CS 210, Fall 2013
6502 Lab

Assigned: Tuesday Nov. 5

Part 1 Due: Nov. 12, Part 2 Due: Nov. 21 @ 1:30PM

1 Introduction

The year is 3020 and you are part of team that has unearthed a "cassette tape" (`http://en.wikipedia.org/wiki/Compact_Cassette`) that comes from the time before the "global nuclear war". It is believed that this material is an important part of human history and holds the key to understanding what the world was like before the war `http://en.wikipedia.org/wiki/A_Canticle_for_Leibowitz`).

Unfortunately no one knows what to do with it. One of your colleagues has been able to get the contents of the tape into a binary file. Your group has figured out that the file is a memory image from an early computer that used a device called the 6502. Copies of 6502 manuals have been located. Using these manuals, a predecessor of yours started writing an emulator so that the tape image could be interpreted and its operation and meaning determined. Sadly your predecessor reached the maximum age limit and was terminated on his birthday (`http://en.wikipedia.org/wiki/Logan's_Run`) before the emulator could be finished. As his apprentice the work has fallen on you. You have been promised that if you can accomplish this task before your birthday, in three weeks, your mandatory termination will be waived.

Good Luck and as they say you life depends on it!

2 Purpose

In this lab we will be putting your knowledge of how a computer works to recreate in software an early computer that operates as per the model we have discussed. Rather than building the computer physically, our goal will be to write a C program that, given a memory image, will operate like the computer we are trying to re-create.

3 How to Work on the Lab

YOU MAY WORK IN GROUPS OF TWO ON THIS ASSIGNMENT.

Download the file

`http://www.cs.bu.edu/fac/jappavoo/Resources/210/assignments/6502.tar`

Start by copying `handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf handout.tar`. This will cause a number of files to be unpacked into a directory called handout. You will be submitting a tar file of your updated version of this directory when you are done.

Your goal will be to create a program called '6502' that implements a simple emulator for a 6502 based computer that is invoked as follows:

```
./6502 <input.img> <output.img> [count]
```

The program reads in the file specified by the 'input.img' argument as the initial image of its memory and then executes based on that memory image. When it stops it will dump its current memory into an output file specified by the 'output.img'. Optionally you should be able to specify a count of instructions that the emulator should execute and then quit. If it is not specified then the emulator should execute until a 'BRK' instruction is encountered. You are given the basic infrastructure of the emulator that you will need to complete it inorder to get the provided memory image files working.

The first thing you should do is update the loop.c file with your team information. After this you should poke around the code and get a feeling for how the basic emulator infrastructure works. Your success largely depends on your ability to read, study and understand the source code you are given. The source code is larger and more complex (and thus more realistic) that other things we have looked at. You will need to develop the skill of how one reads and sorts out a larger body of source code. This is not a single pass process; you need to take notes, draw diagrams, jump around the code as you are reading, and use the debugger. You will find it helpful to talk it out on a whiteboard with your partner as you are trying to figure out how it works.

See the hints section below for an overview of the emulator and its design. Then work on getting the phases as discussed in the evaluation section completed. Note after the 'Compile' phase you will need to start using the 6502 documentation provide in the doc directory to implement the actual functioning of the emulator.

4 Evaluation

You will be evaluated on two parts, and each part is broken down into stages. Part 1 is due first and is comprised of stages that get you up and running on some simple memory images that have been constructed from simple assembly programs that exercise basic features of the processor and simple I/O subsystem of the computer. Part 2 is composed of stages that have you execute a considerably more complex unknown memory image. The makefile has a target for each stage that will test your emulator to see if it passes the stage. You can use these make targets to see if you have gotten things right. For each target you will either see it print out PASS or FAIL.

The stages are described below:

Part 1

1. **Compiling (10 Credits): 'make compile'**: Get the emulator to compile without any warnings. In other words when you run make, you should obtain an executable file named 6502.
2. **Memory Copy (20 Credits): 'make memcpy')**: Get your emulator to correctly execute the memcpy.img file provided. You will find in the apps directory test memory images along with the 'emp' image (emp.img) that you are ultimately trying to get working. One of the test images is a simple image that contains a program that copies a string in memory from one location to another. The 6502 source assembly used to create this image is in apps/memcpy.s along with a simple 6502 assembler (p65) that is used to create the image file from an assembly file ('memcpy.s' file). To get the memcpy image working you will need to get your main loop working and then implement the necessary functionality for the instructions used by the memcpy.s program. To test the emulator on the memcpy.img file by hand, use the following command:

   ```
   ./6502 apps/memcpy.img out.img
   ```

   To poke around the image files, use the provided **./img -h** image command.
3. **Console Copy (20 Credits): 'make concpy')**: Get your emulator to correctly execute the concpy.img file provided. To get this image file working you will need to complete hooking up the IO support in the emulator. The emulator has built-in infrastructure for a simple console device (console.[ch]) that allows characters to be read in from the keyboard and characters to be written to the screen. To accomplish this, the emulator supports the idea of memory mapped devices (mem.[ch]). Where we read or write a certain range of the simulated memory will cause operations on the IO devices. To get the image file working, you need to complete the **console_init** routine (which main invokes) to correctly map the console to the simulated address defined as **CONSOLE_ADDR** in console.h.
4. **Stack Operations (20 Credits): make 'stack')**: Get your emulator to correctly execute the stack.img file provided. To get this one working you will need to implement the instructions that push and pop values off the processors stack correctly plus any other supporting instructions required. When correctly implemented, the image should be able to read in characters and print them back out in reverse order (see the stack.s that was used to generate the stack.img).
5. **Subroutine Operation (20 Credits): make 'jsrrts')**: As we have learnt, a fundamental use of the stack is to implement procedure/functions. In particular a processor typically provides instructions that allow it to jump to a new location while returning back to the instruction after the place where the jump was executed. To accomplish this, a pair of instruction is used to coordinate the stack. In the case of the 6502, the instructions are `JSR` (jump to subroutine) and `RTS` (return from subroutine).

Part 2 You will find the 'C' source `emp.c` for a simple 'C' program that maintains a doubly linked list of employee objects. The program takes in commands that allow a user to operate on the list (including the ability to add items, delete items, and display the contents of the list). Read the `emp.c` source to understand how the emp binary image is to behave when your simulator executes it. For your convenience we have included the assembly file that our compiler generated ain `emp.s`

1. **Unknown 1 (100 Credits): 'make unknown1')**: You are now ready to start working on the unknown image. The first test will send the image a simple command to see that you have been

able to get the image started up. You may find it useful to use the provided **./img** command to dump parts of the image file. In particular you can use it to dump the reset vector to determine where in the image file the opcodes are located. You can then dump the opcodes using the same command. You may also find it useful to use one of the online 6502 disassemblers to get a dump of the actual 6502 assembly code that the unknown image file contains. GOOD LUCK AND HAVE FUN!

2. **Unknown 2 (BONUS Credits): 'make unknown2')**: Congratulations! If you have gotten this far you now know what the program in the unknown image is. The unknown2 target will test more of its functionality thus requiring you to implement more instructions in the emulator to pass this stage.

5 Handin Instructions

You will do two handins for this lab. Part 1 will be due on Nov 12 and Part 2 on Nov 21. In each case, do the following:

> **Create a tar file of your emulator directory called 6502.tar and then use gsubmit to submit your solution. If you have any questions about how to create the tar file speak with the TF.**

Before submitting, ensure that your solution behaves as expected on *csa2.bu.edu* as this is where we will grade your solution.

6 Hints

In this section we provide you with some background about the emulator and some general hints for this assignment. But perhaps the most important of all is that this assignment requires you to explore the problem using the knowledge you have learnt in this class. There is no prescribed cookie-cutter solution; you must be creative and inquisitive.

6.1 Emulator

An emulator is a program that is designed to mimic the behavior of an entire computer system. In our case, we are trying to construct a program that emulates a simple computer system that is based on a cpu called the MOS 6502 (which was used by several early home micro-computers such as the KIM-1, Apple I and II, PET, Vic-20, Commodore64 and others). Along with the CPU, our emulator needs to implement the memory contents and I/O capabilities of the simple computer. Memory can be easily emulated by an array that is indexed by the simulated addresses. I/O devices are generally more complicated to emulate, but we take a very simple approach to I/O that is discussed in the next sub-subsection.

In machine.h you will find the definition of the emulator's state. It defines a machine structure that contains the memory array along with the CPU state. The CPU state is define as a structure that contains the internal state used to implement the functioning of the CPU along with the registers that the 6502 programs can manipulate through the 6502 instructions. The 6502 is an 8-bit microprocessor with a 16-bit address bus. This means that all its data registers are 8-bits and its memory size is 2^{16} bytes big. As such, addresses are 2 bytes and data values are 1 byte. The machine.h file contains typedefs for byte and address. All of the CPU and memory states are defined using these two types.

The basic idea of the emulator is to implement the CPU's behavior by implementing a loop that fetches, decodes, and executes instructions from the simulated memory. Each instruction should create a change in the simulated machine state, either the simulated memory or simulated CPU state. You will need to complete the definition of this loop in loop.c. To do this you will find that the functions that implement fetch, decode and execute already exist but you must structure their invocation in loop.c. While these functions exist, the actual implementation of the 6502 instructions are not complete and the majority of your work will be to correctly implement the 6502 instructions that are needed by the memory images.

The loop function is invoked by main. In main, the processor state and memory is initialized so that the simulated 6502 can begin execution at the point of reset. As per the manual, when the 6502 is powered on, it follows a reset sequence that causes it to initialize the Program Counter (pc) register to the value stored at a specific location in memory called the reset vector. The machine.h file documents this location along with the other vector locations. The reset vector, specifically, is located at $0xFFFC$. The logic invoked in main prior to loop will load the input memory image in (as specified as the first argument to the program) and loads the pc with the address located at $0xFFFC$ and initialized all other registers as per the manual's specification. From that point on, the loop function controls the operation of the emulator. You may use the ./img command to dump the value of a memory image's reset vector location (./img restvec <file.img>) or you can set it with (./img resetvec <file.img> 0xHHLL) where HH is the high byte of the address and LL is the low byte of the address that you want to pc to be loaded with by the reset sequence. Be sure you know what you are doing if you change the resetvec of the provided images.

The fetch.c file defines a fetch function that you should not need to modify. The fetch function loads an internal register called the instruction register ('ir') with the byte located at the address stored in the pc. All opcodes for the 6502 are defined by a single byte. Next, the fetch function retrieves the necessary operands for the specified opcode. To do this, it consults a table that describes all the opcodes (OPCodeTable). The tables structure is described in instruction.h and initialized in instruction.c.

Each entry of the OPCodeTable gives the details for how to interpret each opcode. This includes the addressing mode and a pointer to a function which implements the actual behavior of the opcode (the instruction function). The 6502 supports 16 address modes described in the manuals. The address mode describes how to interpret the operand bytes (eg. are they an immediate constant, are they an address which contains the actual operand etc.). The instruction.c file contains empty functions for the address modes and instruction functions. The Opcode table has be setup so that these empty functions will automatically be invoked by the fetch, decode, and execute functions as discussed below. Your major effort will be to implement the details of the address mode functions and the instruction functions that are in instrument.c. BE WARNED DO NOT BLINDLY START IMPLEMENTING THEM. You should do this as an iterative process. All the default addressing and instruction functions will print out a message to standard error and terminate the emulator if they are invoked. This means that once you structure the loop function correctly you can start executing the emulator with a memory image and see what functions get triggered and need implementing to get the memory image processed.

The decode.c file contains a decode function that again you should not need to modify. The decode function forms the next stage of processing of an opcode. The decode function uses the current state of the cpu and the OPCodeTable to load internal state of the CPU with the information to conduct the execute step.

The execute.c file contains an execute function that you also should not need to modify. The execute function uses the state of the CPU that is set up by the fetch and decode functions. The execute function invokes the right instruction function in instruction.c to actually carry out the operation of the current instruction. The fetch function returns the return code from the instruction function back to the invoking function.

Again your loop function should use the fetch, decode and execute functions to implement the behavior of the CPU. You will likely want to return out of the loop if any of these functions returns a value less than 0. This will ensure that if any of the address mode functions or instruction functions return a failure, your emulator will quit. In particular, you will want to make sure that when a BRK instruction is executed, the emulator will halt. To do this, ensure that the BRK implementation returns a negative value back and that the loop exits based on this return code.

6.1.1 IO

In general, emulating IO devices can be very complicated. In our case, the emulator already has support for a simple model for interacting with IO devices. In particular, it supports Memory Mapped IO. What this means is that an IO device can take over one or more of the simulated memory addresses. When a program being executed by the emulator reads or writes these address they will be operating the device mapped to those addresses . As such, if the program are written to operate the devices by reading and writing the appropriate memory locations, the IO will occur as intend. In our case, the memory images that do IO expect that when they write a byte to address **0xFF00, CONSOLE_ADDR in console.h**, this will cause the byte to be written to the output device (eg. screen). In this way, if they want to write an ascii string to the screen they simply write a sequence of ascii byte values to the “memory mapped” console address. Similarly, the programs in the memory images expect that if they read from the same address, they will cause the emulator to read a character from the input device (eg. keyboard). The read will not return until a character is read. In this way, the programs can read data from the user.

All of the IO infrastructure is provided for you, but you must read the code and figure out how to complete the initialization so that the console code is mapped to the correct address.

When implementing the behavior of the instruction functions, you will want to use the provided inst_load and inst_store functions to ensure that the memory mapped device infrastructure is triggered on all memory operations. These routines check to see if the addresses being read or written are to any memory mapped addresses and, if so, are causing the appropriate device specific functions to be invoked.

6.2 GDB

As always, the debugger is your friend! However, things are a little more subtle when working on this project. Remember that GDB is used to debug the actual x86 6502 emulation program not the 6502 software running in the emulator. That being said, clearly the data structures and state of the emulator can be used to trace and debug the 6502 software. In this kind of situation, it is typical to use the programmability of GDB to define new commands that make your life easier. Specifically, GDB automatically looks for a file named .gdbinit in the directory you are in when executing GDB. In this file, you can define your own commands that then become available for your use within GDB. We have provided an example of such a file that has many useful commands. For instance, we can define a command sinfo that prints the contents of the 6502 registers in a nicely formatted way including dumping the sr bits in binary and disassembling the current instruction that the pc is pointing to. We encourage you to look at this file and use these commands. Here is some info to get you started.

6.2.1 .gdbinit file

In addition to looking at the contents of the .gdbinit, you can issue `help user` to see a list of all the user define commands we have provided. `help <cmd>` can be used to print a documentation that we have provided for a particular command `<cmd>`.

Some commands of particular interest are `strace, sstep, sinfo, sdisassemble, sxbyte, sxshort`.
In general, you will want to start GDB and then issue `strace` in order to use these commands. This will set a default breakpoint in the right spot to allow you to single step 6502 instruction execution via the `sstep` command. The following is an example of a typical GDB session:

```
$ gdb 6502
...
...Reading symbols for shared libraries .. done

gdb) help user
User-defined commands.
The commands in this class are those defined by the user.
Use the "define" command to define a command.

List of commands:

sbreak -- Set a simulated break point on pc == $arg0
sbt -- Print attempt to produce a back trace from 6502 stack page this
       may not alway work as there maybe other values pushed on the
       stack other than JSR addresses
sdisassemble -- Sdisassemble <addr> [count]
sinfo -- Pretty print register state
sirq -- Print the address of the installed irq interrupt service routine
snmi -- Print the address of the installed non-maskable interrupt service routine
sopdecode -- Print the details of opcode = $arg0
sreg -- Print 6502 registers
srst -- Print the address of the installed reset interrupt service routine
ssbyte -- Print cc65 argument stack as bytes
ssp -- Print cc65 argument stack pointer (initialized to 0xFEFF in crt0
ssshort -- Print cc65 argument stack as shorts/addresses
sstep -- Single step simulated instruction
strace -- Set breakpoint for single stepping simulated instructions
sxbyte -- Print value of simulated memory at address arg0
sxshort -- Print address located in memory at address $arg0

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) help strace
set breakpoint for single stepping simulated instructions

(gdb) strace
Breakpoint 1 at 0x1000015cc: file fetch.c, line 47.
(gdb) run apps/memcpy.img memcpy.oimg
Starting program: /Users/jonathan/Work/Projects/PSML/6502/6502 apps/memcpy.img memcpy.oimg
Reading symbols for shared libraries +............................ done

Breakpoint 1, fetch (m=0x10000b3c8) at fetch.c:47
47   m->cpu.ir = mem_get(m, m->cpu.reg.pc);

(gdb) sinfo
pc=0x0200 ac=0x00 x=0x00 y=0x00 sr=0x14 sp=0x0100 asp=0x0000 srbits:
$1 = 10100

(gdb) sd 0x200 10
0x0200: 0x00      : LDX 0x00   (#$BB)
```

```
0x0202: 0x00 0x03 : LDA 0x0300 ($HHLL,X)
0x0205: 0x00 0x04 : STA 0x0400 ($HHLL,X)
0x0208: 0x04      : BEQ 0x04   ($BB)
0x020a:           : INX
0x020b: 0x02 0x02 : JMP 0x0202 ($HHLL)
0x020e:           : BRK
0x020f:           : BRK
0x0210:           : BRK
0x0211:           : BRK

(gdb) sst

Breakpoint 1, fetch (m=0x10000b3c8) at fetch.c:47
47   m->cpu.ir = mem_get(m, m->cpu.reg.pc);
pc=0x0202 ac=0x00 x=0x00 y=0x00 sr=0x16 sp=0x0100 asp=0x0000 srbits:
$2 = 10110
0x0202: 0x00 0x03 : LDA 0x0300 ($HHLL,X)

(gdb) sxb 0x300 16
0x0300: 0x48 (H) 0x65 (e) 0x6c (l) 0x6c (l)
0x0304: 0x6f (o) 0x20 ( ) 0x57 (W) 0x6f (o)
0x0308: 0x72 (r) 0x6c (l) 0x64 (d) 0x00 ()
0x030c: 0x00 () 0x00 () 0x00 () 0x00 ()

(gdb) sst

Breakpoint 1, fetch (m=0x10000b3c8) at fetch.c:47
47   m->cpu.ir = mem_get(m, m->cpu.reg.pc);
pc=0x0205 ac=0x48 x=0x00 y=0x00 sr=0x14 sp=0x0100 asp=0x0000 srbits:
$3 = 10100
0x0205: 0x00 0x04 : STA 0x0400 ($HHLL,X)
(gdb)
```

6.3 Additional Info

- Study machine.h, loop.c, and instruction.[ch]; you will be spending a lot of time on them.
- You will want to figure out how to use the cpu.dbb and cpu.abr fields correctly.
- inst_load, inst_store routines in instruction.c should be used when implementing reads or writes of the emulated memory in the instruction functions to ensure that the memory mapped IO will work correctly.
- You may find it useful to run the emulator using the images by hand so that you can interact with it. You can do this by running the emulator directly. For example to directly run the emulator with the emp image you would issue the following command:

  ```
  ./6502 apps/emp.img myout.img 2> mytrace
  ```

  This will run your emulator (6502) in the current directory on the emp image file in the apps directory such that the final output memory image will be placed in a file called myout.img in the current directory and any output written by the emulator to standard error will be sent to the file mytrace in the current directory. If you are unsure about what mytrace is and how to use it ask the TF.
- Work incrementally.
- Get comfortable locating and interpreting the information in the manuals.
- Be sure to read the manual sections on how the 6502 zero-page and stack work.

- To get the jsrrts image file working you will need to get the JSR and RTS instructions working. These routine use the stack to implement subroutines. You will want to read the section in the programmers manual carefully to get these right. You will also want to trace and test the functioning of these carefully. The inst_push and inst_pop routines located in instruction.h will come in very handy.
- We have provide an img command that can be used to manipulate and dump the image files. You can use this to poke around the image files. Issue './img -h' to learn how to use it.
- In the apps directory is a simple 6502 assembler that you can use to create your own image files that contain opcode sequences that you want to test. If you do so, don't forget to set the reset vector of your new image to point at wherever you put your instruction sequence.

6.4 ASL,LSR,ROL,ROR

As a hint, I want to point out a subtlety to implementing 4 of the instructions that you will need to get right in order to get `unknown2` to work.

If you carefully study the 6502_A_4_gsX_.pdf, you will notice that the instructions break down according to their addressing modes such that most instructions support either a memory based operand that is specified by one of the following addressing modes:

```
ABS,         // abs: Absolute a
ABSX,        // abs,X: Absolute Indexed with X a,x
ABSY,        // abs,Y: Absolute Indexed with Y a,y
IMM,         // #: Immediate #
IND,         // ind: Absolute Indirect (a)
XIND,        // X,ind: Zero Page Index Indirect (zp,x)
INDY,        // ind,Y: Zero Page Indirect Indexed with Y (zp),y
REL,         // rel: Program Counter Relative r
ZP,          // zpg: Zero Page zp
ZPX,         // zpg,X: Zero Page Index with X
ZPY,         // zpg,Y: Zero Page Index with Y
```

or non-memory based oprands specified by the immediate addressing mode:

```
IMPL,        // impl: Implied i
```

but NOT both!

This means with the memory based ones, simply setting up the abr value in your address mode function will allow you to implement the instruction function by using the abr value. In the case of the implied, the instruction function can know that it does not need the abr value and is coded appropriately.

HOWEVER:

There are four instructions that support the Accumulator address mode, and these instructions also support memory based addressing modes:

```
  ACC,          // A: Accumulator A
```

specifically, the instructions are: `ASL, LSR, ROL, ROR`

When implementing these 4 instructions, you need to check to see what the address mode is of the opcode that triggered the instruction function to be invoked. You can then tell if you should use the abr to locate the operand in memory or if you should use the accumulator register. The following is a template of the kind of code you might use for these instructions:

```
  if (op->am->mode == ACC) {
     do accumulator based behavior here
  } else {
     do memory  based behavior here using the abr value
  }
```