# MEMOCODE 2014 Software Design Contest: Space Invaders Emulator

Stephen A. Edwards
Department of Computer Science, Columbia University
New York, NY, USA
Email: sedwards@cs.columbia.edu

Hiren Patel
Electrical and Computer Engineering, University of Waterloo,
Waterloo, Ontario, Canada
Email: h.patel@ece.uwaterloo.ca

***Abstract*—The MEMOCODE design contest for 2014 was centered around the emulation of the 1978 Taito video game *Space Invaders.* The challenge is to improve the speed of a cycle-accurate software emulator for the game. Contestants had a month toope improve the provided code, which already ran fairly well on the ARM-based Raspberry Pi platform. Entries were judged on how much faster their code ran and its quality. The winning groups used a variety of optimization techniques ranging from dynamic binary translation, data-structure restructuring, and improving instruction and data caching.**

***Index Terms*—Instruction set simulator; Hardware emulation; Hardware/software codesign; Raspberry Pi;**

## I. Introduction

Since 2007, the organizers of the MEMOCODE Design Contest have proposed design problems and invited teams from around the world to come up with innovative hardware/-software solutions. Past years' problems have ranged from matrix multiplication to network simulation. This year, two problems were proposed: one focused on hardware and one focused on software.

This year's software challenge was to improve the speed of a processor system simulator, specifically one for Taito's wildly successful 1978 video arcade game *Space Invaders*. While this challenge was chosen in part for its amusement value, it is representative of a common challenge: designers often rely on an accurate system simulator to develop software while the target hardware is still under development. Like all simulators, such a simulator typically runs slower than the target hardware, making its optimization critical.

## II. Space Invaders

*Space Invaders* is a processor-based system consisting of an Intel 8080 CPU running at roughly 2 MHz, 8K of ROM, and 8K of RAM. Most of the memory (7K) is a simple one-bit-per-pixel framebuffer that delivers a 256 $\times$ 244 display to a black-and-white CRT. In the arcade cabinet, the CRT is tilted 90 degrees to deliver a portrait display and covered with a banded colored overlay that renders certain areas green and red.

The remaining I/O includes two byte-wide ports that indicate which buttons (left, right, and fire) the player has pressed, some ports that, when written, trigger the (analog) sound hardware (which is not emulated), and a barrel shifter that returns an arbitrary eight-bit segment of a 16-bit register, which the software uses to improve graphics rendering speed since the 8080 can only shift by one bit. In the game hardware, the shifter was implemented as a sixteen-bit register driving eight eight-input muxes.

## III. The Simulator

The contenstants were given a complete, working simulator based on the 8080 emulator from http://emulator101.com. The core of this is a function that simulates a single 8080 instruction:

```
int Emulate8080Op(State8080 *state) {
  uint8_t  *opcode = &state->memory[state->pc];
  ++state->pc;
  switch (*opcode) {
    case 0x00: break;           /* NOP */
    case 0x01:                  /* LXI B,word */
      state ->c = opcode[1];
      state ->b = opcode[2];
      state ->pc += 2;
     break;
    case 0x02:                  /* STAX B */
    /*  ...  */
  }
  return cycles8080[*opcode];
}
```

A struct State8080 holds the 8080 registers (a, b, etc.), a pointer to the base of the array representing memory, a pointer to a function that reads from an I/O port, and a pointer to a function that write to an I/O port.

## IV. Winning Contestants

Three contestants were successful in completing this software design contest. These contestants employed a variety of techniques to both discover and optimize performance bottlenecks in the emulator. We highlight a few of the many optimizations the contestants made in their contest submissions. The third place contestants focused their optimization efforts in improving the caching behavior by using techniques such as dead-code and data elimination and inlining. The second place contestants centered their efforts in optimizing function calls that were frequently invoked, and re-organizing data-structures to better match the underlying Raspberry Pi's hardware architecture. In addition to some of the above-mentioned optimizations, the winner of the contest used direct-threaded dispatch and dynamic binary translation techniques.