COLLEGE OF ENGINEERING

Electrical Engineering and Computer Science

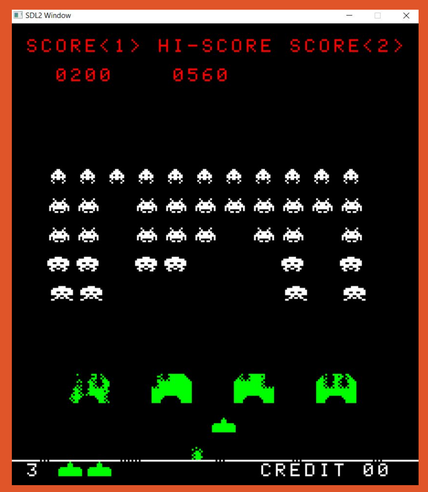
Space Invaders 8080

Craig Harris (harricra@oregonstate.edu)
Nicholas Herman (hermnich@oregonstate.edu)
Damian Russ (russda@oregonstate.edu)
Julie Weber (weberjul@oregonstate.edu)

Emulating an Intel 8080 CPU and Running the Space Invaders 8080 ROM

ABOUT THE GAME AND CPU

- Space Invaders (Taito, 1978) is a fixed shooter in which the player moves a laser cannon horizontally across the bottom of the screen and fires at aliens overhead. (Wiki)
- The CPU is the Intel 8080, which was the second 8-bit processor designed and manufactured by Intel. It was released in April 1974 and was an extended and enhanced variant of the 8008 design without binary compatibility. (Wiki)
- Space Invaders is one of the most emulated games ever. In part, due to its historical value as being the first original videogame. Moreover, a Space Invaders emulator can be used as a jumping off point for programmers interested in emulators in general. (Space Invaders Didactic Emulator, https://walkofmind.com/programming/side/side.htm)
- The current high score was set by Jon Tannahill at 218,870 points. He played on a 1978 Taito arcade cabinet and lasted 3 hours 47 minutes and 56 seconds. (Source: Guinness World Records)



Oregon State University

```c
State *Init8080(void)
{
    // Reserve ememory for the state struct
    State *state = calloc(1, sizeof(State));
    if (state == NULL)
    {
        printf("Error: State allocation failed.\n");
        exit(1);
    }

    // Reserve memory for the 16KB of RAM
    state->memory = calloc(MEM_SIZE, sizeof(uint8_t));
    if (state->memory == NULL)
    {
        printf("Error: State memory allocation failed.\n");
        free(state);
        exit(1);
    }

    // Clear the registers and set the initial program counter
    state->a = 0;
    state->b = 0;
    state->c = 0;
    state->e = 0;
    state->h = 0;
    state->l = 0;

    state->conditions.sign = 0;
    state->conditions.zero = 0;
    state->conditions.pad5 = 0; // always 0, per manual
    state->conditions.aux_carry = 0;
    state->conditions.pad3 = 0; // always 0, per manual
    state->conditions.parity = 0;
    state->conditions.pad1 = 0; // PUSH PSW will set this to 1.
    state->conditions.carry = 0;

    state->interrupt_enabled = 0;

    // The program ROM starts at 0x0000 in memory
    state->pc = 0x0000;

    // Stack pointer is initialized by the ROM after start
    state->sp = 0x0000;

    for (int i = 0; i < sizeof state->ports; i++)
    {
        state->ports[i] = 0;
    }

    return state;
}
```

Representation of the CPU state, including memory, registers, flags, program counter, stack pointer, interrupt, and ports; state.c

```c
case 0x3b: // DCX SP    SP = SP-1
{
    state->pc += opbytes;
    state->sp -= 1;
    wait_cycles(5);
    break;
}
case 0x3c: // INR A     A <- A+1
{
    state->pc += opbytes;
    state->a = increment_8b(state, state->a);
    wait_cycles(5);
    break;
}
case 0x3d: // DCR A     A <- A-1
{
    state->pc += opbytes;
    state->a = decrement_8b(state, state->a);
    wait_cycles(5);
    break;
}
case 0x3e: // MVI A, D8  A <- code[1]
{
    opbytes = 2;
    state->pc += opbytes;
    state->a = code[1];
    wait_cycles(7);
    break;
}
case 0x3f: // CMC   CY != CY
{
    state->pc += opbytes;
    state->conditions.carry = ~state->conditions.carry;
    wait_cycles(4);
    break;
}
case 0x40:  // MOV B,B
{
    mov_reg_to_reg(state, &state->b, &state->b);
    break;
}
case 0x41:  // MOV B,C
{
    mov_reg_to_reg(state, &state->b, &state->c);
    break;
}
case 0x42:  // MOV B,D
{
    mov_reg_to_reg(state, &state->b, &state->d);
    break;
}
```

A section of the emulated opcodes (0x3b - 0x42); shell.c

```c
void update_keyboard_input(State *state, SDL_Event *e)
{
  // Handle key press events
  if (e->type == SDL_KEYDOWN)
  {
      switch (e->key.keysym.sym)
      {
      case SDLK_c:     { state->ports[1] &= ~(1 << 0); break; } // Coin insert (active low)
      case SDLK_2:     { state->ports[1] |=  (1 << 1); break; } // P2 start button
      case SDLK_1:     { state->ports[1] |=  (1 << 2); break; } // P1 start button
      case SDLK_UP:
      case SDLK_SPACE: { // P1 & P2 shoot button
                         state->ports[1] |=  (1 << 4);
                         state->ports[2] = state->ports[1];
                         break;
                       }
      case SDLK_LEFT:  { // P1 & P2 joystick left
                         state->ports[1] |=  (1 << 5);
                         state->ports[2] = state->ports[1];
                         break;
                       }
      case SDLK_RIGHT: { // P1 & P2 joystick right
                         state->ports[1] |=  (1 << 6);
                         state->ports[2] = state->ports[1];
                         break;
                       }
      case SDLK_1:     { state->ports[2] |=  ((state->ports[2] + 1) % 4); break; } // num_lives
      case SDLK_t:     { state->ports[2] |=  (1 << 2); break; } // tilt 'button'
      case SDLK_b:     { state->ports[2] |=  (1 << 3); break; } // dipswitch bonus life
      case SDLK_i:     { state->ports[2] |=  (1 << 7); break; } // dipswitch coin info
      case SDLK_q:     { state->halt = 1; break;}
      default: break;
      }
  }
  // Handle key release events
  ~
```

Key Down events; controls.c

```c
int initialize_audio()
{

    // Set up the audio stream
    int errCode = Mix_OpenAudio(44100, AUDIO_S16, 1, 512);
    if (errCode < 0)
    {
        fprintf(stderr, "Unable to open audio: %s\n", SDL_GetError());
        return 1;
    }

    // Determine the number of mixing channels
    errCode = Mix_AllocateChannels(10);
    if (errCode < 0)
    {
        fprintf(stderr, "Unable to allocate mixing channels: %s\n", SDL_GetError());
        return 1;
    }

    // Load Samples
    for (int i = 0; i < NUM_WAVEFORMS; i++)
    {
        _sample[i] = Mix_LoadWAV(_waveFilePaths[i]);
        if (_sample[i] == NULL)
        {
            fprintf(stderr, "Unable to load wave file: %s\n", _waveFilePaths[i]);
            return 1;
        }
    }

    return 0;
}
```

Init Audio; sound.c

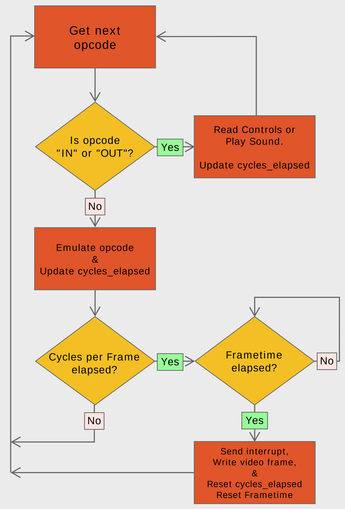
```c
emulate8080(state);

if (state->interrupt_enabled)
{
  if (midscreen_interrupt == 0 && cycles_elapsed > (cycles_per_frame/2.f))
  {
      wait_for_frametime_elapsed(1000000.f/120.f); // given in useconds;
      generate_interrupt(state, 1); // interrupt 1.
      midscreen_interrupt = 1;
  }
  else if (midscreen_interrupt == 1 && cycles_elapsed > cycles_per_frame)
  {
      wait_for_frametime_elapsed(1000000.f/120.f); // given in useconds;
      generate_interrupt(state, 2); // interrupt 2.
      midscreen_interrupt = 0;
      cycles_elapsed = 0;
     // Draw Screen.
      spinvaders_vram_matrix_to_surface(state, surface);
      SDL_UpdateWindowSurface(window);
  }
}
```

Mid & fullscreen interrupt logic; machine.c

```c
void machine_out(State *state, uint8_t port)
{
  switch (port)
  case 2: shift_offset = state->a & 0x7; break;
  case 3:
  {
      if ((state->a & 0b00000001) != 0) play_audio(0); // bit 0=UFO (repeats)
      else stop_audio(0);
      if ((state->a & 0b00000010) != 0) play_audio(1); // bit 1=Shot
      else stop_audio(1);
      if ((state->a & 0b00000100) != 0) play_audio(2); // bit 2=Flash (player die)
      if ((state->a & 0b00001000) != 0) play_audio(3); // bit 3=Invader die
      if ((state->a & 0b00010000) != 0) play_audio(9); // bit 4=Extended play
      break;
  }
  case 4: { shift0 = shift1; shift1 = state->a; break; }
  case 5:
  {
      if ((state->a & 0b00000001) != 0) play_audio(4); // bit 0=Fleet movement 1
      if ((state->a & 0b00000010) != 0) play_audio(5); // bit 1=Fleet movement 2
      if ((state->a & 0b00000100) != 0) play_audio(6); // bit 2=Fleet movement 3
      if ((state->a & 0b00001000) != 0) play_audio(7); // bit 3=Fleet movement 4
      if ((state->a & 0b00010000) != 0) play_audio(8); // bit 4=UFO Hit
      break;
  }
  default: break;
  }
}
```

machine_out() showing bit shift and audio output handling; machine.c



Control flow diagram.
Simplified to show only one screen interrupt. In reality there are are two: The midscreen and fullscreen interrupts. See details in Emulating the Hardware section below.

EMULATING THE 8080 CPU

The 8080 CPU emulator emulates all the opcodes described in the Intel 8080 Programming Manual. The 8-bit CPU allows for a potential 256 instructions, though the manual describes 78 instructions with bits of the opcodes reserved for specifying which registers the instructions operate on. To interpret the instructions, the emulator passes the 8-bit code into a switch statement and runs the function created to emulate the specific instruction being called.

The CPU state is also emulated, and is represented by a "State" struct. The State struct includes the accumulator (A) and registers (B, C, D, E, H, L), memory, flags, program counter, stack pointer, and input/outputs.

- The flag bit order is maintained, reading from left to right as sign, zero, pad value, aux carry, pad value, parity pad value, and carry. The values are stored in a "Conditions" struct.
- There are 65,536 direct-addressable bytes of memory. The game ROM and RAM both share the available memory space.
- Ports are tracked as an 8-byte array.

EMULATING THE HARDWARE

DISPLAY

- The screen updates at 60Hz. There is a **midscreen interrupt** and a **fullscreen interrupt**. These interrupts indicate to the game logic that either half of the screen is finished updating. The logic will then know it's safe to write to the portion of memory storing data for that part of the screen. This process ensures there are no concurrent read/writes from this section of memory and prevents *screen tearing*.
- The game logic relies on these interrupts arriving precisely once every **1/120th of a second** (two interrupts per frame), as they cause the decrement of an internal delay counter. The rate at which these interrupts arrive is directly related to the rate that the game will run, from the user's perspective.
- This emulator tracks each opcode's cycle count. When the number of cycles expected to occur within 1/120th of a second (based on the **2 MHz** speed of the processor) has elapsed, the mid or full screen interrupt is generated. The emulator also tracks time since the last interrupt. If 1/120th of a second hasn't elapsed, wait. This ensures the game doesn't run too fast on power computers.

CONTROLS

- Controls are run through ports 1 & 2, using c (insert coin), spacebar (shoot), and L/R arrows for movement.
- This consists of a function called get_keyboard_input that has key down and key up states to reflect when a button is pressed and released.

SOUND

- The original Space Invaders game used a complex sound chip to create the various sound effects used by the game. To simplify this, sound samples are used to emulate the original game audio, and played back when the game requests the different effects.
- The SDL2_mixer library is used to interface with the audio hardware. This library allows for multiple audio files to easily be mixed and sent to the output, emulating the original hardware's ability to play more than one sound effect at a time, a revolutionary technology for the time.