end if;
138     end process;
139
140     dataOut <= sys_RAM( to_integer( unsigned( read_address )));
141 end Behavioral;
```

# 7 Sources

## References


[1] RCA. *COSMAC VIP - Instruction Manual - 1978*

[2] Joseph Weisbecker. *An Easy Programming System* BYTE magazine, 12(3):108-122, 1978.

[3] Cowgod *CHIP-8 Technical Reference v1.0*
http://devernay.free.fr/hacks/chip8/C8TECH10.HTM

[4] Matthew Mikolay *Mastering CHIP-8*
http://mattmik.com/files/chip8/mastering/chip8.html

[5] Albert Fazakas Nexys-4 Sample VGA Controller
https://github.com/Digilent/Nexys4DDR/blob/master/Projects/User_Demo/src/hdl/Vga.vhd