8080a Simulation:
Space Invaders

Prepared by
Anoakie Ray Turner

for
D. Pheanis
Arizona State University

Spring 2002

Table of Contents

Introduction

This document examines simulation of the 8080a processor and the *Space Invaders* ROM set. The introduction covers the abstract, the scope of the study, and the report format. This document assumes that the reader has access to the original source code and has an understanding of C and assembly. The terminology used in this document can be found in the definitions section.

Overview

My motivation behind this project is to learn how to simulate a processor with software, preferably focusing on simulating *Space Invaders*. To solve this problem, I researched ways in which other programmers wrote their simulators and from this research, determined the most beneficial structure for my simulator. I found static byte code interpretation to be the best method of simulation from a student's perspective, because it produces readable and understandable code. From this study, I learned that simulation was possible and how a simulator functions. It was an educational experience that forced me to apply most of my computer science knowledge to create the final product.

Scope of Study

The scope of this study was to gain a greater understanding of the 8080a processor and how it interfaced with program ROMs, specifically *Space Invaders*. The project's scope involved the following:

- Learning how the 8080a fetches and decodes instructions
- Learning how the 8080a reads program ROMs
- Learning how to correctly simulate video and I/O
- Writing an accurate simulator, despite poor documentation
- Writing a compatible and optimized simulator

Definitions

TTL – Dictionary.com defines TTL as “Transistor-transistor logic” (Dictionary.com/TTL). It relates to the switching time and power consumption of an integrated circuit.

Vectored interrupt – A vectored interrupt is an interrupt that the processor executes depending on the last three bits of the instruction. The processor multiplies these bits by eight and resets the PC to this value. i.e., If the processor senses interrupt three, it executes the RST3 instruction. The program counter is reset to byte 3*8 (byte 0x18h).

SDL (Simple DirectMedia Layer) – This is a cross-platform multimedia library (Simple DirectMedia Layer).

Blit – Blitting is the method of painting pixels to the screen. Dictionary.com defines blit as “To copy a large array of bits from one part of a computer's memory to another part, particularly when the memory is being used to determine what is shown on a display screen” (Dictionary.com/blit).

Attract Mode – Arcade games run demos of game play to attract users to the machine and entice users to play game.

History

Intel released it's third microprocessor, the 8080, in 1974. The 8080 is based on the original 8008 design, which in turn are based on the 4004 design. The 8080a is the same as the 8080 processor in every aspect, except for its TTL. The 8080a processor is compatible with standard TTL, while the 8080 is only compatible with low power TTL (“Shima Oral History”). The 8080a processor has enjoyed many years of success and is used in a variety of consumer electronics, including: personal computers (Altair 8800b “Computers by CPU”), health monitoring hardware (H8, “Love the H8”), and video games (Space Invaders).

The 8080a processor has a 16-bit bus, which addresses a maximum of 64k of memory, and an 8-bit data bus, which supports up to 256 inputs and outputs. It has ten 8-bit registers, which include three pairs of two 8-bit registers that the processor uses as 16-bit registers, an accumulator, a processor stats register (Fig. 1), and two temporary registers. It also features two 16-bit only registers, which it uses as a stack pointer and a program counter, respectively (“Intel 8080a CPU”) (Fig. 2). The processor's clock speed is 2Mhz and it supports up to eight vectored interrupts. The 8080a is also a little-endian processor.

Space Invaders' release date was four years after the 8080 processor's release, in 1978. It is an arcade game originally designed by Taito and licensed by Midway for US distribution. *Space Invaders*' has surpassed the popularity of all other arcade games in history. Upon its release in Japan, it caused a yen shortage, created a boom in arcade ownership, and incited juvenile crime ("*Space Invaders* – The Classic"). *Space Invaders* featured 8k of game data (ROM) and 1k of work RAM (see 8080avar.h) (Fig. 3). It also used an SN76477 analog/digital sound chip to generate complex sounds ("*Space Invaders* schematics"), a 224x256 resolution display (see graphics.c), and two 4-way joysticks, each with one button. Since it is an arcade machine, it also includes eight dip switches, and one coin slot (see input.c).

Conceptual Model

Proper implementation of a simulator depends on two primary elements: design and structure. I define design as a general overview the code's functionality and any additional considerations in the creation of the code, while structure refers to the way the a programmer forms code. Both design and structure enforce portability and maintenance.

Design is an integral part of a simulator. A poorly designed simulator can be difficult to maintain. Since this project needed to be presented on a multitude of operating systems, I gave portability the highest priority in the early stages of design. A cross-platform compiler with a cross-platform media library that offered a balance of performance and compatibility was the best choice. There were two options available: Java or C. I decided to use C because the performance penalty Java imposes at runtime, which does not offer a favorable environment for a simulator, and my familiarity with C. C does not come with a cross-platform media library, so I chose SDL as a dependency. To keep the simulator modular, I decided to implement a removable core. This makes simplifies maintenance and allows programmers to include the in other projects requiring 8080a simulation. (I will discuss this in more detail in the maintenance subsection below). There were several resource constraints, including: Processor speed (I targeted my 300Mhz

Pentium II Laptop), a small memory footprint (although much of this is reliant on the OS and the libraries the binary links to), and a target platform of x86 (I have tested the simulator on big-endian machines, like the PowerPC, and it does work). Other design constraints include: targeting *Space Invaders* as the main program that will be run with this simulator, using sound samples for sound generation (see the Sound section in "Supporting *Space Invaders*"), not using dynamic byte code translation (see byte code translation), and other constraints that I imposed as I wrote my first draft of the code. I cover these topics throughout this paper.

Correct structure creates a hospitable environment for future modifications and maintenance. The execution of the simulator is broken into three phases: initialization, execution, and cleanup. The first stage the simulator enters is the initialization phase. This phase is typically used for allocating memory and setting variables. During this phase, the simulator resets any global variables needing to be set, creates a graphics surface, initializes sound devices, and loads ROM sets. After this phase is complete, the simulator enters the main loop, where the actual simulation occurs. In the execution phase the simulator fetches an instruction, decodes the instruction, and executes the instruction. This action continues until a user force quits the loop. Once the main loop has finished, the simulator enters the cleanup stage. This stage

consists of freeing allocated memory and cleanly exiting the program (Fig. 4).

Documentation is a large part of the code's structure, and important for future revisions. Documentation for this project followed a strict design created for readability and understandability. Ryan Jones designed the original format for the documentation, and I adapted it to fit this project. I divided the documentation into two categories: C files and header files. At the top of each C file, proceeding the license, there is a heading named “Preface,” which quickly touches on the functions the file contains and what actions they perform. Each function contains a header that also maps out what actions it performs (Fig. 5). I documented every line of code in each function, along with any other additional definitions in the file which may fall out of the scope of a function (see any C file included with this document). At the top of each header file, after the license, there is a “Note” to inform the reader where documentation is located relating to the functions that I defined in the file. Following the note is a section named “Contents” that offers a quick overview of the functions contained in the file as well as any important global variables definitions. I concisely documented each function definition in the header file. This helps the programmer who intends to include these files in their project, but is only interested in a terse description (Fig. 6). I noted every declaration in the header file by its type (function, variable, etc.), followed by a description of its uses. 8080a Instructions follow a similar documentation pattern as functions, in which the name of the OPCODE is documented first, followed by a terse description of what the instruction does. Next is a verbose description covering all facets of the instruction's execution (Fig. 7a). Finally, there is a section named "Note" that includes extraneous information about the instruction, including any related instructions. If an instruction is related (the "child" of) to another instruction, the documentation is terse, while its "parent" gets full documentation (Fig. 7b). For more examples of documentation, please read the source files included with this report.

Maintenance is an important for future revisions of this simulator. I wrote this simulator with two maintenance concerns in mind, which were: the ease in which the simulation programmer can remove the core from the program, and the ability to add new functionality to the simulator. I wrote the simulation core as modular as possible. It depends on six files: 8080a.c, 8080a.h, 8080avar.h, 8080aio.c, 8080aio.h, and opcodes.h. 8080a.c contains the following functions: 8080amain, which function contains the simulation core, LoadRoms, which loads a ROM set from the disk, and ResetProc, which resets the processor. 8080a.h contains the function definitions for the functions defined above. 8080avar.h contains information about the 8080a processor and 8080a data structures. The opcodes.h file contains all of the 8080a opcode definitions and mnemonics. 8080aio.c handles the 8080a's input and output system with 8080ainput and 8080aoutput, respectively. 8080aio.h contains function definitions for the functions defined above. With these six files, the developer only needs to reimplement the ReadInput, UpdateScreen, and PlaySound functions to have a functional *Space Invaders* simulator. The developer may be interested in adding more functionality to the simulator or support more programs. By modifying the functions input8080a and output8080a in the file 8080aio.c, the developer can add and modify the inputs and outputs. To add new sounds, graphics, and input, the developer just needs to edit the ReadInput, UpdateScreen, and PlaySound functions. Modifications to the LoadRoms function may be necessary if the ROM set that is being loaded has odd sized ROM files. The source code requires no further modification to add support for new ROMs.

Simulating The 8080a Processor

Simulating the core of the 8080a processor required that I implement the following: instruction handling, memory, and timing.

There are three parts involved in handling instructions with the 8080a processor: the instruction table, fetch/decode/execute phase, and flags.

The instruction table is a fundamental component of a simulator. It contains the mnemonics for each instruction and that instruction's hexadecimal representation (see opcodes.h). The decode loop uses these mnemonics to determine what instruction to execute (see the switch statement in 8080a.c). Further optimizations are possible, which I will discuss in the corollaries section.

The program must fetch an instruction before it decodes it. During the fetch stage, the 8080a processor fetches instructions sequentially from the ROM and passes them to the decode phase. (I named the fetched instruction “OPCODE.” See 8080a.c). Once the program extracts the instruction from the ROM, the simulator switches on the value stored in the OPCODE. The switch contains case statements with mnemonics taken from the instruction table. If a match is found, the execute phase

begins. My execution phase involved setting flags, executing the instruction, and then resetting flags if needed (see the case statements in 8080a.c). If no match is found, the simulator fails.

Flag modification is the most time consuming task involved in instruction handling during processor simulation. Unlike the physical processor, in which it sets the flags by the time an instruction finishes executing, simulation flag setting consumes most of the time per instruction. For the 8080a, there may be as many as five flags that need to be set per instruction (see the PSW in 8080a.h and Fig. 1). To enforce the portability design constraint, I decided to take the slower route, setting the flags individually per instruction (see various case statements in main8080a in 8080a.c), but I did investigate another method that decreases the time it takes to set the flags. I discuss this alternative method in the corollaries section.

I had to implement memory management before instruction handling could occur. I divided the simulator's memory management issues as follows: register simulation and RAM/ROM simulation.

Simulating registers was a simple task. The 8080a processor has two types of registers: 8-bit registers and 16-bit registers. In order to simulate the 8-bit registers, I had to define an equivalent 8-bit type. I defined the accumulator A and

the processor status word PSW as an unsigned 8-bit integer to handle the 8-bit registers. There are only two 16-bit only registers on the 8080a processor: the stack pointer and the program counter. The remaining 16-bit registers are 8-bit register pairs: BC, DE, and HL, respectively. I used a union to handle the 16-bit register's 8-bit contents. With this union, I could access the register pair by requesting the the .pair of the union and the individual 8-bit registers by requesting .reg.H and .reg.L of the union. A union redefines the same piece of data, allowing the individual 8-bit registers to point to the same piece of memory that the 16-bit register resides in. This means that the 16-bit registers reflect changes made to one of its 8-bit contents. I defined the 16-bit registers as unsigned 16-bit integers (see the Variables section in 8080avar.h).

The 8080a processor can address up to 65536 bytes of memory. To simulate the 8080a's RAM and ROM, I defined a 0x10000 hex byte 8-bit unsigned integer array. Since there is no logical distinction between RAM and ROM in simulation, this memory is just a sequential block. The simulator can read the content of any address by just accessing RAM[address]. However, writing to memory with the simulator is a much more time consuming task. If the program is stored on a ROM chip instead of in RAM, the simulator has to be able to prevent write attempts to the ROM. I accomplished this task with the WriteRAM() macro, which tests to see if the offset is less than the WORKRAM address. If it is, the simulator is addressing ROM memory, and the write will not take place. The program consequently prints an error to the screen if the attempted write occurs. If the program is loaded in pure RAM, then it is safe to assume the simulator will use self-modifying code. There is a compiler switch to enable support for self-modifying code and skip write protection (see the WriteRAM macro in 8080a.h).

Accurate timing is essential in order to correctly simulate the 8080a processor. Counting cycles and calling interrupts at the proper time enforces timing.

Programmers of assembled software write 8080a programs with hardware constraints in mind. They attempt to use every cycle available and correctly time their code in order to maximize the amount of code executed before an interrupt occurs. Because of the time-based nature of interrupts, timing cannot be inaccurate by even a few cycles, or the simulator may fail to run correctly. To calculate the number of cycles per instruction, I created a table named CYCLES (see the CYCLES definition in opcodes.h). Each number in the table represents the number of cycles required to execute an instruction. The current instruction's opcode indexes this number. After the program fetches the instruction from the ROM, it retrieves the number of cycles for that instruction from the CYCLES table and adds it to a variable named COUNTER (see references to COUNTER in 8080a.c).

There is a small problem with this method; the program does not take extended or conditional instructions into account. Since the 8080a processor does not have documented extended instructions, I did not worry about handling extended instruction timing. Nonetheless, the processor does have conditional instructions. To handle these instructions, I set the number of cycles executed for a conditional instruction as if it did not meet the condition. Once the instruction starts to execute, if it meets the condition, the program adjusts the number of cycles the instruction consumes by the quantity of the number of the cycles taken as if the it met the condition minus the number of cycles taken as if the it missed the condition (see conditional instructions, like CALLs, in 8080a.c).

The 8080a's interrupts are hardware dependent. Since I was simulating *Space Invaders*, I knew what hardware I was writing for: a 60hz NTSC monitor and a 60hz I/O system, which would require two interrupts. Both interrupts occur every 34133 cycles. The first occurs after cycle 17066, and the second occurs after cycle 34132 (see the Interrupts timings section in 8080avar.h). I cover this in greater depth in the testing section.

By addressing instruction handling, memory management, and timing issues it is possible to have a functional 8080a core. First the simulator fetches an instruction from ROM and increases the number of cycles that it has executed by the time it takes to execute the fetched instruction. The simulator then decodes the instruction, then executes it, setting processor flags based on the result of the instruction. After execution, the simulator checks how many cycles it has been executed and if an interrupt needs to occur. Once this process is complete, the simulator returns to the fetch phase and the process repeats. (Fig. 4)

Supporting *Space Invaders*

Supporting specific ROM sets on the 8080a requires knowledge of how the ROM's program code uses the 8080a's memory and I/O. *Space Invaders* was my target ROM set, so I had to deal with video, input, and sound simulation.

There are three pieces involved in to simulating *Space Invaders*' video: determining where in the RAM the game stores the bitmap (Fig. 3), determining how to copy the array of bits to the host computers video memory, and deciding how to correctly color the pixels.

Space Invaders arcade uses a 224x256 resolution display. With the help of an arcade simulation FAQ (see the SOURCES file), I learned that the video RAM starts at location 0x2400 hex. With this information, I was able to determine that *Space Invaders* uses 0x1C00 hex bytes of video RAM by taking the y resolution of the video memory and multiplying it by the pitch of the video memory, which is 32. I arrived at the number 32 because the game is black and white; therefore, I used the equation (256 pixels wide)*(1/8 bytes per pixel). 32*224 is equal to 7168, or 0x1C00 hex bytes (see the UpdateScreen function in graphic.c and global definitions in global.h).

With Space Invaders' video RAM in mind, I moved onto blitting the pixels to the screen. Blitting is not as easy as it sounds; it sometimes requires pixel or surface manipulations to correctly blit the bits to the target surface. Space Invaders' video memory is rotated 90 degrees clockwise (Fig. 8a), so it must be rotated 270 degrees clockwise while the pixels are drawn to the screen (Fig. 8b). Since video operations are the slowest part of a graphic program, I took the liberty of optimizing the pixel plotting algorithm so I could squeeze as much performance out of the method as possible. By taking advantage of bit shifts, the way 8080a stores its video memory (this is a linear group that contains multiple off bits in a row), and only plotting 'on' pixels, I was able to increase the speed of the blits (Fig. 9). With these assumptions, I was able to decrease the time it took to perform a screen update, compared to updating the screen by linearly plotting every pixel in memory (see the UpdateScreen function in graphics.c).

As previously mentioned *Space Invaders* was a black and white game, using only one bit per pixel; however, many arcades had color versions of the traditionally black and white *Space Invaders*. These interesting hybrid machines were created by laying green and red plastic over the monitor to give the illusion of color. These color layers make the player's score appear red, the invaders appear white, and the base appear green. I have included this option in my simulator by coloring pixels on the top and bottom of the screen to get the desired effect (see the UpdateScreen function in graphics.h). The user can toggle this effect by pressing the 'c' key on the keyboard during play, assuming they compile color support (see input.c and the Makefile).

Space Invaders uses discrete logic chips for I/O. For input, it polls a pair of external joysticks and calculates bitmaps. For output, it plays sounds and sets variables to create bitmaps.

The most important aspect of input and output is the bit-shift created bitmap (see testing section for more information). Shift amounts are loaded into three external variables: a left shift amount, a high order 8-bit left shift value, and a low order 8-bit left shift value. The simulator writes these variables during the 8080a's output phase. Once the correct variables have been set, *Space Invaders*' code calls an input to calculate a shifted value based on the data just written (Fig. 10). The game uses these bitmap shifts to draw the *Invaders* on the screen and calculate collisions. They are also used to move *Invaders* (see case 3 in input8080a and cases 2 and 4 in output8080a in 8080aio.c).

User input is broken into two sections: joystick input and dip switches. With the help of SDL, I was able to create an

array of currently pressed keys and map the joystick buttons to certain keys. Dip switches required me to create three static variables that stored the state of the switches. Since the user can toggle these switches any time, I toggled one of the three static variables every time a certain key was pressed. The dip switch values were added onto the joystick input and returned (see input.c). The way I handled player input slightly contradicts the conventional way processors handle it, which is reading the joystick during an interrupt. I moved the player input section into the I/O section to keep the code readable and modular, so when a player input interrupt occurs, the simulator does not read any input.

Sound is the trickiest part of simulating *Space Invaders*. *Space Invaders* boards have a special sound chip named the SN76477 complex sound generator. This chip generates complex analog and digital sounds and is difficult to simulate. Due to of the complexity of this chip, I chose to use samples for my sound simulation. However, I still encountered problems. The 8080a processor creates two types of outputs while playing sounds. There is a type 3 sound, which is analog and a type 5 sound, which is digital. Unfortunately, there is no defined starting and stopping point with the complex analog sound output, and the code will continue generating requests indefinitely. To handle this problem, I had to create a variable, named lastout, to track what complex sound is currently playing and to check if the game

requests two of the same complex sounds. If the game requests a complex sound and there are no other complex sounds playing, and lastout is not equal to the value of the new sound, the simulator queues it. If the game repeatedly requests a complex sound, the simulator ignores it. If the game requests a complex sound, and it has the value of 0x20, then I treated this as an "end of analog output" marker, which means lastout is set to zero. If lastout is equal to zero, the simulator automatically queues the next complex sound it encounters (Fig. 11). Without these changes, complex sounds repeat infinitely (see the QueueSound function in sound.c).

By adding video, input, and sound simulation to the core 8080a simulator, I was able to support the *Space Invaders* ROM set. While the 8080a core is running, if the program encounters an input or output instruction is encountered, it either reads *Space Invaders*' input or output. Depending on the value of the data read from the ROM, the simulator reads player input, plays a sound, or performs a bitmap shift. After the execute phase, and after every second interrupt, the program updates the screen. With these additions in place, I had a working 8080a simulator that I could use to play *Space Invaders*.

Corollaries

Every program balances performance and accuracy. Accuracy is important for a correctly functioning system, but sometimes the programmer must sacrifice accuracy to ensure the program performs well. In my simulator, I focused mainly on accuracy and readability, but I still worked to optimize performance given my constraints.

I had difficulty addressing processor flag handling. As stated earlier, setting flags can take up to five times as many operations as a normal instruction (see the ACIn instruction in 8080a.c). The 8080 chip turned out to be the base processor for all subsequent x86 processor generations. Intel followed the 8080 chip up with the 8088, and then the 8086, which both had instruction sets that were extremely similar to one another. Since the 586 (Pentium) processor is based on the original workings of the 8086, it is possible to use the x86's processor flags to simulate the 8080a's processor flags. This, of course, would mean that I would need to use assembly, and the program could possibly lose cross-platform portability. In the interest of obtaining the highest performing implementation of the 8080a processor, this tradeoff is essential. With this consideration, executing an instruction no longer takes five flag modifications, but the flag itself is set while the instruction executes.

I also considered another optimization option: dynamic byte code translation. With proper byte code translation, a translation program can rebuild ROM sets into a native program that is executable on a target processor. Since the x86 instruction set is so close to the 8080a instruction set, a byte for byte translation can occur. With this method, the translator builds a new executable based on a ROM set, which performs like a native x86 application. No interpretation of 8080a opcodes is necessary while the program is running because the translator interprets instructions ahead of time. This method simulates 8080a ROM sets more accurately and performs better than a runtime opcode interpreting simulator. However, I did not choose to program the simulator this way because my program would resemble an assembler more then a simulator.

I decided not to include extended instructions, such as those used by z80 processors, in order to keep less then 256 instructions in simulator's core. At that size, a compiler can turn the switch statement into a 8-bit vector addressable jump table, which will give a slight performance boost for each OPCODE the simulator executes (see 8080a.c).

I made many other small tradeoffs to achieve higher levels of performance. The user can disable most of these "tradeoffs" in the Makefile during compilation (see Makefile). Some of the tradeoffs include: Correct coloring, which allows

the painting of color overlays, Sleep cycles, which allows the processor to go into a sleep mode while syncing, undocumented 8080a instructions, which supports seven extra undocumented 8080a instructions, self-modifying code, which enables faster operation while writing RAM because the simulator does not perform any bounds checking, and verbose messages, which prints debug information while the program is running and will print instruction information if the program encounters an unknown OPCODE.

Testing

When I started writing my simulator, I used a large block of printf statements to display debugging information. In this block, I included the instruction's mnemonic, the current opcode, the program counter, the stack pointer, the program status word, all 8-bit registers, the 16-bit paired registers, and the content of the next three bytes of RAM. With this information I was able to detect if an instruction acted as I expected. Once I felt confident that my instructions where correct, I loaded more advanced ROM sets. The final ROM set that I tested was the *Space Invaders* set, which was 8192 bytes of 8080a code. I kept an indexed array of opcodes that the simulator had been recently executed, which helped me pinpoint bugs within my code. I also kept an indexed array of opcodes that the simulator executed over the life of the program, allowing me to observe which instructions the simulator was not executing, and to test them separately. Most of the bugs in my program, apart from typos, were flag related bugs. Some instructions required estimation (such as how flags will be set in certain cases), because the documentation was inadequate or even wrong at times. The DAA instruction was especially cumbersome because it required that I translate a hexadecimal number into its decimal equivalent, as well as requiring interpretation and modification of the carry flags (see the DAA instruction in 8080a.c).

Of course, I could not run the *Space Invaders* ROM set without interrupts. After trying to decipher why my first attempt at running *Space Invaders* failed, I realized that CPUs use interrupts for I/O processing. *Space Invaders* has two types on interrupts: a video interrupt and an input interrupt. I did not perform any input or output processing during the interrupts; I simply simulated them at the necessary time and called the input and output related functions when needed (see the end of 8080a.c). When I first implemented the interrupt system, I only knew about the video interrupt, which was monitor dependent. Initially, I assumed that an interrupt occurred once every 1000 instructions. Using this estimation, I only received an output for a few stray pixels. After some thought, calculated the interrupts by taking the speed of the processor, 2048000 cycles per second, and divided it by the number of times the monitor refreshed per second, 60hz. As a result, I discovered that an instruction needed to occur approximately every 34133 cycles. Consequently, I set the new interrupt timing system, which forced me to implement a cycles per instruction table (see the CYCLES definition in opcodes.h). These new modifications gave me recognizable output. Unfortunately, this method only sustained the system for a few seconds before freezing. I did some further investigation and learned about NMI interrupts. I determined that *Space Invaders* needed this for polling the joysticks. I setup another interrupt, and estimated that it too needed to be called every 34133 cycles, but opposite of the video interrupt, otherwise they would conflict. I divided the number of cycles per interrupt by two, and I determined that approximately 17066 cycles were needed per interrupt. With the input interrupts attached, I was able to get graphics and a start-up screen.

Once I had my interrupt system fully functional, I moved onto the I/O. Testing I/O is extremely difficult because most of the I/O for the 8080a is ROM specific. *Space Invaders'* bitmap shifts gave me the most trouble before I learned how they functioned. Before a bitmap shift takes place, 16-bits of data are loaded into the bitmap output. I encountered some trouble while loading the 16-bits of data required by this output because an 8080a output uses the data contained in the 8-bit accumulator. After trial and error, I learned that the bitmap shifter required two bytes in successive order, or in other words, the game calls this output two times in a row. When the simulator writes a byte to the bitmap shifter, it takes the value of the last byte written to it and stores that as the most significant byte, using the current output as the least significant byte. There is also a bitmap-shift output used to set the left shift value for bitmap shifts (see cases two and four for 8080aoutput in 8080aoutput.c). If the game requests input from the bitmap shifter, this input shifts the 16-bit value (see case two for 8080ainput in 8080aio.c). The simulator then returns the least significant byte of this shift (Fig. 10). This shifter is what allows *Space Invaders* to function. Without this shifter, the game cannot draw the Invaders, cannot test for collisions, and crashes if the user leaves attract mode.

Testing my simulator was more of a trial and error process than the straightforward process that I have encountered while debugging higher level languages. In order to find out if my testing was successful, I acquired a copy of *Invaders Revenge*, the sequel to *Space Invaders*, and tried it on my simulator without any modifications to the code. The game ran correctly, except for the sound sets, which would have to be modified by hand to accommodate other games and programs.

Conclusion

The success of this project was a direct result of many hours of problem solving, trial and error, and research. This project allowed me to create a functional 8080a simulator that supports *Space Invaders* ROMs. I have learned many things from this project, such as how microprocessor simulation works and the importance of documentation.

References

Computers by CPU. 10 Apr. 2002. <http://www.geocities.com/~compcloset/ComputersbyCPU.htm>

Dictionary.com/blit. Dictionary.com. 10 Apr. 2002. <http://www.dictionary.com/search?q=blit>

Dictionary.com/microprocessor. Dictionary.com. 10 Apr. 2002. <http://www.dictionary.com/search?q=microprocessor&r=2>

Dictionary.com/TTL. Dictionary.com. 10 Apr. 2002. <http://www.dictionary.com/search?q=transistor-transistor%20logic>

Intel 8080a CPU. 23 July 2002. History of Computing Foundation. 10 Apr. 2002. <http://www.thocp.net/hardware/intel_8080a.htm>

Love the H8. 16 June 2001. 10 Apr. 2002. <http://home.attbi.com/~davidwallace2000/h8/Introduction.htm>

Simple DirectMedia Layer. 10 Apr. 2002. <http://www.libsdl.org/>

Shima Oral History. 17 May 1994. IEEE. 10 Apr. 2002. <http://www.ieee.org/organizations/history_center/oral_histories/transcripts/shima.html>

Space Invaders – *The Classic Arcade Games Shrine*. Retrogames.com. 10 Apr. 2002. <http://spaceinvaders.retrogames.com/html/Space_Invaders.html>

Space Invaders *schematics*. 1978. Taito. Space Invaders Color schematics can be found at <http://www.spies.com/arcade/schematics/>

Figures

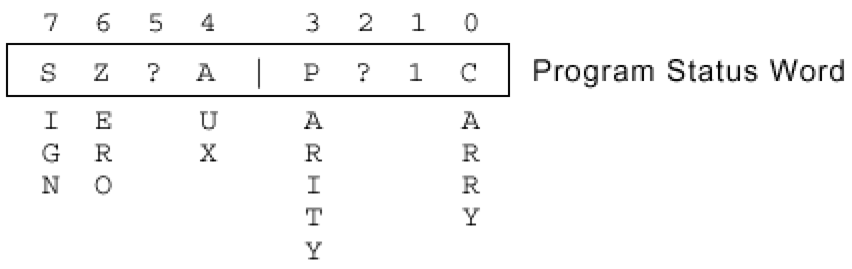


Fig. 1. The 8080a's Program Status Word.

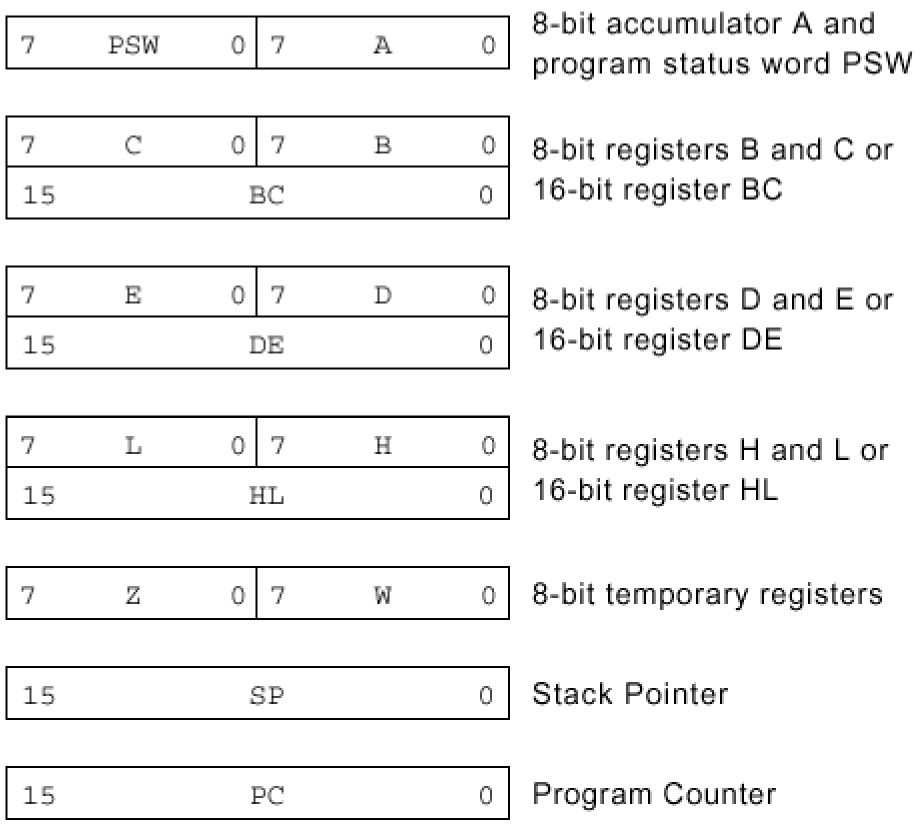
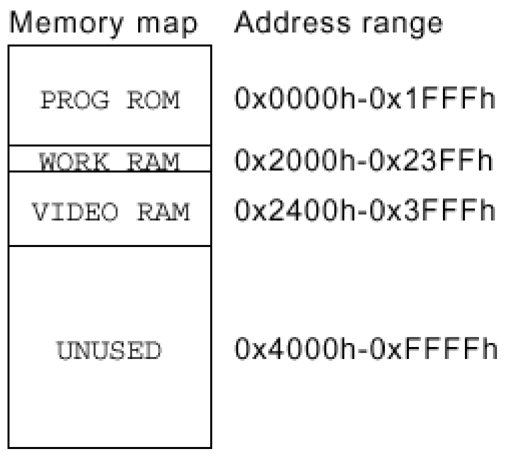

7 PSW 0 | 7 A 0
8-bit accumulator A and program status word PSW
7 C 0 | 7 B 0
15 BC 0
8-bit registers B and C or 16-bit register BC
7 E 0 | 7 D 0
15 DE 0
8-bit registers D and E or 16-bit register DE
7 L 0 | 7 H 0
15 HL 0
8-bit registers H and L or 16-bit register HL
7 Z 0 | 7 W 0
8-bit temporary registers
15 SP 0
Stack Pointer
15 PC 0
Program Counter

Fig. 2. The 8080a's registers. Note that the 8080a processor is little endian.

Fig. 3. The 8080a's memory map for *Space Invaders*


Memory map
Address range
PROG ROM
0x0000h-0x1FFFh
WORK RAM
0x2000h-0x23FFh
VIDEO RAM
0x2400h-0x3FFFh
UNUSED
0x4000h-0xFFFFh

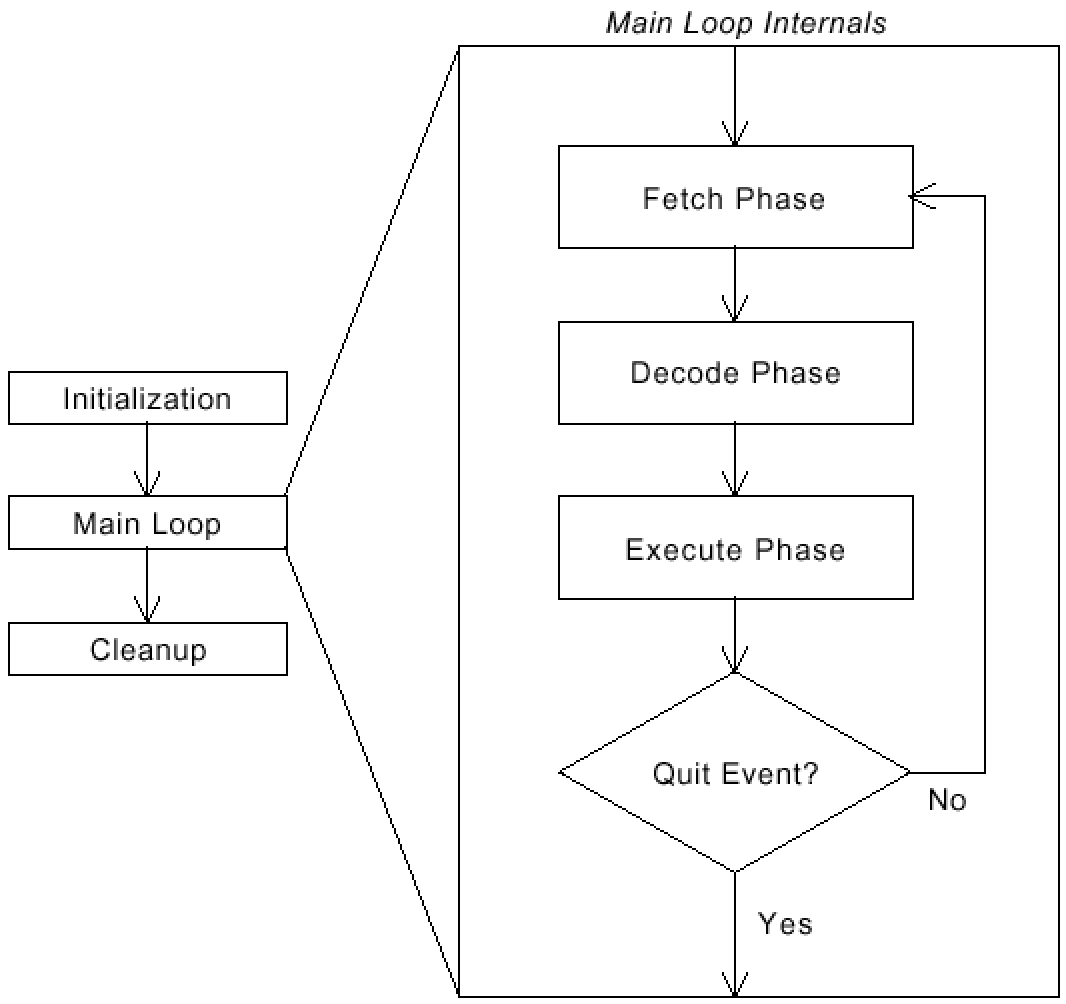


Fig. 4. The 8080a simulator's structure.

```
/***************************************************************************

"FunctionName":
--------------

   "Function description and verbose actions performed by function".


Input:
-----

   "VARIABLE" – "Description of VARIABLE".


Output:
------

   (return value) – "Description of return value".

***************************************************************************/
```

Fig. 5. C file source code example.

```
/***************************************************************************
*
* Function:
* --------
*
*   "FunctionName" - "Function description and actions performed
*                     by function".
*
***************************************************************************/
```

Fig. 6. Header file source code example.

```
/***************************************************************************
* "OPCODE" – "Terse Description"
* --------
*
*   "Verbose description".
*
* Note:
* ----
*
*   OPCODEs include ... (child instructions go here, if any exist).
***************************************************************************/
```

Fig. 7a. Documentation of the parent instruction.

```
/*****************************************************************************
*   If the value of the variable OPCODE is equal to "OPCODE",
* "shortened verbose parent description"
*****************************************************************************/
```

Fig. 7b. Documentation of the child instruction.

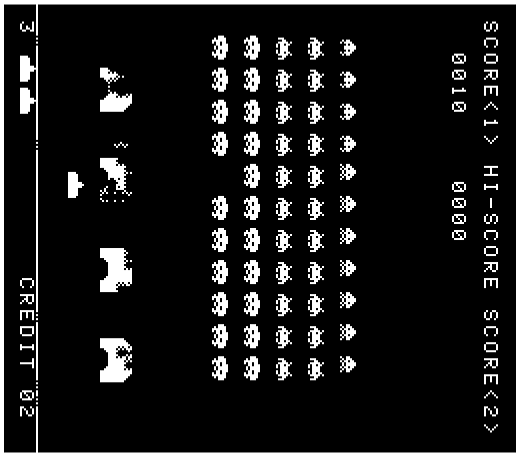


Fig. 8a. A snapshot of *Space Invaders*' video RAM during play. Note that black pixels are 'off' (0) bits and white pixels are 'on' (1) bits.

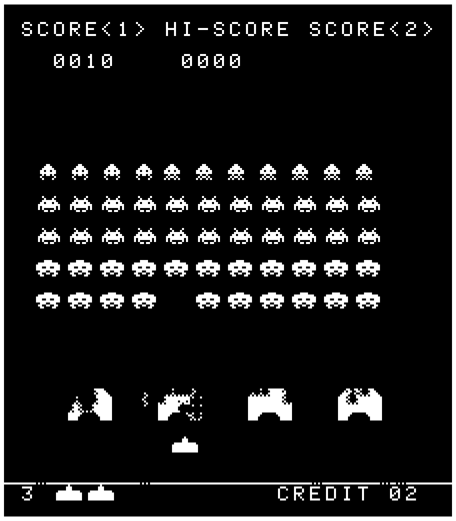


Fig. 8b. A snapshot of *Space Invaders*' video RAM during play after 270 degree rotation. Note that black pixels are 'off' (0) bits and white pixels are 'on' (1) bits.

	Old graphics routine	**New graphics routine**
Min	80 fps	145 fps
Mean	84.548 fps	157.308 fps
Median	84 fps	146 fps
Mode	84 fps	159 fps
Max	87 fps	170 fps

Fig. 9. Benchmarks of the graphics routines. 26 samples were taken while the simulator was running. See doc/Bench for data set and compilation options.

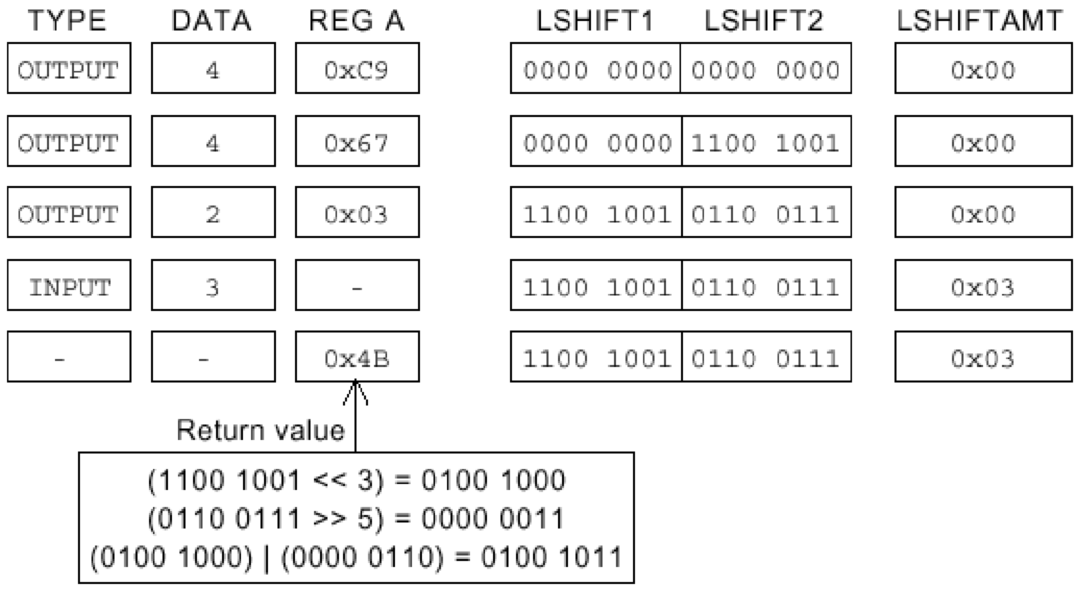


Fig. 10. This figure represents a sample trace of *Space Invaders*' bitmap shifting.

Type	ID	Complex	lastout	Action	Queue
3	0x04	Yes	0x00	Queue sound (lastout != ID)	0x00
3	0x08	No	0x04	Queue sound	0x04
3	0x04	Yes	0x04	Ignore sound, (lastout == ID)	0x04, 0x08
-	0x04	-	-	Sound finished, remove 0x04 from queue	0x04, 0x08
3	0x04	Yes	0x04	Ignore sound (lastout == ID)	0x08
3	0x20	No	0x04	Reset lastout	0x08
3	0x04	Yes	0x00	Queue sound (lastout != ID)	0x08
-	0x08	-	0x04	Sound finished, remove 0x08 from queue	0x04, 0x08

Fig. 11. This table represents a trace of sound requests generated by *Space Invaders*' OUTp instructions. Note that the variable lastout contains the last complex analog sound generated.