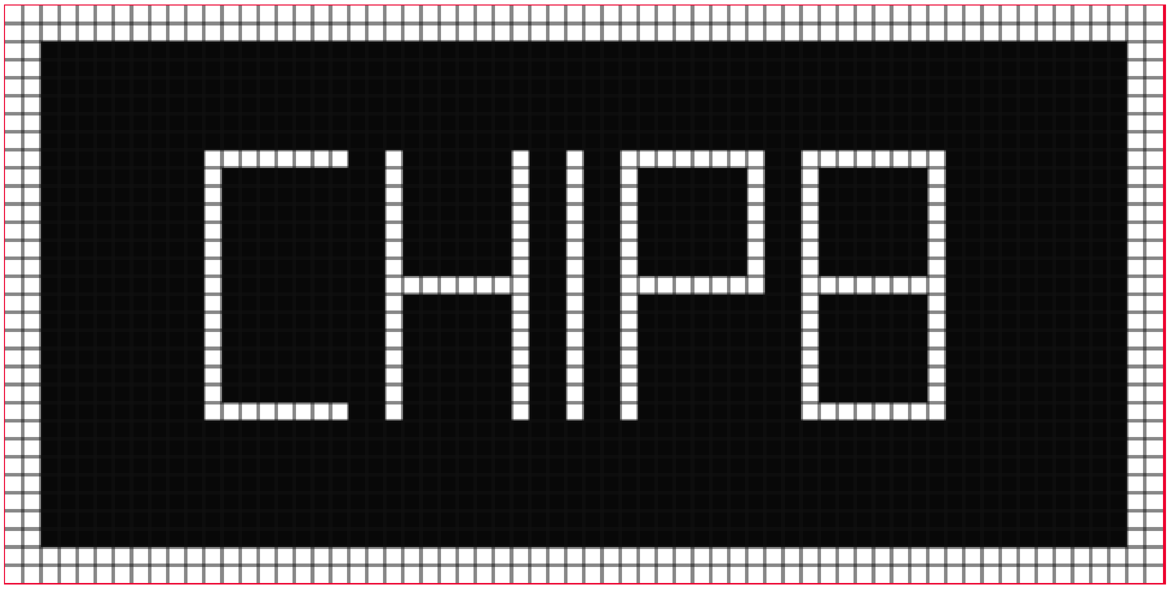


CHIP8 Emulator on the Intel DE1-SoC

Elysia Witham, Yuhang Zhu, Xin Gao, Daniel Indictor

Table of Contents

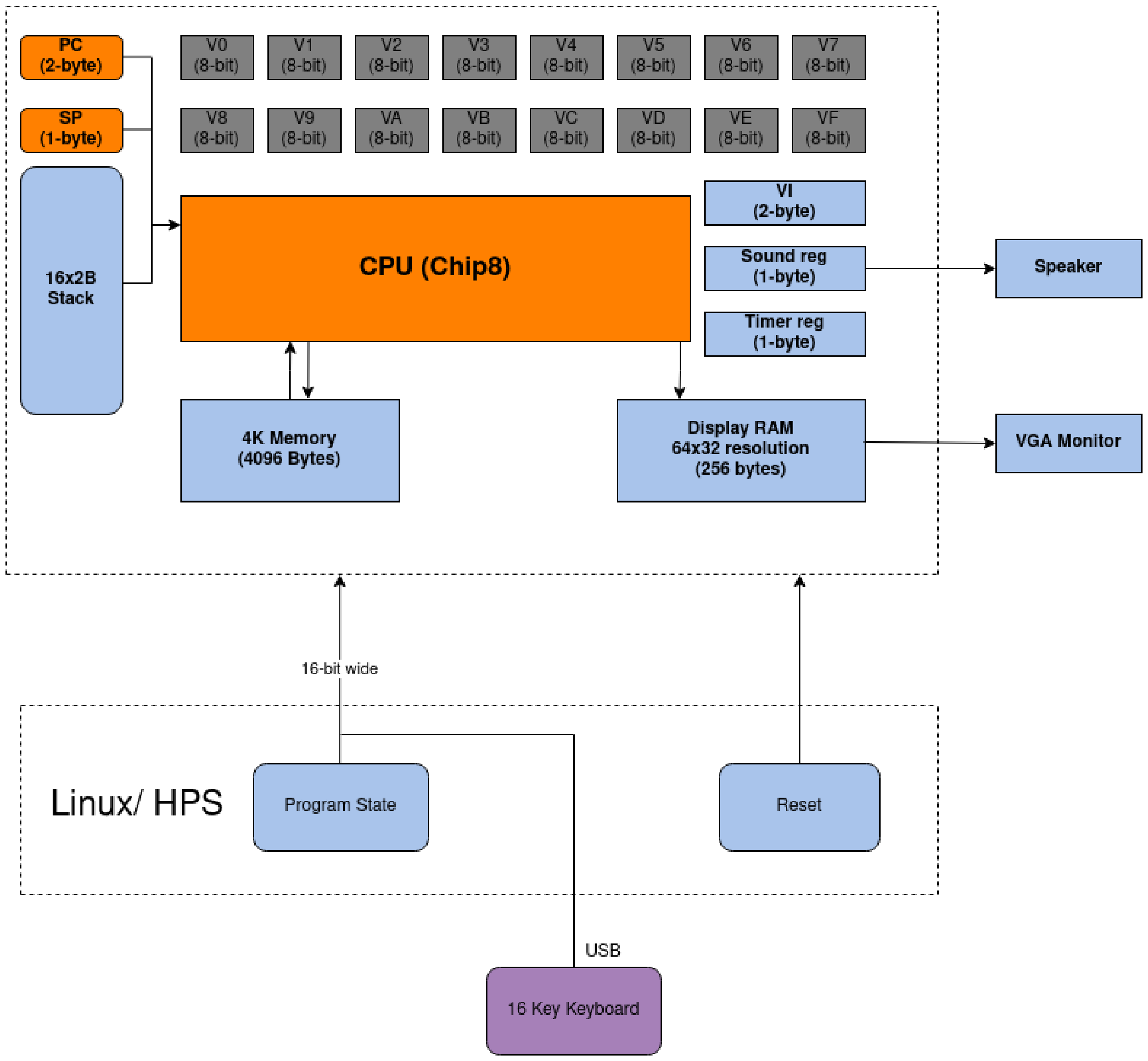
Introduction

CHIP-8 is a super simple interpreted programming language that offers programs a well-defined virtual machine. Many toy programs and games have already been written in this language, which supports a low-resolution display, a hexadecimal numberpad, and a rudimentary sound output. Nowadays, CHIP-8 is regarded as an easy first project for people getting into software emulation.

As far as programming languages go, CHIP-8 bytecode can be interpreted like Assembly (containing 35 opcodes total), making it an excellent candidate for hardware emulation due to its simplicity. As for software, we created a userspace application with a few pre-selected games which will be shown on the VGA display. Keyboard presses are captured by software, with the appropriate data being communicated to hardware. Once a game is selected via the keyboard, the hardware emulator will take over and a user will be able to play the selected game using the keyboard.

Resource Requirements

The resource requirements of a CHIP8 program are very light: 16 general purpose 1-byte registers, 4KB working memory, a program counter, a 256-byte display RAM, a 16-layer long call stack (and its associated stack pointer), and three special-purpose registers to provide timing and sound functionality. The block diagram below shows how we organize the software and hardware. The responsibility of the hard processor system is to interface with a QWERTY USB keyboard (and send it to CHIP8 in the form CHIP8 expects keyboards to be), as well as load games from the filesystem and send them to the CPU.



Hardware-Software Interface Design

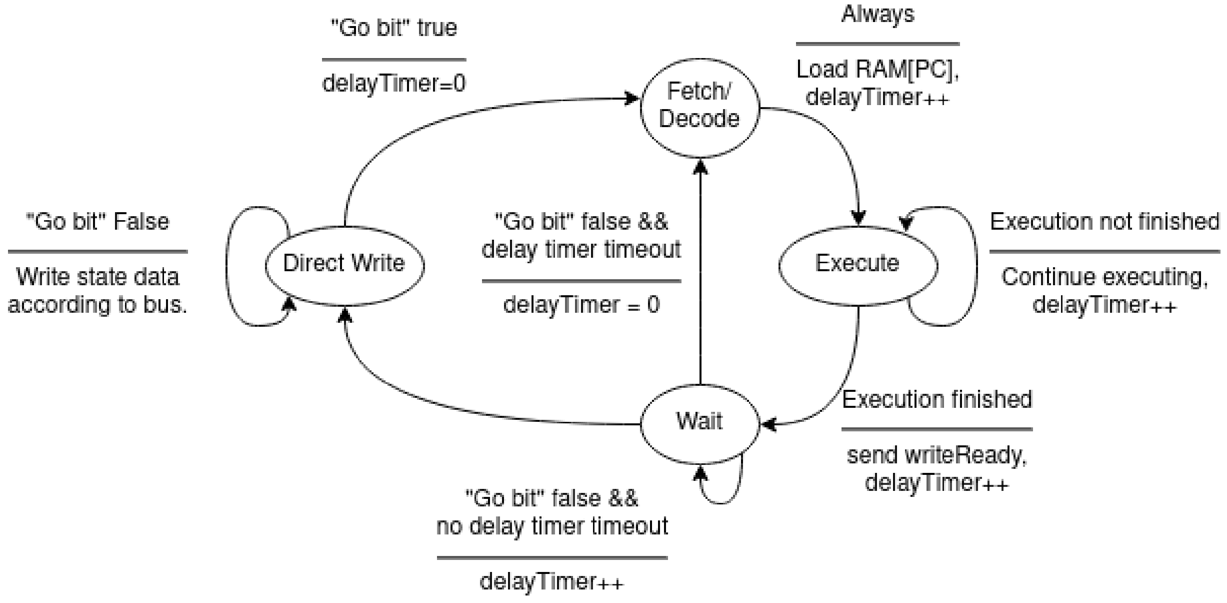
The general strategy to send gamedata and keypresses to the CHIP8 hardware is simple. As soon as the hardware detects any write to either the display or memory memories, it immediately stops what it's doing and lets the software write whatever data it needs, after which point the software will write to a special "control byte address" (0xFFF) to indicate that the hardware may proceed. Whenever the software writes to the display or memory rams, the cpu module (explained later) clears its internal state, including, for example, the general purpose registers. Note that since the write bus is two bytes long, we can send the entire keyboard state whenever the user presses a key as a bit-vector of length 16, corresponding to the 16 total keys on the keyboard.

Bus Address	End	(hex)	Start	(hex)	Size (bytes)
128x32B (4K) Memory	2047	(7FF)	0	(0)	4096
Framebuffer (256 bytes)	2175	(87F)	2048	(800)	256
Keyboard State	4094	(FFE)	4094	(FFE)	2
Control Byte	4095	(FFF)	4095	(FFF)	2

Note that as per the CHIP8 spec, all multi-byte values are MSB-first

Hardware Design

The behavior of the hardware is best described with the FSM below. Briefly, the hardware constantly fetches and executes instructions, and then waits for a timer period to expire (timer period explained later) before continuing to fetch the next instruction. At any point, a write to hardware ram ("Direct Write" on the diagram) may interrupt it and it will reset its internal non-RAM state in anticipation of new gamedata being written.


"Go bit" true
delayTimer=0
Always
Load RAM[PC], delayTimer++
Fetch/ Decode
"Go bit" False
Write state data according to bus.
Direct Write
"Go bit" false && delay timer timeout
delayTimer = 0
Execute
Execution not finished
Continue executing, delayTimer++
Wait
Execution finished
send writeReady, delayTimer++
"Go bit" false && no delay timer timeout
delayTimer++

Since the FPGA can execute programs far faster than the original CHIP-8 programs were meant to be run, we have to artificially slow down our program using a delay timer. This timer starts counting every time an instruction is fetched and counts up to some constant, "Delay Timer Max", preventing the execution of the next instruction until it finishes. The value of this maximum depends on the FPGA clock speed and the target instruction clock. For instance, if the FPGA is running at 50MHz and we're targeting 700 instructions per second, we would set our Delay Timer Max to about 71428.

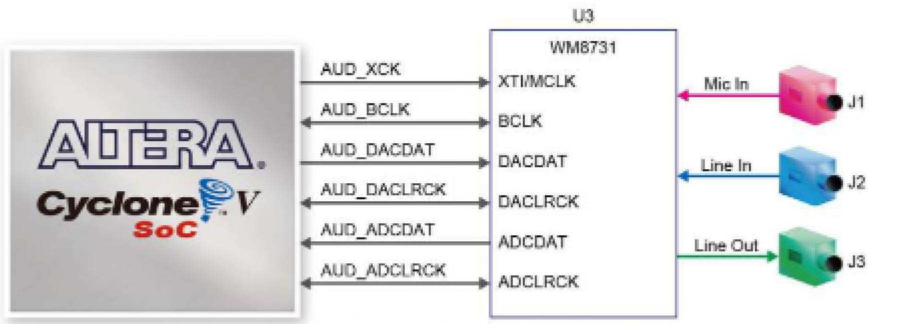
Hardware Organization

We organized our code into a number of modules:

- display_ram: A simple 256-byte two-port read-write RAM to be read by the display module and written by either the software driver or the CPU module.
- chip_ram: A 4096-byte two-port read-write RAM to be read by the CPU and written by the CPU or the software driver.
- display: This module does the job of reading data from the display ram and scaling/fitting the 64x32 pixel display native to CHIP8 to a 640x480 pixel VGA display. It thus also handles setting VGA protocol signals.
- cpu: This module is incomplete (as you shall see) but in theory it should house the finite state machine given earlier and a huge switch statement to handle the opcodes.
- sound: This module is complete (but not integrated). In theory, it should just make the FPGA make a sound on its AUX port given the signal to do so.
- yachip: This is the top-level module (completed but not tested) that delegates access to the display and chip8 rams to the CPU and display module. As discussed, it gives priority to the software driver.
- timer: Timers are necessary in three parts in our code: first, for two special purpose registers which, when set to a non-zero value tick down to 0 at 60Hz, and another to slow down the CPU.

Sound Implementation Details

On the DE1-SoC, there is a device called Wolfson WM8731 which has two 24 bits ADC, two 24 bits DAC and a headphone amplifier and supports 8-96 kHz sampling rates. The WM8731 is controlled via a serial I2C bus, which is connected to HPS or Cyclone V SoC FPGA through an I2C multiplexer. The connection of the circuit is shown as follows.

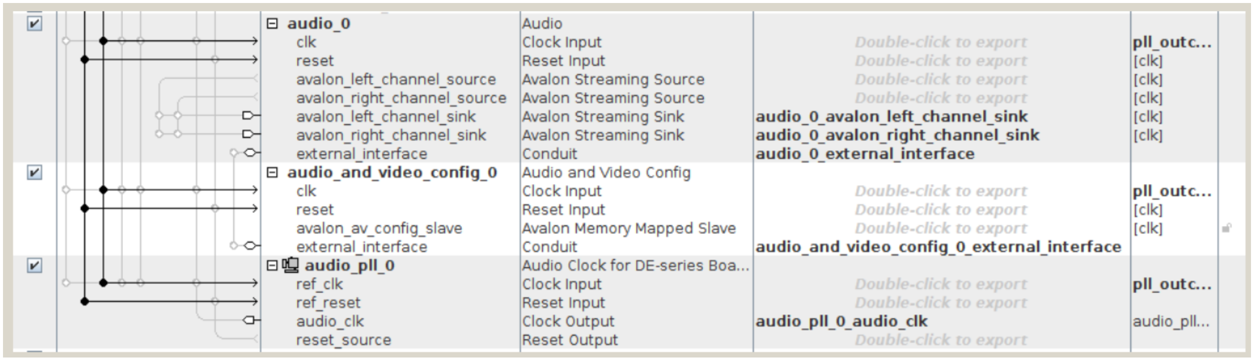


Pin Assignment of Audio CODEC

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
AUD_ADCLRCK	PIN_K8	Audio CODEC ADC LR Clock	3.3V
AUD_ADCDAT	PIN_K7	Audio CODEC ADC Data	3.3V
AUD_DACLRCK	PIN_H8	Audio CODEC DAC LR Clock	3.3V
AUD_DACDAT	PIN_J7	Audio CODEC DAC Data	3.3V
AUD_XCK	PIN_G7	Audio CODEC Chip Clock	3.3V
AUD_BCLK	PIN_H7	Audio CODEC Bit-stream Clock	3.3V
I2C_SCLK	PIN_J12 or PIN_E23	I2C Clock	3.3V
I2C_SDAT	PIN_K12 or PIN_C24	I2C Data	3.3V

The following describes briefly how we got the sound working.

First, we found some audio effect files (.wav) from the website. After checking the bit rate, we used MATLAB to generate a .mig file. To make things clear, we combine VGA display and audio parts as a whole. The final qsys connection is shown in the figure below.



⊟ audio_0	Audio		
clk	Clock Input	Double-click to export	pll_outc...
reset	Reset Input	Double-click to export	[clk]
avalon_left_channel_source	Avalon Streaming Source	Double-click to export	[clk]
avalon_right_channel_source	Avalon Streaming Source	Double-click to export	[clk]
avalon_left_channel_sink	Avalon Streaming Sink	audio_0_avalon_left_channel_sink	[clk]
avalon_right_channel_sink	Avalon Streaming Sink	audio_0_avalon_right_channel_sink	[clk]
external_interface	Conduit	audio_0_external_interface	
⊟ audio_and_video_config_0	Audio and Video Config		
clk	Clock Input	Double-click to export	pll_outc...
reset	Reset Input	Double-click to export	[clk]
avalon_av_config_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]
external_interface	Conduit	audio_and_video_config_0_external_interface	
⊟ audio_pll_0	Audio Clock for DE-series Boa...		
ref_clk	Clock Input	Double-click to export	pll_outc...
ref_reset	Reset Input	Double-click to export	
audio_clk	Clock Output	audio_pll_0_audio_clk	audio_pll...
reset_source	Reset Output	Double-click to export	

Fig. Qsys connection for Audio parts

Reference: Audio tutorial
https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/optional-tonegen.html

Software Organization

We used a Linux kernel driver (written in C) to communicate with the hardware over the Avalon Bus (as a memory-mapped device). This kernel driver offers a special device, /dev/chip8, which may be ioctl'd by a userspace application (written in C++) to deliver information to the hardware.

The kernel driver closely mirrors the code given in lab 3 and as such we shall not expound upon it here.

The userspace application does the job of loading games from the filesystem when they are requested (via the GAME keys, number keys 5-9). The games are stored as .hex files and are loaded on-the-fly. Essentially, it resets all relevant memory and peripherals, and then loads roms based on which game is requested. In order to do so, it steps through the hex output of a rom and sends each byte one at a time to memory. Then it sends relevant key presses when the game is running to the processor as well. When the ESC key is pressed, it closes that game, resets the system (writing zeros to all rams), and waits until a new game is selected to start the process all over again.

We encapsulated the keyboard interface, which is provided via libusb, with a C++ class. It contains a struct for representing key presses, which holds values for game selection keys, up down left and right keys, the escape and enter keys, and the CHIP8 keypad. ESC is used to exit a game, while ENTER is used to select one. Aside from that, the numbers 5-9 have been reserved for game selection, and the CHIP8 keypad has been assigned to 1-4, q-r, a-f, and z-v

Keyboard Keymap
We opted to use a USB keyboard rather than a dedicated pin pad for CHIP8 inputs. Many online CHIP8 emulators support a keyboard with the same keys as the ones we are using mapped to the same inputs. The mapping is as follows:

1 2 3 4		1 2 3 C
q w e r		4 5 6 D
a s d f	-->	7 8 9 E
z x c v		A 0 B F

The left side represents a standard keyboard, with the right side representing what it is being translated as for the CHIP8 device.

Development Strategies
Since there are a lot of moving parts in this project, we decided to take a careful approach to development. First we wrote a complete software prototype to emulate CHIP8, as suggested by a TA. This allowed us to understand a general strategy for getting the hardware design working, as well as the format of CHIP8 files we found online. We additionally wrote an instruction-by-instruction test suite using Google Test to test both the software prototype and the hardware design, as verification after-the-fact is very difficult for a module as complex as the CPU. All tests are in _test.cpp files in their respective directories. The software prototype is completely tested, and so are some of the hardware modules (particularly the rams).

The software prototype as well as the hardware tests are built using CMake. The software driver is built using a Makefile.

What doesn't work:
Due to time constraints, we were not able to finish the CPU module or integrate the sound module. Additionally, though we believe our driver works as-is, there are likely hidden errors since we weren't able to actually, you know, test it.

Code Listings

/CMakeLists.txt

```
 cmake_minimum_required (VERSION 3.16)
project (yachip8 C CXX)
SET(CMAKE_EXPORT_COMPILE_COMMANDS ON)
set(CMAKE_CXX_STANDARD 20)

cmake_host_system_information(RESULT HNAME QUERY HOSTNAME)

if ("${HNAME}" STREQUAL "de1-soc")
  set(ISDE1SOC ON CACHE BOOL "Are we running on FPGA or desktop?")
  message(STATUS "Detected running on FPGA; not compiling tests.")
else()
  set(ISDE1SOC OFF CACHE BOOL "Are we running on FPGA or desktop?")
  message(STATUS "Detected running on desktop.")
endif()

set(COMPILE_SWPROTO OFF CACHE BOOL "Compile software prototype and tests.")

# GoogleTest snippet stolen from
# https://cmake.org/cmake/help/latest/module/FetchContent.html
include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        "release-1.11.0"
)
# For Windows: Prevent overriding the parent project's compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
# FetchContent_MakeAvailable is to be set in subdirs as needed.

set(WARNING_LIST
  -Wall
  -Wextra
  -pedantic
  -Werror
  -Wno-unused
  -Wno-missing-field-initializers
)
string(REPLACE ";" " " WARNING_FLAGS "${WARNING_LIST}")

if (COMPILE_SWPROTO)
  add_subdirectory (swproto)
endif()

# add_subdirectory (sw)
add_subdirectory (hw)
```

Hardware Code:

/hw/CMakeLists.txt

```
if (ISDE1SOC)

else()
  FetchContent_MakeAvailable(googletest)

  find_package(verilator HINTS $ENV{VERILATOR_ROOT} ${VERILATOR_ROOT})
  if (NOT verilator_FOUND)
    message(FATAL_ERROR
      "Verilator not found. Install it or set VERILATOR_ROOT environment variable")
  endif()

  #file(GLOB_RECURSE ${VERILATOR_ROOT} "*.c" "*.h")

  set(hw_test_sources hw_test.hpp main.cpp
    chip_ram_test.cpp display_ram_test.cpp cpu_test.cpp)
  list(LENGTH hw_test_sources hw_test_sources_nr)

  set(hw_test_include_dirs "")
  list(LENGTH hw_test_include_dirs hw_test_include_dirs_nr)

  add_executable(hw_test ${hw_test_sources})
  target_link_libraries(hw_test PRIVATE gtest)
  set_target_properties(hw_test PROPERTIES
    CXX_STANDARD 20
    CXX_STANDARD_REQUIRED ON)
  # Each verilate() call may have only one top level module which gets turned
  # into a .h file.
  verilate(hw_test TRACE SOURCES chip_ram.sv)
  verilate(hw_test TRACE SOURCES display_ram.sv)
  verilate(hw_test TRACE SOURCES display.sv)
  verilate(hw_test TRACE SOURCES cpu.sv)

  # We have to enable warnings piecemeal so as not to interfere
  # with Verilator.

  # Move verilator includes to system includes.
  get_target_property(
    verilator_hw_test_include_dirs
    hw_test
    INCLUDE_DIRECTORIES
  )
  set_target_properties(hw_test PROPERTIES INCLUDE_DIRECTORIES "")
  target_include_directories(hw_test SYSTEM
    PRIVATE ${verilator_hw_test_include_dirs})

  set(TRACE_DIR "${CMAKE_CURRENT_BINARY_DIR}/traces/"
    CACHE PATH
    "Where to put Verilator test traces.")

  file(MAKE_DIRECTORY "${TRACE_DIR}")
  target_compile_definitions(hw_test PRIVATE TRACE_DIR="${TRACE_DIR}")

  # Enable warnings for our code, only.
  set_property(
    SOURCE ${hw_test_sources}
    APPEND PROPERTY COMPILE_FLAGS " ${WARNING_FLAGS} ")


endif()
```

/hw/sound.sv

```
// Due to time constraints, we were unable to integrate the sound module.
// This file is mixed with some old testing code but does contain the guts
// to make the sound work.
module vga_ball(input logic        clk,
               input logic           reset,
             input logic [15:0]  writedata,
             input logic     write,
             input           chipselect,
//           input logic [4:0]  address,
             output logic [7:0] addr,

             input left_chan_ready,
             input right_chan_ready,

             output logic [7:0] VGA_R, VGA_G, VGA_B,
             output logic    VGA_CLK, VGA_HS, VGA_VS,
                               VGA_BLANK_n,
             output logic    VGA_SYNC_n,
             output logic [15:0] sample_data_l,
             output logic sample_valid_l,
             output logic [15:0] sample_data_r,
             output logic sample_valid_r);


   logic [10:0]        hcount;
   logic [9:0]     vcount;
   logic [7:0] data;


   logic [7:0]         background_r, background_g, background_b;
//   logic [15:0]    game_data;

       reg [11:0] counter;
       //logic flag1;
       //logic flag2;
       logic flag3;

   fb_counters counters(.clk50(clk), .*);
   character_sprites boss(.addr(addr), .data(data));
   parameter DISP_COLS = 10'd640;
   parameter DISP_ROWS = 10'd480;
   parameter DISP_COL_OFFSET = 10'd64;
   parameter DISP_ROW_OFFSET = 10'd112;
   parameter CHIP8_SCALING_FACTOR = 8;

   logic [9:0] disp_col;  // hcount[10:1] is pixel column
   logic [9:0] disp_row;  // vcount[9:0] is pixel row
   logic chip8_area;
   logic is_foreground;
   logic [5:0] chip8_col;
   logic [4:0] chip8_row;
   logic [7:0] d_is_foreground;
   logic [7:0] misc;
   assign misc = 8'h80 >> (chip8_col % 8);

   assign disp_col = hcount[10:1];
   assign disp_row = vcount;

   assign chip8_col = (disp_col - DISP_COL_OFFSET) / CHIP8_SCALING_FACTOR;
   assign chip8_row = (disp_row - DISP_ROW_OFFSET) / CHIP8_SCALING_FACTOR;
   assign chip8_area = (disp_col >= DISP_COL_OFFSET) && (disp_col < (DISP_COLS -
DISP_COL_OFFSET))
                          && (disp_row >= DISP_ROW_OFFSET) && (disp_row < (DISP_ROWS -
DISP_ROW_OFFSET));
   assign addr = (chip8_row * 8'h08) + ( chip8_row / 8);
   assign d_is_foreground = data & misc;
   assign is_foreground = d_is_foreground > 0 ? 1 : 0;

       always_comb begin
              {VGA_R,VGA_G,VGA_B}={8'h00,8'h00,8'h00};
              if ((VGA_BLANK_n) && (chip8_area) && (is_foreground))
              begin
                     {VGA_R,VGA_G,VGA_B} = {8'hff, 8'hff, 8'hff};
              end
       else begin
              {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};
       end
end

       reg    [13:0] address1;
       wire   [15:0] q1;

       shoot audio1(.address(address1), .clock(clk), .q(q1)); //1653

logic left_chan_ready1;
logic right_chan_ready1;
initial begin
       left_chan_ready1 = 1'b1;
       right_chan_ready1 = 1'b1;
end
       always_ff @(posedge clk) begin
              if(reset) begin
                     counter <= 0;
                     sample_valid_l <= 0; sample_valid_r <= 0;
              end

              else if(left_chan_ready1 == 1 && right_chan_ready1 == 1 && counter <
3125) begin
                     counter <= counter + 1;
                     sample_valid_l <= 0; sample_valid_r <= 0;
              end
              else if(left_chan_ready1 == 1 && right_chan_ready1 == 1 && counter ==
3125) begin
                     counter <= 0;
                     sample_valid_l <= 1; sample_valid_r <= 1;

                     if (is_foreground==1'b1 || flag3 ==1'b0) begin
                            if (address1 < 13'd5832) begin
```

```
                                address1 <= address1+1;
                                flag3 <= 1'b0;
                        end
                        else begin
                                address1 <=0;
                                flag3 <= 1'b1;
                        end
                        sample_data_l <= q1;
                        sample_data_r <= q1;
                end

                else begin
                        sample_data_l <= 0;
                        sample_data_r <= 0;
                end
        end

        else begin
        sample_valid_l <= 0; sample_valid_r <= 0;
        end
    end



/*   always_comb begin
       case (address)
         3'h3 : begin
                hposnext    = {hpos[9:8], writedata};
                vposnext    = vpos;
               end
         3'h4 : begin
                hposnext    = {writedata[1:0], hpos[7:0]};
                vposnext    = {vpos[8:6], writedata[7:2]};
               end
         3'h5 : begin
                hposnext    = hpos;
                vposnext    = {writedata[2:0], vpos[5:0]};
               end
         default: begin
                hposnext    = hpos;
                vposnext    = vpos;
               end
        endcase

   end*/

/*   always_ff @(posedge clk)
   begin
       hpos <= hpos;
       vpos <= vpos;
     if (reset) begin
       hpos <= 10'd56;
       vpos <=  9'd56;
       background_r <= 8'h0;
       background_g <= 8'h0;
       background_b <= 8'h80;
     end else if (chipselect && write)
       case (address)
        3'h0 : background_r <= writedata;
        3'h1 : background_g <= writedata;
        3'h2 : background_b <= writedata;
        default: begin
             hpos <= hposnext;
             vpos <= vposnext;
          end
       endcase
   end*/

/*   always_comb begin
      {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};
      if (VGA_BLANK_n )
       if ( ((hcount[10:1] - hpos)*(hcount[10:1] - hpos) + (vcount[8:0] -
vpos)*(vcount[8:0] - vpos)) <= 256 )
         {VGA_R, VGA_G, VGA_B} = {8'h10, 8'hff, 8'h00};
       else
         {VGA_R, VGA_G, VGA_B} =
             {background_r, background_g, background_b};
   end*/

endmodule

module fb_counters(
 input logic      clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic            VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *             _______________              ________
 * ___________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _______________________      _____________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                            HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                            VBACK_PORCH; // 525

   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else             hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          vcount <= 0;
     else if (endOfLine)
       if (endOfField)   vcount <= 0;
       else              vcount <= vcount + 10'd 1;

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                   !(hcount[7:5] == 3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

   assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   // 101 0000 0000  1280         01 1110 0000  480
   // 110 0011 1111  1599         10 0000 1100  524
   assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                    !( vcount[9] | (vcount[8:5] == 4'b1111) );

   /* VGA_CLK is 25 MHz
    *             __    __    __
    * clk50    __|  |__|  |__|
    *
    *             _____       __
    * hcount[0]__|     |_____|
    */
   assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule
```

/hw/chip_ram.sv

```
module chip_ram(
  input logic clk,
  input logic [11:0] aa, ab,   // Addresses
  input logic [7:0] da, db,  // Data in
  input logic wa, wb,         // Write Enables
  output logic [7:0] qa, qb  // Data out
);

logic [7:0] mem [4095:0];
always_ff @(posedge clk) begin
  if (wa) begin
    mem[aa] <= da;
    qa <= da;
  end else qa <= mem[aa];
end
always_ff @(posedge clk) begin
  if (wb) begin
    mem[ab] <= db;
    qb <= db;
  end else qb <= mem[ab];
end

endmodule
```

/hw/cpu.sv

```
module cpu(input logic clk,
  input logic [15:0] keystate,
  input logic reset,

  output logic disp_ram_req,  // If we're asking for rw op on display ram.

  output logic [7:0] disp_aa, disp_ab, // Addresses
  output logic [7:0] disp_da, disp_db, // Data in
  output logic disp_wa, disp_wb,       // Write Enables
  input logic [7:0] disp_qa, disp_qb,  // Data out

  output logic [11:0] ch_aa, ch_ab,     // Addresses
  output logic [7:0] ch_da, ch_db,     // Data in
  output logic ch_wa, ch_wb,           // Write Enables
  input logic [7:0] ch_qa, ch_qb       // Data out

);

`define PROG_START 16'h200;

enum {FETCHING, EXECUTING} state;

// Every variable with a _next counterpart stores the value in the
// register. the _next counterpart combinationally determines
// the value the register will be assigned on the next posedge clk.

logic [15:0] pc /* register */, pc_next /* wire */;
// Stack pointer points to location where next address will be written.
logic [3:0] sp /* register */, sp_next /* wire */;
logic [15:0] stack[16];
logic [7:0] regs[8];
logic [15:0] opcode;  // Currently executing opcode.
logic do_opcode_capture_next_cycle;

initial begin
  state = FETCHING;
  sp = 0;
  pc = `PROG_START;
end

always_comb begin
  /* Fill me in! */
  pc_next = pc;
  sp = sp_next;

end


always_ff @(posedge clk) begin
  if (reset) begin
    sp <= 0;
  end
end
```

```
endmodule
```

/hw/display.sv

```
module display(input logic clk,
  // We ask for the byte at some address on the screen.
  output logic [7:0] addr,
  input logic [7:0] data,

  output logic [7:0] VGA_R, VGA_G, VGA_B,
  output logic         VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
  output logic         VGA_SYNC_n);

   logic [10:0]        hcount;
   logic [9:0]     vcount;

   fb_counters counters(.clk50(clk), .*);
   parameter DISP_COLS = 10'd640;
   parameter DISP_ROWS = 10'd480;
   parameter DISP_COL_OFFSET = 10'd64;
   parameter DISP_ROW_OFFSET = 10'd112;
   parameter CHIP8_SCALING_FACTOR = 8;

   logic [9:0] disp_col;  // hcount[10:1] is pixel column
   logic [9:0] disp_row;  // vcount[9:0] is pixel row
   logic chip8_area;
   logic is_foreground;
   logic [5:0] chip8_col;
   logic [4:0] chip8_row;
   logic [7:0] d_is_foreground;
       logic [7:0] misc;
   assign misc = 8'h80 >> (chip8_col % 8);

   assign disp_col = hcount[10:1];
   assign disp_row = vcount;

   assign chip8_col = (disp_col - DISP_COL_OFFSET) / CHIP8_SCALING_FACTOR;
   assign chip8_row = (disp_row - DISP_ROW_OFFSET) / CHIP8_SCALING_FACTOR;
   assign chip8_area = (disp_col >= DISP_COL_OFFSET) && (disp_col < (DISP_COLS -
DISP_COL_OFFSET))
                          && (disp_row >= DISP_ROW_OFFSET) && (disp_row < (DISP_ROWS -
DISP_ROW_OFFSET));
   assign addr = (chip8_row * 8'h08) + ( chip8_row / 8);
   assign d_is_foreground = data & misc;
   assign is_foreground = d_is_foreground > 0 ? 1 : 0;

       always_comb begin
             {VGA_R,VGA_G,VGA_B}={8'h00,8'h00,8'h00};
             if ((VGA_BLANK_n) && (chip8_area) && (is_foreground))
             begin
                   {VGA_R,VGA_G,VGA_B} = {8'hff, 8'hff, 8'hff};
             end
       else begin
             {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};
       end
end

endmodule

module fb_counters(
 input logic      clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic            VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *            _______________            ________
 * ___________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *      _______________________      _____________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                            HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                            VBACK_PORCH; // 525

   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else             hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          vcount <= 0;
     else if (endOfLine)
       if (endOfField)   vcount <= 0;
       else              vcount <= vcount + 10'd 1;
   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                   !(hcount[7:5] == 3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

   assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   // 101 0000 0000  1280         01 1110 0000  480
   // 110 0011 1111  1599         10 0000 1100  524
   assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                    !( vcount[9] | (vcount[8:5] == 4'b1111) );

   /* VGA_CLK is 25 MHz
    *             __    __    __
    * clk50    __|  |__|  |__|
    *
    *             _____       __
    * hcount[0]__|     |_____|
    */
   assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule
```

/hw/display_ram.sv

```
module display_ram(
  input logic clk,
  input logic [7:0] aa, ab,   // Addresses
  input logic [7:0] da, db,  // Data in
  input logic wa, wb,         // Write Enables
  output logic [7:0] qa, qb  // Data out
);

logic [7:0] mem [255:0];
always_ff @(posedge clk) begin
  if (wa) begin
    mem[aa] <= da;
    qa <= da;
  end else qa <= mem[aa];
end
always_ff @(posedge clk) begin
  if (wb) begin
    mem[ab] <= db;
    qb <= db;
  end else qb <= mem[ab];
end

endmodule
```

/hw/timer.sv

```
module timer_reg_60hz (
  input logic clk50,  // 50 MHz
  input logic we, // Write enable.
  input logic [7:0] newval,
  output logic [7:0] val
);

// Count down from COUNTER_MAX. Counter will hit 0 in 1/60 seconds.
`define COUNTER_MAX = 833333;

logic [19:0] counter, counter_next;
assign counter_next = |counter_next ? (counter_next - 1) : counter_next;

logic val_next;
always_comb begin
  if (we) begin
    val_next = newval;
  end if (we) begin
    val_next = newval;
  end
end

always_ff @(posedge clk) begin
  counter <= counter_next;
end

endmodule
```

/hw/yachip8.sv

```
module yachip8(input logic clk,
  input logic reset,
  input logic [15:0] writedata,
  //output logic [15:0] readdata,
  input logic          write,
  input                chipselect,
  input logic [11:0]  address,

  output logic [7:0] VGA_R, VGA_G, VGA_B,
  output logic         VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
  output logic         VGA_SYNC_n);

// Miscellaneous variables.
logic [15:0] keystate;  // updated in "control logics" section.

// Note that the word "req" here is shorthand for "request".

//
// Instantiate rams.
//

logic [7:0] disp_aa, disp_ab;
logic [7:0] disp_da, disp_db;
logic       disp_wa, disp_wb;
logic [7:0] disp_qa, disp_qb;

display_ram disp_ram(.clk,
  .aa(disp_aa), .ab(disp_ab),
  .da(disp_da), .db(disp_db),
  .wa(disp_wa), .wb(disp_wb),
  .qa(disp_qa), .qb(disp_qb)
);

logic [11:0] ch_aa, ch_ab;
logic [7:0] ch_da, ch_db;
logic ch_wa, ch_wb;
logic [7:0] ch_qa, ch_qb;
chip_ram ch_ram(.clk,
  .aa(ch_aa), .ab(ch_ab),
  .da(ch_da), .db(ch_db),
  .wa(ch_wa), .wb(ch_wb),
  .qa(ch_qa), .qb(ch_qb)
);

//
// Instantiate cpu
//

logic cpu_may_run, cpu_disp_ram_req;
logic  [7:0] cpu_disp_aa_req, cpu_disp_ab_req;
logic  [7:0] cpu_disp_da_req, cpu_disp_db_req;
logic        cpu_disp_wa_req, cpu_disp_wb_req;
logic [11:0] cpu_ch_aa_req, cpu_ch_ab_req;
logic  [7:0] cpu_ch_da_req, cpu_ch_db_req;
logic        cpu_ch_wa_req, cpu_ch_wb_req;

cpu ch(.clk, .keystate,
  .may_run(cpu_may_run),

  .disp_ram_req(cpu_disp_ram_req),
  .disp_wa(cpu_disp_wa_req), .disp_wb(cpu_disp_wb_req),
  .disp_da(cpu_disp_da_req), .disp_db(cpu_disp_db_req),
  .disp_aa(cpu_disp_aa_req), .disp_ab(cpu_disp_ab_req),
  .disp_qa, .disp_qb,
  .ch_wa(cpu_ch_wa_req), .ch_wb(cpu_ch_wb_req),
  .ch_da(cpu_ch_da_req), .ch_db(cpu_ch_db_req),
  .ch_aa(cpu_ch_aa_req), .ch_ab(cpu_ch_ab_req),
  .ch_qa, .ch_qb
);

//
// Instantiate display
//

logic [7:0] display_addr_req;

display disp(.clk,
  .addr(display_addr_req), .data(disp_qa),
  .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS, .VGA_BLANK_n, .VGA_SYNC_n
);

//
// Control logics
//

// Logics to control who is touching what.
// The hierarchy is as follows:
// 1. the controller may write to any address.
// 2. Else, we let the cpu ask.
// 3. Third, we let the display ask.

// Characterize read/writes.
// (whether the requested address is the 4k ram, a keystate, etc).
logic req_addr_is_ch_ram, req_addr_is_disp_ram,
  req_addr_is_keystate, req_addr_is_control;
assign req_addr_is_ch_ram   = (address < 12'h800);
assign req_addr_is_disp_ram = (12'h800 <= address) && (address < 12'h880);
assign req_addr_is_keystate = (12'hFFE == address);
assign req_addr_is_control  = (12'hFFF == address);

logic [11:0] address_to_ch_ram_lower_addr, address_to_ch_ram_upper_addr;
assign address_to_ch_ram_lower_addr = address << 1;
assign address_to_ch_ram_upper_addr = address << 1 | 12'b1;
logic [7:0] address_to_disp_ram_lower_addr, address_to_disp_ram_upper_addr;
assign address_to_disp_ram_lower_addr = {1'b0, address[6:0]};
assign address_to_disp_ram_upper_addr = {1'b0, address[6:0]} << 1 | 8'b1;

// Delegate ram access to display, ram, or user of this module.
always_comb begin
```

```
  cpu_may_run = 0;
  {disp_aa, disp_ab, disp_da, disp_db, disp_wa, disp_wb} = 0;
  {ch_aa, ch_ab, ch_da, ch_db, ch_wa, ch_wb} = 0;

  if (write && req_addr_is_ch_ram) begin
    ch_aa = address_to_ch_ram_lower_addr;
    ch_ab = address_to_ch_ram_upper_addr;
    ch_da = writedata[15:8];
    ch_da = writedata[7:0];
    {ch_wa, ch_wb} = 2'b11;
  end else if (write && req_addr_is_disp_ram) begin
    disp_aa = address_to_disp_ram_lower_addr;
    disp_ab = address_to_disp_ram_upper_addr;
    disp_da = writedata[15:8];
    disp_da = writedata[7:0];
    {disp_wa, disp_wb} = 2'b11;
  end else begin
    // If the controller isn't writing, cpu can do what it wants.
    cpu_may_run = 1;
    // If the controller isn't writing, cpu controls ch_ram.
    {ch_aa, ch_ab} = {cpu_ch_aa_req, cpu_ch_ab_req};
    {ch_da, ch_db} = {cpu_ch_da_req, cpu_ch_db_req};
    {ch_wa, ch_wb} = {cpu_ch_wa_req, cpu_ch_wb_req};

    // display only gets disp_ram if cpu isn't asking for it.
    if (cpu_disp_ram_req) begin
      {disp_aa, disp_ab} = {cpu_disp_aa_req, cpu_disp_ab_req};
      {disp_da, disp_db} = {cpu_disp_da_req, cpu_disp_db_req};
      {disp_wa, disp_wb} = {cpu_disp_wa_req, cpu_disp_wb_req};
    end else begin
      disp_aa = display_addr_req;
    end
  end
end


initial keystate = 0;
always_ff @(posedge clk) begin
  if (write && req_addr_is_keystate) begin
    keystate <= writedata;
  end
end

endmodule
```

/hw/chip_ram_test.cpp

```
#include "hw_test.hpp"

#include "Vchip_ram.h"

using ChipRamTest = VTest<Vchip_ram>;

TEST_F(ChipRamTest, TestSimpleWriteRead) {
  eval();
  dut.wa = 1;
  dut.wb = 1;
  // Write mem[0x01] = 0x23; mem[0x45] = 0x67;
  dut.aa = 0x01;
  dut.da = 0x23;
  dut.ab = 0x45;
  dut.db = 0x67;
  dut.wa = 1;
  dut.wb = 1;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0x67);

  // Make sure it does nothing with write enable off on port a.
  dut.wa = 0;
  dut.da = 0xff;
  // Write mem[0x89] = 0xAB;
  dut.ab = 0x89;
  dut.db = 0xAB;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0xAB);

  // Read address 0x01 with port b.
  dut.wb = 0;
  dut.ab = 0x01;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0x23);
}
```

/hw/cpu_test.cpp

```
#include "hw_test.hpp"

#include "Vdisplay_ram.h"

using DisplayRamTest = VTest<Vdisplay_ram>;

TEST_F(DisplayRamTest, TestSimpleWriteRead) {
  eval();
  dut.wa = 1;
  dut.wb = 1;
  // Write mem[0x01] = 0x23; mem[0x45] = 0x67;
  dut.aa = 0x01;
  dut.da = 0x23;
  dut.ab = 0x45;
  dut.db = 0x67;
  dut.wa = 1;
  dut.wb = 1;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0x67);

  // Make sure it does nothing with write enable off on port a.
  dut.wa = 0;
  dut.da = 0xff;
  // Write mem[0x89] = 0xAB;
  dut.ab = 0x89;
  dut.db = 0xAB;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0xAB);

  // Read address 0x01 with port b.
  dut.wb = 0;
  dut.ab = 0x01;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0x23);
}
```

/hw/display_ram_test.cpp

```
#include "hw_test.hpp"

#include "Vdisplay_ram.h"

using DisplayRamTest = VTest<Vdisplay_ram>;

TEST_F(DisplayRamTest, TestSimpleWriteRead) {
  eval();
  dut.wa = 1;
  dut.wb = 1;
  // Write mem[0x01] = 0x23; mem[0x45] = 0x67;
  dut.aa = 0x01;
  dut.da = 0x23;
  dut.ab = 0x45;
  dut.db = 0x67;
  dut.wa = 1;
  dut.wb = 1;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0x67);

  // Make sure it does nothing with write enable off on port a.
  dut.wa = 0;
  dut.da = 0xff;
  // Write mem[0x89] = 0xAB;
  dut.ab = 0x89;
  dut.db = 0xAB;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0xAB);

  // Read address 0x01 with port b.
  dut.wb = 0;
  dut.ab = 0x01;
  ticktock();
  ASSERT_EQ(dut.qa, 0x23);
  ASSERT_EQ(dut.qb, 0x23);
}
```

/hw/hw_test.hpp

```
#ifndef YACHIP8_HW_HW_TEST_HPp
#define YACHIP8_HW_HW_TEST_HPp
#include <algorithm>
#include <any>
#include <bitset>
#include <functional>
#include <gtest/gtest.h>
#include <initializer_list>
#include <memory>
#include <sstream>
#include <verilated_vcd_c.h>

static const uint64_t PICOS = 1000000000000ul;
static const uint64_t HZ = 50000000ul;
static const uint64_t HALF_CYCLE_TIME = PICOS / HZ / 2;

extern const std::unique_ptr<VerilatedContext> contextp;
// dut is short for Device Under Test
template <typename VModel> class VTest : public ::testing::Test {
protected:
  // Device under test.
  VModel dut;
  // Trace file
  VerilatedVcdC tf;
  uint64_t time;
  const testing::TestInfo *const test_info =
      testing::UnitTest::GetInstance()->current_test_info();

  std::string get_tf_name() {
    std::stringstream name;
    name << TRACE_DIR << test_info->test_suite_name() << '_'
         << test_info->name() << ".vcd";
    return name.str();
  }

  VTest() : dut{}, tf{}, time{0} {
    dut.final();
    dut.trace(&tf, 99);
    auto tf_name = get_tf_name();
    tf.open(tf_name.c_str());
    if (!tf.isOpen()) {
      std::cerr << "Failed to open " + tf_name + "; aborting.\n";
      abort();
      return;
    }

    dut_clk_member() = 0;
  }

  auto &&dut_clk_member();

  inline void eval() { dut.eval(); }
  inline void tick() {
    dut_clk_member() = 1;
    dut.eval();
    tf.dump(time);
    time += HALF_CYCLE_TIME;
  }
  inline void tock() {
    dut_clk_member() = 0;
    dut.eval();
    tf.dump(time);
    time += HALF_CYCLE_TIME;
  }

  inline void ticktock() {
    tick();
    tock();
  }

  ~VTest() {
    dut.final();
    if (tf.isOpen()) {
      tf.close();
    }
  }
};

template <typename VModel> auto &&VTest<VModel>::dut_clk_member() {
  return dut.clk;
}

#endif // YACHIP8_HW_HW_TEST_HPp
```

/hw/main.cpp

```
#include "hw_test.hpp"

const std::unique_ptr<VerilatedContext> contextp =
    std::make_unique<VerilatedContext>();

int main(int argc, char **argv) {
  contextp->commandArgs(argc, argv);
  contextp->traceEverOn(true);
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

/swproto/font.hpp

```
#ifndef SWIMP_FONT_H
#define SWIMP_FONT_H

#include <array>
#include <cstdint>

/*
 * Font specified by http://devernay.free.fr/hacks/chip8/C8TECH10.HTM
 */
const static std::size_t FONT_SPRITE_SIZE = 5;
extern const std::array<uint8_t, FONT_SPRITE_SIZE * 16> font;

#endif // SWIMP_FONT_H
```

/swproto/font.cpp

```
#include "font.hpp"

const std::array<uint8_t, FONT_SPRITE_SIZE * 16> font = {
    0xF0, 0x90, 0x90, 0x90, 0xF0, // 0
    0x20, 0x60, 0x20, 0x20, 0x70, // 1
    0xF0, 0x10, 0xF0, 0x80, 0xF0, // 2
    0xF0, 0x10, 0xF0, 0x10, 0xF0, // 3
    0x90, 0x90, 0xF0, 0x10, 0x10, // 4
    0xF0, 0x80, 0xF0, 0x10, 0xF0, // 5
    0xF0, 0x80, 0xF0, 0x90, 0xF0, // 6
    0xF0, 0x10, 0x20, 0x40, 0x40, // 7
    0xF0, 0x90, 0xF0, 0x90, 0xF0, // 8
    0xF0, 0x90, 0xF0, 0x10, 0xF0, // 9
    0xF0, 0x90, 0xF0, 0x90, 0x90, // A
    0xE0, 0x90, 0xE0, 0x90, 0xE0, // B
    0xF0, 0x80, 0x80, 0x80, 0xF0, // C
    0xE0, 0x90, 0x90, 0x90, 0xE0, // D
    0xF0, 0x80, 0xF0, 0x80, 0xF0, // E
    0xF0, 0x80, 0xF0, 0x80, 0x80  // F
};
```

/swproto/statemachine.hpp

```
#ifndef SWIMP_STATEMACHINE_H
#define SWIMP_STATEMACHINE_H

#include <array>
#include <cstdint>
#include <initializer_list>
#include <span>
#include <stack>
#include <vector>

class statemachine {
public:
  const static unsigned MEMORY_SIZE = 4096;
  const static unsigned STACK_SIZE = 16;
  const static unsigned DISPLAY_WIDTH = 64;
  const static unsigned ROW_SIZE = DISPLAY_WIDTH / 8;
  const static unsigned DISPLAY_HEIGHT = 32;
  const static unsigned ROW_MASK = DISPLAY_HEIGHT - 1;
  const static unsigned ROW_OFFSET_MASK = ROW_SIZE - 1;
  const static unsigned DISPLAY_SIZE = ROW_SIZE * DISPLAY_HEIGHT;
  const static unsigned PROG_BEGIN = 0x200;

  // Non-negative statuses are those from which the state machine may recover.
  // Negative statuses are those with overflows.
  enum status {
    NO_ERROR = 0,
    NOT_IMPLEMENTED = 1,
    WAITING_FOR_KEYPRESS = 2,
    POPPED_EMPTY_STACK = -2,
    PUSHED_FULL_STACK = -3,
    MEMORY_OVERFLOW = -5,
    IMPOSSIBLE_KEYPRESS_REQUEST = -6,
    PC_UNALIGNED = -7,
    PC_OUT_OF_RANGE = -8,
    DEBUG_ERROR = -100, // Thrown for testing purposes.
  };

  struct init_conf {
    uint16_t pc;
    uint16_t font_begin;
    bool quirk_shift : 1;
    bool quirk_load_store : 1;
  };

  statemachine(std::array<uint8_t, MEMORY_SIZE> mem, init_conf conf = {});

  statemachine(std::initializer_list<uint16_t> instructions,
               init_conf conf = {});

  /**
   * Executes one instruction
   * @param ticks Number of 60Hz whole ticks that have passed since the last
   * invocation.
   */
  status step(uint16_t keystate, bool tick);

  /// Get current value of special register I.
  inline uint16_t reg_I() const { return m_reg_I; };

  /// Get current value of delay timer register.
  inline uint8_t reg_DT() const { return m_reg_DT; };

  /// Get current value of sound timer register.
  inline uint8_t reg_ST() const { return m_reg_ST; };

  /// Get current display
  inline const std::array<uint8_t, DISPLAY_SIZE> display() const {
    return m_display;
  };

  /// Get current memory
  inline std::span<const uint8_t, MEMORY_SIZE> memory() const { return m_mem; };

  /// Get current registers
  inline std::span<const uint8_t, 16> regs() const { return m_regs; };

  /// Get current program counter.
  inline uint16_t pc() const { return m_pc; };

  /// Get current stack
  inline std::span<const uint16_t> stack() const {
    return m_stack.const_view();
  };

  inline uint16_t curr_instruction() const {
    return (static_cast<uint16_t>(m_mem[m_pc]) << 8) |
           static_cast<uint16_t>(m_mem[m_pc + 1]);
  }

private:
  class instruction_stack : public std::stack<uint16_t, std::vector<uint16_t>> {
  public:
    std::span<const uint16_t> const_view() const;
  };

  std::array<uint8_t, MEMORY_SIZE> m_mem;
  std::array<uint8_t, DISPLAY_SIZE> m_display;
  std::array<uint8_t, 16> m_regs;
  instruction_stack m_stack;
  uint16_t m_pc;
  uint16_t m_font_begin;
  uint16_t m_reg_I;
  // Timer registers.
  uint8_t m_reg_DT; // delay timer.
  uint8_t m_reg_ST; // sound timer
  /* Quirk behavior detailed in
   * http://mir3z.github.io/chip8-emu/doc/chip8-cpu.js.html#sunlight-1-line-119
   */
  bool m_quirk_shift : 1;
  bool m_quirk_load_store : 1;
};

#endif // SWIMP_STATEMACHINE_H
```

/swproto/statemachine.cpp

```
#include <algorithm>
#include <bitset>
#include <cassert>
#include <clocale>
#include <cmath>
#include <cstddef>
#include <iomanip>
#include <ios>
#include <iostream>
#include <iterator>
#include <mutex>
#include <random>
#include <vector>

#include "font.hpp"
#include "statemachine.hpp"

static std::mt19937 random_generator;
static std::mutex random_mutex;

inline static std::array<uint8_t, statemachine::MEMORY_SIZE>
instructions_decode(const std::initializer_list<uint16_t> instructions) {

  assert(instructions.size() <= (statemachine::MEMORY_SIZE / 2));

  std::array<uint8_t, statemachine::MEMORY_SIZE> mem{0};
  unsigned i = 0;
  for (uint16_t instruction : instructions) {
    // Necessary to do it this way b/c of endianness correctness.
    mem[i++] = instruction >> 8;
    mem[i++] = instruction & 0xFF;
  }

  return mem;
}

statemachine::statemachine(std::array<uint8_t, MEMORY_SIZE> mem,
                           statemachine::init_conf conf)
    : m_mem(mem), m_display{0}, m_regs{0}, m_stack{}, m_pc(conf.pc),
      m_font_begin(conf.font_begin), m_reg_I(0), m_reg_DT(0), m_reg_ST(0),
      m_quirk_shift(conf.quirk_shift),
      m_quirk_load_store(conf.quirk_load_store) {}

statemachine::statemachine(std::initializer_list<uint16_t> instructions,
                           statemachine::init_conf conf)
    : m_mem(instructions_decode(instructions)), m_display{0}, m_regs{0},
      m_stack{}, m_pc(conf.pc), m_font_begin(conf.font_begin), m_reg_I(0),
      m_reg_DT(0), m_reg_ST(0), m_quirk_shift(conf.quirk_shift),
      m_quirk_load_store(conf.quirk_load_store) {}

statemachine::status statemachine::step(uint16_t keystate, bool tick) {

  if (m_pc & 1) [[unlikely]] {
    return PC_UNALIGNED;
```

```
}
if (m_pc >= statemachine::MEMORY_SIZE) [[unlikely]] {
  return PC_UNALIGNED;
}

// Opcodes are stored in most-significant-byte-first.
uint16_t opcode =
    m_mem.at(m_pc | 1) | (m_mem.at(static_cast<uint16_t>(m_pc)) << 8);

static uint16_t last_pc = 0xFFFF;

if (m_pc != last_pc) {

  std::cout << "pc: " << m_pc << " opcode: " << std::hex << std::setw(4)
            << std::setfill('0') << opcode << '\n';

  last_pc = m_pc;
}

uint16_t nnn = opcode & 0xFFF;
uint16_t n = opcode & 0xF;
uint8_t x = (opcode >> 8) & 0xF;
uint8_t y = (opcode >> 4) & 0xF;
uint8_t kk = opcode & 0xFF;

// Tick tick tick
if (tick) {
  if (m_reg_DT > 0) {
    --m_reg_DT;
  }
  if (m_reg_ST > 0) {
    --m_reg_ST;
  }
}

// Grab first hexadigit.
switch (opcode >> 12) {
case 0x0: {
  if (opcode == 0x00E0) {
    std::fill(m_display.begin(), m_display.end(), 0);
  } else if (opcode == 0x00EE) {
    if (m_stack.empty()) [[unlikely]] {
      return POPPED_EMPTY_STACK;
    }
    m_pc = m_stack.top();
    m_stack.pop();
    return NO_ERROR;
  } else {
    // Ignore SYS
  }
} break;

case 0x1: {
  m_pc = nnn;
  return NO_ERROR;
```

```
} break;

case 0x2: {
  if (m_stack.size() == STACK_SIZE) [[unlikely]] {
    return PUSHED_FULL_STACK;
  } else {
    m_stack.push(m_pc + 2);
    m_pc = nnn;
    return NO_ERROR;
  }
}

case 0x3: {
  if (m_regs.at(x) == kk) {
    m_pc += 4;
    return NO_ERROR;
  }
} break;

case 0x4: {
  if (m_regs.at(x) != kk) {
    m_pc += 4;
    return NO_ERROR;
  }
} break;

case 0x5: {
  if (m_regs.at(x) == m_regs.at(y)) {
    m_pc += 4;
    return NO_ERROR;
  }
} break;

case 0x6: {
  m_regs[x] = kk;
} break;

case 0x7: {
  m_regs[x] += kk;
} break;

case 0x8: {
  switch (opcode & 0xF) {
  case 0x0: {
    m_regs[x] = m_regs.at(y);
  } break;

  case 0x1: {
    m_regs[x] |= m_regs.at(y);
  } break;

  case 0x2: {
    m_regs[x] &= m_regs.at(y);
  } break;
```

```
    case 0x3: {
      m_regs[x] ^= m_regs.at(y);
    } break;

    case 0x4: {
      // Octo spec (see octo/examples/test/testquirks)
      // expects carry flag to be written last.
      uint16_t sum = static_cast<uint16_t>(m_regs.at(x)) +
                     static_cast<uint16_t>(m_regs.at(y));
      m_regs[x] = sum;
      m_regs[0xF] = (sum > 0xFF) ? 1 : 0;
    } break;

    case 0x5: {
      bool not_borrow = m_regs.at(x) >= m_regs.at(y);
      m_regs[x] -= m_regs.at(y);
      m_regs[0xF] = not_borrow; // As octo does.
    } break;

    case 0x6: {
      auto src_idx = m_quirk_shift ? x : y;
      auto vsrc = m_regs.at(src_idx);

      m_regs[x] = vsrc >> 1u;
      m_regs[0xF] = vsrc & 1;
    } break;

    case 0x7: {
      bool not_borrow = m_regs.at(y) >= m_regs.at(x);

      m_regs[x] = m_regs.at(y) - m_regs.at(x);
      m_regs[0xF] = not_borrow;
    } break;

    case 0xE: {
      auto src_idx = m_quirk_shift ? x : y;
      auto vsrc = m_regs.at(src_idx);

      m_regs[x] = vsrc << 1u;
      m_regs[0xF] = vsrc >> 7u;
    } break;

    default:
      return NOT_IMPLEMENTED;
    }
    break;
  } break;

  case 0x9: {
    if (m_regs.at(x) != m_regs.at(y)) {
      m_pc += 4;
      return NO_ERROR;
    }
  } break;
```

```
case 0xA: {
  m_reg_I = nnn;
} break;

case 0xB: {
  m_pc = nnn + m_regs.at(0);
  return NO_ERROR;
} break;

case 0xC: {
  std::scoped_lock lk(random_mutex);
  m_regs[x] = random_generator() & kk;
  break;
}
case 0xD: {
  if ((m_reg_I + n) > MEMORY_SIZE) {
    return MEMORY_OVERFLOW;
  }
  uint8_t vx = m_regs.at(x);
  uint8_t vy = m_regs.at(y);
  m_regs[0xF] = 0;

  /* std::cout << "DRAW: I=" << m_reg_I << " x=" << (uint16_t)x << " y=" <<
   * (uint16_t)y << " vx=" << (int)vx << " vy=" << (int)vy << std::endl; */

  for (uint16_t mem_sprite_pos = m_reg_I, mem_sprite_end = m_reg_I + n,
                row = vy, row_end = vy + n;
       row < row_end; ++row, ++mem_sprite_pos) {
    uint8_t sprite_row_contents = m_mem.at(mem_sprite_pos & 0xFFF);

    // In drawing each line of the sprite, we will cross a byte-boundary if
    // VX isn't divisible by 8. Thus, we have to flip the bits in each byte
    // across the boundary carefully.

    // To figure out which bits go where, we shift the sprite row
    // in a uint16_t which should span both possible bytes that our
    // shifted should now store in its upper byte the bits that will be
    // flipped in the first byte and the bits that will be flipped in the
    // second byte in the lower byte.
    // For example, for vx=3, when the sprite row contents are 0x10011001,
    // shifted should be 0b0001001100100000.
    uint16_t shifted = static_cast<uint16_t>(sprite_row_contents) << 8;
    shifted >>= vx & 0b111;

    // Now we fill the bytes spanned by this sprite.
    size_t row_begin = (row & ROW_MASK) * ROW_SIZE;
    size_t first_idx = row_begin + ((vx >> 3) & ROW_OFFSET_MASK);
    size_t last_idx = row_begin + (((vx >> 3) + 1) & ROW_OFFSET_MASK);

    auto first_iter = m_display.begin() + first_idx;
    auto last_iter = m_display.begin() + last_idx;

    uint16_t display_bits =
        (static_cast<uint16_t>(*first_iter) << 8) | *last_iter;
    /* std::cout << "  first_idx=" << first_idx << " last_idx=" << last_idx <<
     * "\n   shifted=" << std::bitset<16>(shifted) << "\n      mask=" <<
     * std::bitset<16>(display_bits) << std::endl; */

    m_regs[0xF] |= !!(shifted & display_bits);

    display_bits ^= shifted;

    *first_iter = display_bits >> 8;
    *last_iter = display_bits & 0xFF;
  }
} break;

case 0xE:
  switch (kk) {
  case 0x9E: {
    if ((keystate >> m_regs.at(x)) & 1) {
      m_pc += 4;
      return NO_ERROR;
    }
  } break;
  case 0xA1: {
    if (!((keystate >> m_regs.at(x)) & 1)) {
      m_pc += 4;
      return NO_ERROR;
    }
  } break;
  default: {
    return NOT_IMPLEMENTED;
  }
  }
  break;

case 0xF:
  switch (kk) {
  case 0x07: {
    m_regs[x] = m_reg_DT;
  } break;

  case 0x0A: {
    if (keystate) {
      for (unsigned i = 0, mask = 1; i < 16; ++i, mask <<= 1) {
        if (mask & keystate) {
          m_regs[x] = i;
          break;
        }
      }
    } else {
      // Returning early means that PC isn't incremented
      return WAITING_FOR_KEYPRESS;
    }
  } break;

  case 0x15: {
    m_reg_DT = m_regs.at(x);
    } break;

    case 0x18: {
      m_reg_ST = m_regs.at(x);
    } break;

    case 0x1E: {
      m_reg_I += m_regs.at(x);
    } break;

    case 0x29: {
      m_reg_I = m_font_begin + (m_regs.at(x) * FONT_SPRITE_SIZE);
    } break;

    case 0x33: {
      auto vx = m_regs.at(x);
      m_mem[(m_reg_I + 2) & 0xFFF] = vx % 10;
      vx /= 10;
      m_mem[(m_reg_I + 1) & 0xFFF] = vx % 10;
      vx /= 10;
      m_mem[m_reg_I & 0xFFF] = vx % 10;
    } break;

    case 0x55: {
      for (unsigned i = 0; i <= x; ++i) {
        m_mem[(m_reg_I + i) & 0xFFF] = m_regs.at(i);
      }
      if (!m_quirk_load_store) {
        m_reg_I += x + 1;
      }
    } break;

    case 0x65: {
      for (unsigned i = 0; i <= x; ++i) {
        m_regs[i] = m_mem.at((m_reg_I + i) & 0xFFF);
      }
      if (!m_quirk_load_store) {
        m_reg_I += x + 1;
      }
    } break;
    default:
      return NOT_IMPLEMENTED;
    }
    break;
  }

  m_pc += 2;
  return NO_ERROR;
}

std::span<const uint16_t> statemachine::instruction_stack::const_view() const {
  return {c.begin(), c.size()};
}
```

/swproto/statemachine_test.cpp

```
#include <algorithm>
#include <bitset>
#include <functional>
#include <gtest/gtest.h>
#include <initializer_list>
#include <sstream>

#include "font.hpp"
#include "statemachine.hpp"

std::string regs_of(const statemachine &mach) {
  using namespace std;
  stringstream reg_summary;
  reg_summary << "Regs: ";
  for (uint8_t reg : mach.regs()) {
    reg_summary << hex << setw(2) << setfill('0') << (int)reg << " ";
  }

  return reg_summary.str();
}

std::string
disp_str(std::span<const uint8_t, statemachine::DISPLAY_SIZE> display) {
  using namespace std;
  stringstream ret;
  ret << "display:\n";
  for (unsigned row = 0; row < statemachine::DISPLAY_HEIGHT; ++row) {
    cout << ' ';
    for (unsigned row_offset = 0; row_offset < statemachine::ROW_SIZE;
         ++row_offset) {
      cout << bitset<8>(display[row * statemachine::ROW_SIZE + row_offset]);
    }
    cout << '\n';
  }

  return ret.str();
}

std::string mem_of(const statemachine &mach) {
  using namespace std;
  stringstream mem_str;
  mem_str << "mem:";
  unsigned i = 0;
  for (uint8_t val : mach.memory()) {
    if ((i++ % 32) == 0) {
      mem_str << "\n  ";
    }
    mem_str << hex << setw(2) << setfill('0') << (int)val << " ";
  }
  return mem_str.str();
}

bool is_zero(uint8_t x) { return x == 0; }
```

```
inline void
ASSERT_STEP(statemachine &mach, uint16_t keystate, bool tick,
            statemachine::status expected_status = statemachine::NO_ERROR) {
  using namespace std;

  auto next_instruction = mach.curr_instruction();
  auto resultant_status = mach.step(keystate, tick);

  ASSERT_EQ(resultant_status, expected_status)
      << "while executing opcode 0x" << hex << setfill('0') << setw(4)
      << next_instruction;
}

TEST(StateMachineTest, Test00EE_2nnn) {
  std::initializer_list<uint16_t> instructions = {
      0x2004, // CALL 0x004
      0x1002, // JP 0x002 (hang in place)
      0x2008, // CALL 0x008
      0x00EE, // RET (when executed, should jump to JP instruction)
      0x00EE  // RET (should jump to above RET)
  };

  statemachine machine(instructions);
  ASSERT_STEP(machine, 0, 0);
  ASSERT_EQ(machine.pc(), 0x004);
  ASSERT_STEP(machine, 0, 0);
  ASSERT_EQ(machine.pc(), 0x008);
  ASSERT_STEP(machine, 0, 0);
  ASSERT_EQ(machine.pc(), 0x006);
  ASSERT_STEP(machine, 0, 0);
  ASSERT_EQ(machine.pc(), 0x002);
}

TEST(StateMachineTest, Test6xkk_7xkk) {
  statemachine machine({
      0x6789, // LD V7, 0x89
      0x7710, // ADD V7, 0x10
      0x77FF  // ADD V7, 0x77
  });
  EXPECT_EQ(machine.regs()[0x7], 0);
  ASSERT_STEP(machine, 0, false);
  EXPECT_EQ(machine.regs()[0x7], 0x89);
  ASSERT_STEP(machine, 0, false);
  EXPECT_EQ(machine.regs()[0x7], 0x99);
  ASSERT_STEP(machine, 0, false);
  EXPECT_EQ(machine.regs()[0x7], 0x98);
}

TEST(StateMachineTest, Test1nnn) {
  std::array<uint8_t, statemachine::MEMORY_SIZE> mem;
  mem.fill(0x66); // Fill with LD V6, 0x66
  // The below program should jump between JP instructions
  // and thus should never execute LD V6, 0x66

  // at 0x000: JP 0x020
  mem[0x000] = 0x10;
  mem[0x001] = 0x20;
  // at 0x020: JP 0x070
  mem[0x020] = 0x10;
  mem[0x021] = 0x70;
  // at 0x070: JP 0x100
  mem[0x070] = 0x11;
  mem[0x071] = 0x00;
  // at 0x100: JP 0x000
  mem[0x100] = 0x10;
  mem[0x101] = 0x00;

  statemachine machine(mem);

  // Make sure we never execute LD V6, 0x66
  for (int i = 0; i < 100; ++i) {
    ASSERT_STEP(machine, 0, false);
    ASSERT_EQ(machine.regs()[0x6], 0);
  }

  // Make sure memory was unchanged for no good reason.
  const auto &new_mem = machine.memory();
  ASSERT_TRUE(equal(mem.begin(), mem.end(), new_mem.begin(), new_mem.end()));
}

TEST(StateMachineTest, Test3xkk_4xkk) {
  std::initializer_list<uint16_t> instructions = {
      0x3000, // SE V0 == 0 (Should skip)
      0x6155, // LD V1, 0x55
      0x3001, // SE V0 == 1 (Should not skip)
      0x6255, // LD V2, 0x55
      0x4001, // SNE V0 == 1 (Should skip)
      0x6355, // LD V3, 0x55
      0x4000, // SNE V0 == 0 (Should not skip)
      0x6455, // LD V4, 0x55
  };

  statemachine machine(instructions);

  for (unsigned i = 0; i < 6 /* 2 of 8 instructions should be skipped */; ++i) {
    ASSERT_STEP(machine, 0, false);
  }

  ASSERT_EQ(machine.pc(), instructions.size() * 2)
      << "PC should sit after the final instruction";

  ASSERT_EQ(machine.regs()[0x0], 0) << "V0 should not have changed from 0";
  ASSERT_EQ(machine.regs()[0x1], 0) << "V1 should not have changed from 0";
  ASSERT_EQ(machine.regs()[0x2], 0x55) << "V2 should have changed from 0";
  ASSERT_EQ(machine.regs()[0x3], 0) << "V1 should not have changed from 0";
  ASSERT_EQ(machine.regs()[0x4], 0x55) << "V2 should have changed from 0";
}

TEST(StateMachineTest, Test5xy0_9xy0) {
  std::initializer_list<uint16_t> instructions = {
      0x6101, // LD V1, 0x01

      0x5110, // SE V1, V1 (should skip)
      0x6A55, // LD VA, 0x55
      0x5020, // SE V0, V2 (should skip)
      0x6A55, // LD VA, 0x55
      0x9010, // SNE V0, V1 (should skip)
      0x6A55, // LD VA, 0x55

      0x9110, // SNE V1, V1 (should not skip)
      0x6B55, // LD VB, 0x55
      0x9020, // SNE V0, V2 (should not skip)
      0x6C55, // LD VC, 0x55
      0x5010, // SE V0, V1 (should not skip)
      0x6D55, // LD VD, 0x55
  };

  statemachine machine(instructions);

  // Run until complete
  unsigned i;
  for (i = 0; (i < 20) && (machine.memory()[machine.pc()] != 0); ++i) {
    ASSERT_STEP(machine, 0, false);
  }

  ASSERT_NE(i, 20); // Make sure we didn't go somewhere we're not meant to be.

  ASSERT_EQ(machine.regs()[0xA], 0); // VA should never have been set.
  // VB, VC, and VD should have been set.
  ASSERT_EQ(machine.regs()[0xB], 0x55);
  ASSERT_EQ(machine.regs()[0xC], 0x55);
  ASSERT_EQ(machine.regs()[0xD], 0x55);
}

TEST(StateMachineTest, Test8xy0) {
  std::initializer_list<uint16_t> instructions = {
      0x6144, // LD V1, 0x44
      0x8310, // LD V3, V1
  };
  statemachine machine(instructions);

  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);

  ASSERT_EQ(machine.regs()[0x0], 0x00); // V0 should never have been set.
  ASSERT_EQ(machine.regs()[0x1], 0x44); // V1 should be 0x44
  ASSERT_EQ(machine.regs()[0x2], 0x00); // V0 should never have been set.
  ASSERT_EQ(machine.regs()[0x3], 0x44); // V3 should match V1
  // Remainder should not have been set.
  for (int i = 0x4; i <= 0xF; ++i) {
    ASSERT_EQ(machine.regs()[i], 0x00);
  }
}

TEST(StateMachineTest, Test8xy1_7xy2_8xy3) {
  std::initializer_list<uint16_t> instructions = {
      0x6105, // LD V1, 0x05
      0x62A0, // LD V2, 0xA0

      0x6A55, // LD VA, 0x55
      0x6B55, // LD VB, 0x55
      0x6C55, // LD VC, 0x55
      0x6D55, // LD VD, 0x55
      0x6E55, // LD VE, 0x55
      0x6F55, // LD VF, 0x55

      0x8A11, // OR VA, V1
      0x8B21, // OR VB, V2
      0x8C12, // AND VC, V1
      0x8D22, // AND VD, V2
      0x8E13, // XOR VE, V1
      0x8F23, // XOR VF, V2
  };

  statemachine machine(instructions);

  for (unsigned i = 0; i < instructions.size(); ++i) {
    ASSERT_STEP(machine, 0, false);
  }

  // Check if VA-VF have intended values.
  ASSERT_EQ(machine.regs()[0xA], 0x55);
  ASSERT_EQ(machine.regs()[0xB], 0xF5);
  ASSERT_EQ(machine.regs()[0xC], 0x05);
  ASSERT_EQ(machine.regs()[0xD], 0x00);
  ASSERT_EQ(machine.regs()[0xE], 0x50);
  ASSERT_EQ(machine.regs()[0xF], 0xF5);
}

TEST(StateMachineTest, Test8xy4) {
  statemachine machine({
      0x6082, // LD V0, 0x82
      0x6102, // LD V1, 0x02
      0x8104, // ADD V1, V0
      0x82F0, // LD V2, VF (save VF for later)
      0x8004, // ADD V0, V0 (should overflow)
  });

  for (int i = 0; i < 5; ++i) {
    ASSERT_STEP(machine, 0, false);
  }

  ASSERT_EQ(machine.regs()[0x0], 0x04); // 0x82 + 0x82 = 0x104
  ASSERT_EQ(machine.regs()[0xF], 0x01); // Should have overflowed.
  ASSERT_EQ(machine.regs()[0x1], 0x84); // 0x82 + 0x02 = 0x84
  ASSERT_EQ(machine.regs()[0x2], 0x00); // Should not have overflowed.
}

TEST(StateMachineTest, Test8xy5) {
  statemachine machine({
      0x6092, // LD V0, 0x92
      0x6102, // LD V1, 0x02
      0x8015, // SUB V0, V1 (V0 = V0 - V1)
      0x8A00, // LD VA, V0 (save for later)
      0x8BF0, // LD VB, VF (save for later)
      0x6092, // LD V0, 0x92
      0x6102, // LD V1, 0x02
      0x8105, // SUB V1, V0 (V1 = V1 - V0)
  });

  for (int i = 0; i < 8; ++i) {
    ASSERT_STEP(machine, 0, false);
  }

  ASSERT_EQ(machine.regs()[0xA], 0x90); // 0x92 - 0x02 = 0x90
  ASSERT_EQ(machine.regs()[0xB], 0x01); // 0x92 - 0x02 does not borrow.
  ASSERT_EQ(machine.regs()[0x1], 0x70); // 0x02 - 0x92 = 0x70
  ASSERT_EQ(machine.regs()[0xF], 0x00); // 0x02 - 0x92 DOES borrow.
}

TEST(StateMachineTest, Test8xy6) {
  { // Cases where VF should not be set.
    //
    // We choose immediates with 1's in most significant bit place to make we're
    // the rightshifts aren't sign-extending.
    std::initializer_list<uint16_t> instructions{
        0x6082, // LD V0 0x82
        0x6184, // LD V1 0x84
        0x8016, // SHR V0, V1
    };

    { // Case with shift quirks.
      statemachine machine(instructions, {.quirk_shift = true});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 0x41);
      ASSERT_EQ(machine.regs()[0xF], 0);
    }
    { // Case without shift quirks.
      statemachine machine(instructions, {.quirk_shift = false});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 0x42);
      ASSERT_EQ(machine.regs()[0xF], 0);
    }
  } // Cases where VF should be set.
  {
    std::initializer_list<uint16_t> instructions{
        0x6083, // LD V0 0x83
        0x6185, // LD V1 0x85
        0x8016, // SHR V0, V1
    };
    { // Case with shift quirks.
      statemachine machine(instructions, {.quirk_shift = true});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 0x41);
      ASSERT_EQ(machine.regs()[0xF], 1);
    }
    { // Case without shift quirks.
      statemachine machine(instructions, {.quirk_shift = false});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 0x42);
      ASSERT_EQ(machine.regs()[0xF], 1);
    }
  }
}

// Nearly identical to test for 8xy5
TEST(StateMachineTest, Test8xy7) {
  std::initializer_list<uint16_t> instructions = {
      0x6092, // LD V0, 0x92
      0x6102, // LD V1, 0x02
      0x8017, // SUBN V0, V1 (V0 = V1 - V0)
      0x8A00, // LD VA, V0 (save for later)
      0x8BF0, // LD VB, VF (save for later)
      0x6092, // LD V0, 0x92
      0x6102, // LD V1, 0x02
      0x8107, // SUBN V1, V0 (V1 = V0 - V1)
  };
  statemachine machine(instructions);

  for (unsigned i = 0; i < instructions.size(); ++i) {
    ASSERT_STEP(machine, 0, false);
  }

  ASSERT_EQ(machine.regs()[0xA], 0x70); // 0x02 - 0x92 = 0x70
  ASSERT_EQ(machine.regs()[0xB], 0x00); // 0x02 - 0x92 DOES borrow.
  ASSERT_EQ(machine.regs()[0x1], 0x90); // 0x92 - 0x02 = 0x90
  ASSERT_EQ(machine.regs()[0xF], 0x01); // 0x92 - 0x02 DOES NOT borrow.
}

TEST(StateMachineTest, Test8xyE) {
  { // Cases where VF should not be set.
    std::initializer_list<uint16_t> instructions{
        0x6005, // LD V0 0x05
        0x6107, // LD V1 0x07
        0x801E, // SHL V0, V1
    };

    { // Case with shift quirks.
      statemachine machine(instructions, {.quirk_shift = true});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 10);
      ASSERT_EQ(machine.regs()[0xF], 0);
    }
    { // Case without shift quirks.
      statemachine machine(instructions, {.quirk_shift = false});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 14);
      ASSERT_EQ(machine.regs()[0xF], 0);
    }
  } // Cases where VF should be set.
  {
    std::initializer_list<uint16_t> instructions{
        0x6085, // LD V0 0x85
        0x6187, // LD V1 0x87
        0x801E, // SHL V0, V1
    };

    { // Case with shift quirks.
      statemachine machine(instructions, {.quirk_shift = true});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 10);
      ASSERT_EQ(machine.regs()[0xF], 1);
    }
    { // Case without shift quirks.
      statemachine machine(instructions, {.quirk_shift = false});
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0x0], 14);
      ASSERT_EQ(machine.regs()[0xF], 1);
    }
  }
}

TEST(StateMachineTest, TestEx9E) {
  {
    std::initializer_list<uint16_t> instructions = {
        0x6209, // LD V2, 0x09
        0xE29E, // SKP V2
        0x60FF, // LD V0, 0xFF
        0x0000, // NOOP
    };

    { // Should skip b/c exact match.
      statemachine machine(instructions);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 1u << 9u, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0], 0x00);
    }
    { // Should not skip because does not match.
      statemachine machine(instructions);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 1u << 2u, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0], 0xFF);
    }
  }
  { // Test unreasonable requests.
    std::initializer_list<uint16_t> instructions = {
        0x6277, // LD V2, 0x77
        0xE29E, // SKP V2
        0x60FF, // LD V0, 0xFF
        0x0000, // NOOP
    };
    // Should not because 0x77 is not a real key and thus is never pressed.
    statemachine machine(instructions);
    ASSERT_STEP(machine, 0, false);
    ASSERT_STEP(machine, 0xFFFF /* Press all available keys */, false);
    ASSERT_STEP(machine, 0, false);
    ASSERT_EQ(machine.regs()[0], 0xFF);
  }
}

TEST(StateMachineTest, TestExA1) {
  { // Test cases for reasonable inputs.
    std::initializer_list<uint16_t> instructions = {
        0x6209, // LD V2, 0x09
        0xE2A1, // SKNP V2
        0x60FF, // LD V0, 0xFF
        0x0000, // NOOP
    };

    { // Should not skip b/c exact match.
      statemachine machine(instructions);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 1u << 9u, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0], 0xFF);
    }

    { // Should skip because does not match.
      statemachine machine(instructions);
      ASSERT_STEP(machine, 0, false);
      ASSERT_STEP(machine, 1u << 2u, false);
      ASSERT_STEP(machine, 0, false);
      ASSERT_EQ(machine.regs()[0], 0x00);
    }
  }

  { // Test cases for unreasonable inputs.
    // Should skip because 0x77 is not a real key and thus is never pressed.
    std::initializer_list<uint16_t> instructions = {
        0x6277, // LD V2, 0x77
        0xE2A1, // SKNP V2
        0x60FF, // LD V0, 0xFF
        0x0000, // NOOP
    };
    statemachine machine(instructions);
    ASSERT_STEP(machine, 0, false);
    ASSERT_STEP(machine, 0xFFFF, false);
    ASSERT_STEP(machine, 0, false);
    ASSERT_EQ(machine.regs()[0], 0x00);
  }
}

TEST(StateMachineTest, TestAnnn_Fx1E) {
  std::initializer_list<uint16_t> instructions = {
      0xAF10, // LD I, 0xF10
      0x60E0, // LD V0, 0xE0
      0xF01E, // ADD I, V0
      0xF01E, // ADD I, V0 (should rollover)
      0x0000, // Do nothing (allows m_reg_I to get masked)
  };
  statemachine machine(instructions);

  ASSERT_STEP(machine, 0, false);
  ASSERT_EQ(machine.reg_I(), 0xF10);
  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);
  ASSERT_EQ(machine.reg_I(), 0xFF0);
  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);
  ASSERT_EQ(machine.reg_I(), 0x0D0);
}

TEST(StateMachineTest, TestFx0A) {
  std::initializer_list<uint16_t> instructions = {
      0xF00A, // LD V0, K
      0x0000, // NOOP
  };
  statemachine machine(instructions);

  // Spin a bit, making sure that the PC doesn't progress.
  for (unsigned i = 0; i < 100; ++i) {
    ASSERT_STEP(machine, 0, false, statemachine::WAITING_FOR_KEYPRESS);
    ASSERT_EQ(machine.pc(), 0) << "Machine should not progress.";
  }
  // Press key 7.
  ASSERT_STEP(machine, 1u << 7u, false);
  ASSERT_EQ(machine.pc(), 0x02) << "Machine should progress.";
  ASSERT_EQ(machine.regs()[0], 7);
}

TEST(StateMachineTest, TestFx29) {
  const uint16_t test_font_begin = 0x123;
  std::initializer_list<uint16_t> instructions = {
      0xF029, // LD F, V0
      0x6004, // LD V0, 0x4
      0xF029, // LD F, V0
  };
  statemachine machine(instructions, {.font_begin = test_font_begin});

  ASSERT_STEP(machine, 0, false);
  ASSERT_EQ(machine.reg_I(), test_font_begin);
  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);
  ASSERT_EQ(machine.reg_I(), test_font_begin + (4 * FONT_SPRITE_SIZE));
}

TEST(StateMachineTest, TestFx15_Fx18_Timers) {
  std::initializer_list<uint16_t> instructions = {
      0x6003, // LD V0, 0x03
      0x6102, // LD V1, 0x02
      0xF015, // LD DT, V0
      0xF118, // LD ST, V1

      0x0000, 0x0000, 0x0000, 0x0000 /* NO-OPs */
  };
  statemachine machine(instructions);

  // Make sure the values are loaded.
  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);
  ASSERT_EQ(machine.reg_DT(), 0x03);
  ASSERT_EQ(machine.reg_ST(), 0x02);

  // Make sure clock ticks to, but not past, 0.
  ASSERT_STEP(machine, 0, true);
  ASSERT_EQ(machine.reg_DT(), 0x02);
  ASSERT_EQ(machine.reg_ST(), 0x01);
  ASSERT_STEP(machine, 0, true);
  ASSERT_EQ(machine.reg_DT(), 0x01);
  ASSERT_EQ(machine.reg_ST(), 0x00);
  ASSERT_STEP(machine, 0, true);
  ASSERT_EQ(machine.reg_DT(), 0x00);
  ASSERT_EQ(machine.reg_ST(), 0x00);
}

TEST(StateMachineTest, TestFx33) {
  std::initializer_list<uint16_t> instructions = {
      0xAF10,        // LD I, 0xF10
      0x6000 | 123u, // LD V0, 123
      0xF033,        // LD B, V0
  };
  statemachine machine(instructions);

  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);
  ASSERT_STEP(machine, 0, false);

  ASSERT_EQ(machine.memory()[machine.reg_I()], 1);
  ASSERT_EQ(machine.memory()[machine.reg_I() + 1], 2);
  ASSERT_EQ(machine.memory()[machine.reg_I() + 2], 3);
}

TEST(StateMachineTest, TestFx55) {
  std::initializer_list<uint16_t> instructions = {
      // Fill all the registers with random bytes
      0xC0FF, 0xC1FF, 0xC2FF, 0xC3FF, 0xC4FF, 0xC5FF, 0xC6FF, 0xC7FF,
      0xC8FF, 0xC9FF, 0xCAFF, 0xCBFF, 0xCCFF, 0xCDFF, 0xCEFF, 0xCFFF,
      0xAE00, // LD I, 0xE00
      0xF755, // LD [I], V7
      0xAE00, // LD I, 0xE00
      0xFF55, // LD [I], VF
  };

  for (int quirk_load_store = 0; quirk_load_store <= 1; ++quirk_load_store) {
    statemachine machine(instructions,
                         {.quirk_load_store = !!quirk_load_store});

    // Execute first 16 ops (that fill up V0..VF with randomness).
    for (unsigned i = 0; i < 16; ++i) {
      ASSERT_STEP(machine, 0, false);
    }

    std::array<uint8_t, 16> random_bytes;
    std::copy_n(machine.regs().begin(), 16, random_bytes.begin());

    auto mem = machine.memory();
    auto mem_random_begin = mem.begin() + 0xE00;

    ASSERT_STEP(machine, 0, false); // Executes LD I, 0xE00
    ASSERT_STEP(machine, 0, false); // Executes LD [I], V7

    if (quirk_load_store) {
      ASSERT_EQ(machine.reg_I(), 0xE00);
    } else {
      ASSERT_EQ(machine.reg_I(), 0xE08);
    }

    // Make sure that V0..V7 matches mem[0xE00]..mem[0xE07].
    ASSERT_TRUE(
        equal(random_bytes.begin(), random_bytes.begin() + 8, mem_random_begin))
        << mem_of(machine);
    // Make sure that 0xE08..0xFFF is empty.
    ASSERT_TRUE(std::all_of(mem_random_begin + 8, mem.end(), is_zero))
        << mem_of(machine);

    ASSERT_STEP(machine, 0, false); // Executes LD I, 0xE00
    ASSERT_STEP(machine, 0, false); // Executes LD [I], VF

    // Make sure that V0..VF matches mem[0xE00]..mem[0xE0F].
    ASSERT_TRUE(
        equal(random_bytes.begin(), random_bytes.end(), mem_random_begin))
        << mem_of(machine);
    // Make sure that mem[0xE0F]..mem[0xFFF] is empty.
    ASSERT_TRUE(std::all_of(mem_random_begin + 16, mem.end(), is_zero))
        << mem_of(machine);

    // Make sure register contents untouched.
    ASSERT_TRUE(equal(random_bytes.begin(), random_bytes.end(),
                      machine.regs().begin()));
  }
}

TEST(StateMachineTest, TestFx65) {
  std::initializer_list<uint16_t> instructions = {
      // Nonsense will begin 4 instructions from now.
      0xA008, // LD I, 0x008
      0xF765, // LD V7, [I]
      0xA008, // LD I, 0x008
      0xFF65, // LD VF, [I]
      // Fill 16 bytes with nonsense.
      0xDEAD, 0xBEEF, 0xF00D, 0xF1CE, 0xC0DE, 0xFACE, 0xFEED, 0xF00D};

  for (int quirk_load_store = 0; quirk_load_store <= 1; ++quirk_load_store) {
    statemachine machine(instructions,
                         {.quirk_load_store = !!quirk_load_store});

    ASSERT_STEP(machine, 0, false); // Executes LD I, 0x006
    ASSERT_STEP(machine, 0, false); // Executes LD V7, [0x006]

    if (quirk_load_store) {
      ASSERT_EQ(machine.reg_I(), 0x008);
    } else {
      ASSERT_EQ(machine.reg_I(), 0x008 + 7 + 1);
    }

    {
      // Make sure V0..V7 contains the first 8 bytes of nonsense
      // and that the remainder are empty.
      std::array<uint8_t, 16> expected_regs{0xDE, 0xAD, 0xBE, 0xEF,
                                            0xF0, 0x0D, 0xF1, 0xCE};
      ASSERT_TRUE(equal(expected_regs.begin(), expected_regs.end(),
                        machine.regs().begin()))
          << regs_of(machine);
    }

    ASSERT_STEP(machine, 0, false); // Executes LD I, 0x006
    ASSERT_STEP(machine, 0, false); // Executes LD VF, [0x006]
    {
      // Make sure all the regs are the same nonsense.
      std::array<uint8_t, 16> expected_regs{0xDE, 0xAD, 0xBE, 0xEF, 0xF0, 0x0D,
                                            0xF1, 0xCE, 0xC0, 0xDE, 0xFA, 0xCE,
                                            0xFE, 0xED, 0xF0, 0x0D};
      // Now make sure the regs are completely filled up with nonsense.
      ASSERT_TRUE(equal(expected_regs.begin(), expected_regs.end(),
                        machine.regs().begin()))
          << regs_of(machine);
    }
  }
```

```
}

TEST(StateMachineTest, TestCxkk) {
  // Fill all the registers with random bytes masked with DB.
  std::initializer_list<uint16_t> instructions = {
      0xC0DB, 0xC1DB, 0xC2DB, 0xC3DB, 0xC4DB, 0xC5DB, 0xC6DB, 0xC7DB,
      0xC8DB, 0xC9DB, 0xCADB, 0xCBDB, 0xCCDB, 0xCDDB, 0xCEDB, 0xCFDB,
  };
  statemachine machine(instructions);

  for (unsigned i = 0; i < instructions.size(); ++i) {
    ASSERT_STEP(machine, 0, false);
    // Check mask.
    ASSERT_EQ(machine.regs()[i] & ~0xDB, 0x00);
  }

  // Make sure that not all the registers are identical by making sure
  // they don't all match the first value.
  auto machine_regs = machine.regs();
  ASSERT_TRUE(std::any_of(machine_regs.begin(), machine_regs.end(),
                          [&](auto x) { return x != machine_regs.front(); }));
}

TEST(StateMachineTest, Test00E0_Dxyn) {
  std::initializer_list<uint16_t> instructions = {
      // Some sample data to mess with.
      0x0FF0, 0x8001,
      0xD011, // DRW V0, V0, 1
      0xD011, // DRW V0, V0, 1 (should undo previous instruction)
      0x6002, // LD V0, 0x02
      0xD011, // DRW V0, V1, 1
      0x00E0, // CLS
      0x6035, // LD V0, 0x3A (58)

  };
  statemachine machine(instructions, {.pc = 0x004});
  {
    auto display = machine.display();
    ASSERT_TRUE(std::all_of(display.begin(), display.end(), is_zero));
  }

  {
    ASSERT_STEP(machine, 0, 0); // Executes DRW V0, V1, 1
    auto display = machine.display();
    std::array<uint8_t, statemachine::DISPLAY_SIZE> expected_display{0x0F};
    ASSERT_TRUE(
        std::equal(display.begin(), display.end(), expected_display.begin()))
        << disp_str(display);
    ASSERT_FALSE(machine.regs()[0xF]);
  }
  {
    ASSERT_STEP(machine, 0, 0); // Executes DRW V0, V1, 1
    auto display = machine.display();
    ASSERT_TRUE(std::all_of(display.begin(), display.end(), is_zero));
    ASSERT_TRUE(machine.regs()[0xF]);
  }
  {
    ASSERT_STEP(machine, 0, 0); // Executes LD V0, 0x02
    ASSERT_STEP(machine, 0, 0); // Executes DRW V0, V1, 1
    auto display = machine.display();
    std::array<uint8_t, statemachine::DISPLAY_SIZE> expected_display{
        0b00000011, 0b11000000};
    ASSERT_TRUE(
        std::equal(display.begin(), display.end(), expected_display.begin()))
        << disp_str(display);
    ASSERT_FALSE(machine.regs()[0xf]);
  }
  {
    ASSERT_STEP(machine, 0, 0); // Executes CLS
    auto display = machine.display();
    ASSERT_TRUE(std::all_of(display.begin(), display.end(), is_zero));
  }
  {
    ASSERT_STEP(machine, 0, 0); // Executes LDs.
    auto display = machine.display();
    ASSERT_TRUE(std::all_of(display.begin(), display.end(), is_zero));
  }
}
```

/swproto/emulator.cpp

```cpp
#include <SFML/Graphics.hpp>
#include <SFML/System/Vector2.hpp>
#include <SFML/Window/Event.hpp>
#include <SFML/Window/Keyboard.hpp>
#include <algorithm>
#include <bitset>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <optional>
#include <string>

#include "font.hpp"
#include "statemachine.hpp"

const size_t SCALING_FACTOR = 1700 / statemachine::DISPLAY_WIDTH;

std::string mem_of(const statemachine &mach) {
  using namespace std;
  stringstream mem_str;
  mem_str << "mem:";
  unsigned i = 0;
  for (uint8_t val : mach.memory()) {
    if ((i++ % 32) == 0) {
      mem_str << "\n  ";
    }
    mem_str << hex << setw(2) << setfill('0') << (int)val << " ";
  }
  return mem_str.str();
}

/// Reads file contents into CHIP8 memory and places fonts starting at 0x000.
std::optional<std::array<uint8_t, statemachine::MEMORY_SIZE>>
try_load(std::string path) {
  using namespace std;
  array<uint8_t, statemachine::MEMORY_SIZE> ret;
  copy(font.begin(), font.end(), ret.begin());
  ifstream fs(path, std::fstream::in);
  if (!fs.is_open()) {
    return nullopt;
  }

  fs.read(reinterpret_cast<char *>(ret.data() + statemachine::PROG_BEGIN),
          statemachine::MEMORY_SIZE - statemachine::PROG_BEGIN);

  // If the file doesn't fill up memory,
  // fill the remainder with 0's.
  if (fs.eof()) {
    auto mem_prog_begin = ret.begin() + statemachine::PROG_BEGIN + fs.gcount();
    std::fill(mem_prog_begin, ret.end(), 0);
  }

  // Make sure that the file is no larger than the available space.
  if (fs.peek() != ifstream::traits_type::eof()) {
    return nullopt;
  }

  return ret;
}

/// Returns true if it handled a KeyPressed or KeyReleased event.
bool update_keys(uint16_t &keystate, sf::Event &event) {
  if ((event.type != sf::Event::KeyPressed) &&
      (event.type != sf::Event::KeyReleased)) {
    return false;
  }

  uint16_t update;
  switch (event.key.code) {
  case sf::Keyboard::Num1:
    update = 1 << 0x1;
    break;
  case sf::Keyboard::Num2:
    update = 1 << 0x2;
    break;
  case sf::Keyboard::Num3:
    update = 1 << 0x3;
    break;
  case sf::Keyboard::Num4:
    update = 1 << 0xC;
    break;

  case sf::Keyboard::Q:
    update = 1 << 0x4;
    break;
  case sf::Keyboard::W:
    update = 1 << 0x5;
    break;
  case sf::Keyboard::E:
    update = 1 << 0x6;
    break;
  case sf::Keyboard::R:
    update = 1 << 0xD;
    break;

  case sf::Keyboard::A:
    update = 1 << 0x7;
    break;
  case sf::Keyboard::S:
    update = 1 << 0x8;
    break;
  case sf::Keyboard::D:
    update = 1 << 0x9;
    break;
  case sf::Keyboard::F:
    update = 1 << 0xE;
    break;

  case sf::Keyboard::Z:
    update = 1 << 0xA;
    break;
  case sf::Keyboard::X:
    update = 1 << 0x0;
    break;
  case sf::Keyboard::C:
    update = 1 << 0xB;
    break;
  case sf::Keyboard::V:
    update = 1 << 0xF;
    break;

  default:
    update = 0;
  }

  if (event.type == sf::Event::KeyPressed) {
    keystate |= update;
  } else {
    keystate &= ~update;
  }
  return true;
}

int main(int argc, char **argv) {
  using namespace std;

  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " <ROM.ch8>\n";
    return 1;
  }

  string path = argv[1];

  if (!path.ends_with(".ch8")) {
    cerr << "path has invalid extension (expected .ch8)\n";
    return 1;
  }

  auto possible_mem = try_load(path);
  if (!possible_mem.has_value()) {
    cerr << "Failed to open " << path << endl;
    return 1;
  }

  statemachine machine(*possible_mem, {
                                        .pc = 0x200,
                                        .font_begin = 0x000,
                                    });
  cout << mem_of(machine) << endl;

  cout << "Successfully loaded " << argv[1] << '\n';

  sf::RectangleShape pixel(sf::Vector2f(SCALING_FACTOR, SCALING_FACTOR));
```

```
sf::RenderWindow window(
    sf::VideoMode(SCALING_FACTOR * statemachine::DISPLAY_WIDTH,
                  SCALING_FACTOR * statemachine::DISPLAY_HEIGHT),
    "CHIP8 Emulator");

window.setFramerateLimit(60);

uint16_t keystate = 0;
int ret = 0;
while (window.isOpen()) {
  window.clear();

  for (int i = 0; (ret == 0) && (i < (700 / 60)); ++i) {
    // Process events before every machine step.
    for (sf::Event event; window.pollEvent(event);) {
      // Close window: exit
      if (event.type == sf::Event::Closed) {
        window.close();
      } else {
        update_keys(keystate, event);
      }
    }

    auto status = machine.step(keystate, i == 0 /* Only tick once. */);
    if (status < 0) {
      cerr << "machine reported error " << status << endl;
      window.close();
      ret = 1;
      break;
    }
  }

  auto display = machine.display();

  for (size_t y = 0; y < statemachine::DISPLAY_HEIGHT; ++y) {
    auto row_begin = y * statemachine::ROW_SIZE;
    for (size_t x = 0; x < statemachine::DISPLAY_WIDTH; ++x) {
      if (display.at(row_begin + (x / 8)) & (0x80 >> (x & 0b111))) {
        pixel.setPosition(
            sf::Vector2f(x * SCALING_FACTOR, y * SCALING_FACTOR));
        window.draw(pixel);
      }
    }
  }

  /*
  // Display pixel on bottom left if sound is "playing".
  if (machine.reg_ST()) {
    pixel.setPosition(0, statemachine::DISPLAY_HEIGHT * SCALING_FACTOR);
  }
  // Display pixel next position over if delay timer is counting.
  if (machine.reg_DT()) {
    pixel.setPosition(SCALING_FACTOR,
                      statemachine::DISPLAY_HEIGHT * SCALING_FACTOR);
  }
```

```
    */

    window.display();
  }
  return ret;
}
```

/swproto/CMakeLists.txlt

```
FetchContent_MakeAvailable(googletest)

find_package(SFML COMPONENTS graphics REQUIRED)

set(SWPROTO_LIBRARY_SOURCES statemachine.cpp statemachine.hpp font.cpp font.hpp)
add_executable(emulator emulator.cpp ${SWPROTO_LIBRARY_SOURCES})
set_property(TARGET emulator PROPERTY CXX_STANDARD 20)
set_property(TARGET emulator PROPERTY CXX_STANDARD_REQUIRED ON)
target_link_libraries(emulator sfml-graphics)


add_executable(statemachine_test statemachine_test.cpp ${SWPROTO_LIBRARY_SOURCES})
target_link_libraries(statemachine_test gtest_main)
set_property(TARGET statemachine_test PROPERTY CXX_STANDARD 20)
set_property(TARGET statemachine_test PROPERTY CXX_STANDARD_REQUIRED ON)
```

/sw/Makefile

```
ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := chip8.o

else

# We are being compiled as a module: use the Kernel build system

        KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
        PWD := $(shell pwd)

OBJECTS = chip8_start.o keyboard.o

default: module chip8_start

chip8_start: $(OBJECTS)
        g++ -o chip8 $(OBJECTS) -lusb-1.0

module:
        ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

clean:
        ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
        ${RM} hello


TARFILES = Makefile README vga_ball.h vga_ball.c hello.c
TARFILE = lab3-sw.tar.gz
.PHONY : tar
tar : $(TARFILE)

$(TARFILE) : $(TARFILES)
        tar zcfC $(TARFILE) .. $(TARFILES:%=lab3-sw/%)

endif
```

/sw/keyboard.hpp

```
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <libusb-1.0/libusb.h>

#define USB_HID_KEYBOARD_PROTOCOL 1

/* Modifier bits */
#define USB_LCTRL (1 << 0)
#define USB_LSHIFT (1 << 1)
#define USB_LALT (1 << 2)
#define USB_LGUI (1 << 3)
#define USB_RCTRL (1 << 4)
#define USB_RSHIFT (1 << 5)
#define USB_RALT (1 << 6)
#define USB_RGUI (1 << 7)

struct usb_keyboard_packet {
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keycode[6];
};

class Keyboard {

private:
  struct libusb_device_handle *keyboard;
  uint8_t endpoint_addr;
  uint8_t *endpoint_address = &endpoint_addr;

public:
  struct keys {
    int error;
    bool up : 1, down : 1, left : 1, right : 1;
    bool escape : 1, enter : 1;
    bool game1 : 1, game2 : 1, game3 : 1, game4 : 1, game5 : 1, game6 : 1;
    uint16_t keypad;
    uint8_t keycode[6];
  };
  Keyboard();
  bool find_keyboard();
  struct libusb_device_handle *openkeyboard(uint8_t *);
  keys get_keys();
};
#endif
```

/sw/keyboard.cpp

```
#include "keyboard.hpp"
#include <cstdlib>
#include <iostream>

#define ENTER 0x28
#define ESC 0x29
#define BACKSPACE 0x2A

#define LEFT 0x50
#define RIGHT 0x4F
#define UP 0x52
#define DOWN 0x51

#define FIVE 0x22
#define SIX 0x23
#define SEVEN 0x24
#define EIGHT 0x25
#define NINE 0x26
#define ZERO 0x27

Keyboard::Keyboard() {
  struct libusb_device_handle *keyboard = NULL;
  uint8_t endpoint_addr;
  uint8_t *endpoint_address = &endpoint_addr;
}
struct libusb_device_handle * Keyboard::openkeyboard(uint8_t *endpoint_address) {
  libusb_device **devs;
  struct libusb_device_handle *keyboard = NULL;
  struct libusb_device_descriptor desc;
  ssize_t num_devs, d;
  uint8_t i, k;

  /* Start the library */
  if ( libusb_init(NULL) < 0 ) {
    return NULL;
  }

  /* Enumerate all the attached USB devices */
  if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    return NULL;
  }

  /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */

  for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];
    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      return NULL;
    }

    if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {
      struct libusb_config_descriptor *config;
      libusb_get_config_descriptor(dev, 0, &config);
      for (i = 0 ; i < config->bNumInterfaces ; i++)
        for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
          const struct libusb_interface_descriptor *inter =
            config->interface[i].altsetting + k ;
          if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
               inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL) {
            int r;
            if ((r = libusb_open(dev, &keyboard)) != 0) {
              return NULL;
            }
            if (libusb_kernel_driver_active(keyboard,i))
              libusb_detach_kernel_driver(keyboard, i);
            libusb_set_auto_detach_kernel_driver(keyboard, i);
            if ((r = libusb_claim_interface(keyboard, i)) != 0) {
              return NULL;
            }
            *endpoint_address = inter->endpoint[0].bEndpointAddress;
            goto found;
          }
        }
    }
  }

 found:
  libusb_free_device_list(devs, 1);

  return keyboard;
}


bool Keyboard::find_keyboard() {
  keyboard = Keyboard::openkeyboard(endpoint_address);
  if (keyboard == NULL) return false;
  else return true;
}

Keyboard::keys Keyboard::get_keys() {
  struct keys pressed_keys;
  pressed_keys.keypad = 0;
  struct usb_keyboard_packet packet;
  int transferred;

  libusb_interrupt_transfer(keyboard, endpoint_addr, (unsigned char *)&packet,
                            sizeof(packet), &transferred, 0);
  if (transferred == sizeof(packet)) {
    for (int i = 0; i < 6; i++) {

      /* maps keyboard to chip8 keys as follows:
       * 1 2 3 4        1 2 3 C
       * q w e r        4 5 6 D
       * a s d f  --->  7 8 9 E
       * z x c v        A 0 B F
       */
      switch (packet.keycode[i]) {
      case 0x1E:
```

```
    pressed_keys.keypad |= 1 << 0x1;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x1F:
    pressed_keys.keypad |= 1 << 0x2;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x20:
    pressed_keys.keypad |= 1 << 0x3;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x21:
    pressed_keys.keypad |= 1 << 0xC;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x14:
    pressed_keys.keypad |= 1 << 0x4;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x1A:
    pressed_keys.keypad |= 1 << 0x5;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x08:
    pressed_keys.keypad |= 1 << 0x6;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x15:
    pressed_keys.keypad |= 1 << 0xD;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x04:
    pressed_keys.keypad |= 1 << 0x7;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x16:
    pressed_keys.keypad |= 1 << 0x8;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x07:
    pressed_keys.keypad |= 1 << 0x9;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x09:
    pressed_keys.keypad |= 1 << 0xE;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x1D:
    pressed_keys.keypad |= 1 << 0xA;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
  case 0x1B:
    pressed_keys.keypad |= 1 << 0x0;
    pressed_keys.keycode[i] = packet.keycode[i];
    break;
```

```
      case 0x06:
        pressed_keys.keypad |= 1 << 0xB;
        pressed_keys.keycode[i] = packet.keycode[i];
        break;
      case 0x19:
        pressed_keys.keypad |= 1 << 0xF;
        pressed_keys.keycode[i] = packet.keycode[i];
        break;
      case ESC:
        pressed_keys.escape = 1;
        break;
      case ENTER:
        pressed_keys.enter = 1;
        break;
      case LEFT:
        pressed_keys.left = 1;
        break;
      case RIGHT:
        pressed_keys.right = 1;
        break;
      case UP:
        pressed_keys.up = 1;
        break;
      case DOWN:
        pressed_keys.down = 1;
        break;
      case FIVE:
        pressed_keys.game1 = 1;
        break;
      case SIX:
        pressed_keys.game2 = 1;
        break;
      case SEVEN:
        pressed_keys.game3 = 1;
        break;
      case EIGHT:
        pressed_keys.game4 = 1;
        break;
      case NINE:
        pressed_keys.game5 = 1;
        break;
      case ZERO:
        pressed_keys.game6 = 1;
        break;
      }
    }
  }
  return pressed_keys;
}
```

/sw/chip8.h

```
#ifndef _CHIP8_H
#define _CHIP8_H

#ifdef __cplusplus
extern "C" {
#endif

#include <linux/ioctl.h>

#define CHIP8_MAGIC 'q'

typedef struct {
  unsigned int data;
  unsigned int addr;
} opcode;

/* ioctls and their arguments */
#define CHIP8_INSTR_WRITE _IOW(CHIP8_MAGIC, 1, opcode *)

#define CHIP_RAM_START 0x00
#define CHIP_RAM_END 0x800
#define DISK_RAM_START 0x800
#define DISK_RAM_END 0x880
#define KEYBOARD_ADDR 0xffe
#define CTRL_ADDR 0xfff

#ifdef __cplusplus
}
#endif
#endif
```

/sw/chip8.c

```
/* * Device driver for FPGA memory
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod chip8.ko
 *
 */

/* Adapted for use with CHIP8 Final Project
 * Elysia Witham
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "chip8.h"

#define DRIVER_NAME "chip8"
#define VIRT_OFF(x,a) ((x)+(a))
#define VIRT_OFF_LONG(x, a) ((x)+(2*(a)))

/*
 * Information about our device
 */
struct chip8_dev {
       struct resource res; /* Resource: our registers */
       void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_instr(unsigned int addr, unsigned int instr) {
    iowrite16(instr, VIRT_OFF_LONG(dev.virtbase, addr));
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long chip8_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
	opcode op;

	switch (cmd) {
	case CHIP8_INSTR_WRITE:
		if (copy_from_user(&op, (opcode *) arg,
				   sizeof(opcode)))
			return -EACCES;
		write_instr(op.addr, op.data);
		break;

	default:
		return -EINVAL;
	}

	return 0;
}

/* The operations our device knows how to do */
static const struct file_operations chip8_fops = {
	.owner		= THIS_MODULE,
	.unlocked_ioctl = chip8_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice chip8_misc_device = {
	.minor		= MISC_DYNAMIC_MINOR,
	.name		= DRIVER_NAME,
	.fops		= &chip8_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init chip8_probe(struct platform_device *pdev)
{
	int ret;

	/* Register ourselves as a misc device: creates /dev/chip8 */
	ret = misc_register(&chip8_misc_device);

	/* Get the address of our registers from the device tree */
	ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
	if (ret) {
		ret = -ENOENT;
		goto out_deregister;
	}

	/* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
			       DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	/* Set an initial mem val?  */

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&chip8_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int chip8_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&chip8_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id chip8_of_match[] = {
	{ .compatible = "csee4840,chip8-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, chip8_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver chip8_driver = {
	.driver	= {
		.name	= DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(chip8_of_match),
	},
	.remove	= __exit_p(chip8_remove),
};
```

```
/* Called when the module is loaded: set things up */
static int __init chip8_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&chip8_driver, chip8_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit chip8_exit(void)
{
	platform_driver_unregister(&chip8_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(chip8_init);
module_exit(chip8_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Elysia Witham, Columbia University");
MODULE_DESCRIPTION("CHIP8 driver");
```

/sw/chip8_start.cpp

```
/*
 *
 * CSEE 4840 CHIP8 Final Project for 2022
 *
 * Name/UNI: Elysia Witham (ew2632)
 */

#include "keyboard.hpp"
#include "chip8.h"
#include <iostream>
#include <cstdlib>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <cstring>
#include <csignal>

#define MEMORY_START 0x200
#define GAME_BEGIN 0x100
#define MEMORY_END 0x880

static int CHIP8_FONTSET[] =
       {
              0xF0, 0x90, 0x90, 0x90, 0xF0, //0
              0x20, 0x60, 0x20, 0x20, 0x70, //1
              0xF0, 0x10, 0xF0, 0x80, 0xF0, //2
              0xF0, 0x10, 0xF0, 0x10, 0xF0, //3
              0x90, 0x90, 0xF0, 0x10, 0x10, //4
              0xF0, 0x80, 0xF0, 0x10, 0xF0, //5
              0xF0, 0x80, 0xF0, 0x90, 0xF0, //6
              0xF0, 0x10, 0x20, 0x40, 0x40, //7
              0xF0, 0x90, 0xF0, 0x90, 0xF0, //8
              0xF0, 0x90, 0xF0, 0x10, 0xF0, //9
              0xF0, 0x90, 0xF0, 0x90, 0x90, //A
              0xE0, 0x90, 0xE0, 0x90, 0xE0, //B
              0xF0, 0x80, 0x80, 0x80, 0xF0, //C
              0xE0, 0x90, 0x90, 0x90, 0xE0, //D
              0xF0, 0x80, 0xF0, 0x80, 0xF0, //E
              0xF0, 0x80, 0xF0, 0x80, 0x80  //F
       };

/*
 * References:
 *
 * https://chip8.danirod.es/docs/current/manual/Compatible-ROM-formats.html
 * http://devernay.free.fr/hacks/chip8/C8TECH10.HTM#2.3
 * https://mats.sh/posts/chip8-fpga/#introduction
 * Labs 2 & 3, and other class material
 *
 */

int chip8_fd;
```

```
void close_prog(int signal) {
  close(chip8_fd);
  exit(0);
}

void send_opcode(opcode *op)
{
  if (ioctl(chip8_fd, CHIP8_INSTR_WRITE, op)) {
      perror("ioctl(CHIP8_INSTR_WRITE) failed");
      close_prog(-1);
  }
}

void set_mem(int address, int data) {
  opcode op;
  op.addr = address;
  op.data = data;
  send_opcode(&op);
}

void load_fonts() {
  for (int i = CHIP_RAM_START; i < 80; i++) {
    set_mem(i, CHIP8_FONTSET[i]);
  }
}

void reset_mem() {
  for (int i = CHIP_RAM_START; i < CHIP_RAM_END; ++i) {
    set_mem(i, 0);
  }
}

void load_rom(const char *game) {
  FILE *gamefile;
  char buf[4];
  char buf2[4];
  long filelen;
  long sz = MEMORY_END - MEMORY_START;
  int i;
  int val;
  int val2;
  int adjusted_val;

  gamefile = fopen(game, "r");
  fseek(gamefile, 0, SEEK_END);
  filelen = ftell(gamefile);
  rewind(gamefile);

  for (i = 0; i < (filelen / 2) && i < sz; i++) {
    fread((&buf), 1, 2, gamefile);
    fread((&buf2), 1, 2, gamefile);
    // convert buf to hex
    val = (int)strtol(buf, NULL, 16);
    val2 = (int)strtol(buf2, NULL, 16);
    adjusted_val = val << 8 | val2;
    set_mem(GAME_BEGIN + i, adjusted_val);
  }

  for (int j = i; j < sz; ++j) {
    set_mem(GAME_BEGIN + j, 0);
  }
  set_mem(CTRL_ADDR, 0xFFFF);
}

void send_input(char value) {
  opcode op;
  op.addr = KEYBOARD_ADDR;
  op.data = value;
  send_opcode(&op);
}

void reset_all() {
  reset_mem();
  for (i = DISPLAY_RAM_START; i < DISPLAY_RAM_END; ++i)
    set_mem(i, 0);
  load_fonts();
}

int main()
{

  /* Open the keyboard */
  Keyboard keyboard_obj;
  if (!keyboard_obj.find_keyboard()) {
    std::cerr << "Did not find a keyboard\n";
    exit(1);
  }
  std::cout << "Keyboard initialized\n";
  unsigned int chip8_instr;
  int in_game;

  /* set up chip 8 userspace */
  static const char filename[] = "/dev/chip8";
  std::cout << "CHIP8 Userspace program started\n";

  if ( (chip8_fd = open(filename, O_RDWR)) == -1) {
    std::cerr << "could not open /dev/chip8\n";
    return -1;
  }

  signal(SIGINT, close_prog);
  reset_all();
  load_rom("start.hex");
  /* Look for and handle keypresses */
  for (;;) {
    Keyboard::keys keys = keyboard_obj.get_keys();
    if (in_game) {
      if (keys.escape) {
        in_game = 0;
        reset_all();
        load_rom("start.hex");
        continue;
      }
      send_input(keys.keypad);
    }
    else if (keys.game1) {
      std::cout << "Pong selected\n";
      load_rom("roms/pong.hex");
      in_game = 1;
    }
    else if (keys.game2) {
      std::cout << "Tetris selected\n";
      load_rom("roms/tetris.hex");
      in_game = 1;
    }
    else if (keys.game3) {
      std::cout << "Space Invaders selected\n";
      load_rom("roms/space_invaders.hex");
      in_game = 1;
    }
    else if (keys.game4) {
      std::cout << "Soccer selected\n";
      load_rom("roms/soccer.hex");
      in_game = 1;
    }
    else if (keys.game5) {
      std::cout << "Cave selected\n";
      load_rom("cave.hex");
      in_game = 1;
    }
  }

  return 0;
}
```

Roms folder not included, can be found on our github