Ben Badnani, James Ngo, William Wei, Ryan Velez
6502 Emulator Outline

File Index:
6502.c - contains main, is where the entire simulator is built and run.
6502.h - contains all macro definitions, function declarations, and structure definitions used by either void_functions.c or 6502.c.
void_functions.c - Contains all instruction-named functions and decodes, to be able to execute the correct instructions in RAM based off of the opcode passed in. Is linked with 6502.c to run the simulator.
test_vals.h - Created with the python script in '6502_test_opcodes_scraper.ipynb', uses

Emulator Outline
Computer struct : OurComputer

```
struct Computer{
   byte* RAM;
   struct cpu* cpu_inst;
   struct opcode_table *opcodes;
};
```

1 << 16 byte array : RAM
   a. Address space of 0x00 to 0xFFFF
   b. Little endian

CPU struct: cpu_inst

```
struct cpu{
   address pc;
   byte accumulator, register_x, register_y, status_register,
stack_pointer;
};
```

hash table: opcodes, each hash entry contains the opcode instruction byte as a key and a function pointer to its respective function stored in void_functions.c. The functions are not distinguished by addressing mode, that is the purpose of our 'decode' function in void_functions.c.

```
struct opcode_table{
   byte opcodes_key;
   void (*opcode_function)(byte, address);
   UT_hash_handle hh;
};
```

Stack

l. Address space of 0x100 to 0x1FF
m. Stack pointer is a byte (8 bits) and only points to the lower byte.
n. As seen in void_functions.c, when pushing, we first assign the value and then decrement the stack pointer, and vice versa for the stack pull. Utilized in BRK, JSR, RTS, RTI, PLA, PHA, PLP, PHP.

```
void stack_push(byte val){
 if(OurComputer->cpu_inst->stack_pointer == 0){
   printf("Stack full");
   exit(-1);
 }
 address stack_ptr = 1U << 8 |
OurComputer->cpu_inst->stack_pointer;
 OurComputer->RAM[stack_ptr] = val;
 OurComputer->cpu_inst->stack_pointer--;
}

byte stack_pull(){
 if(OurComputer->cpu_inst->stack_pointer == 0xFF){
   printf("Stack empty");
   exit(-1);
 }
 OurComputer->cpu_inst->stack_pointer++;
 address stack_ptr = 1U << 8 |
OurComputer->cpu_inst->stack_pointer;
 return OurComputer->RAM[stack_ptr];
}
```

Preliminary Setup:

1. For testing purposes, we have web scraped and formatted a binary image test file from http://www.qmtpro.com/~nes/misc/nestest.log, an NES emulator, which uses a 6502 CPU. We first split the html file, treated as a string, into seperate lines, and extracted indices for opcodes, addresses, and correct register values and program counter addresses for each corresponding instruction(s). We then write to 'test_vals.h' all the correct values for registers X,Y, accumulator, program counter address, and stack_pointer value as seperate arrays, consistently indexable by any one for loop.
2. Further, we write the opcode instructions as lines of 16 bytes to 'test_opcodes.txt', and then use xxd to convert this into a binary image 'test_opcodes.img'.

Initialization and CPU execution:

1. Executable testprogram is compiled with ">> make test".
    a. Executable takes one argument, a binary image file of 1<<16 byte values to be loaded into the RAM array in CPU_struct.

b. Upon instantiation, in main of 6502.c,
   i. We read in the binary image to the RAM array stored in our computer struct
   ii. Build the hash table in the computer struct by assigning to each value in the set of opcode values a pointer to a function that covers all its variations in terms of addressing modes.

```c
typedef void (*opcode_function)(byte, address);
opcode_function functions[] = {&ADC ,&BCC ,&BCS ,&BEQ ,&BIT ,&BIT ,&BMI ,&BNE ,&BPL ,&BVC ,&BVS ,&CMP ,&CMP ,&CMP ,&CMP ,&CMP ,&CMP ,&CMP ,&CMP ,&CPX ,&CPX, ... }
```

```c
struct opcode_table* s = NULL;
 for (int i = 0; i  < opcode_size; i++){
   s = (struct opcode_table*) malloc(sizeof(*s)); // check if NULL?
   if(s == NULL){
     printf("Memory allocation err");
     exit(-1);
   }
   s->opcodes_key = opcodes_keys[i]; // initializing key for s
   s->opcode_function = functions[i]; // initializing the value for s
   HASH_ADD(hh,OurComputer->opcodes, opcodes_key, sizeof(uint8_t),s); }
```

   iii. Initialize all registers in our computer, the accumulator, and the program counter to 0. We initialize the stack pointer to 0x01FF, per the guidelines of the 6502 manual. (We in fact initialize the stack pointer to FF, but use bitmasks and logical ors to have it represent 0x01FF, since its range only spans 2^8 - 1).
   iv. Start the cpu: Each execute call takes in an opcode in RAM pointed to by the program counter address, and the program counter address itself.

```c
void start_cpu(){
 while(1){
   execute(OurComputer->RAM[OurComputer->cpu_inst->pc], OurComputer->cpu_inst->pc);
 }
}
```

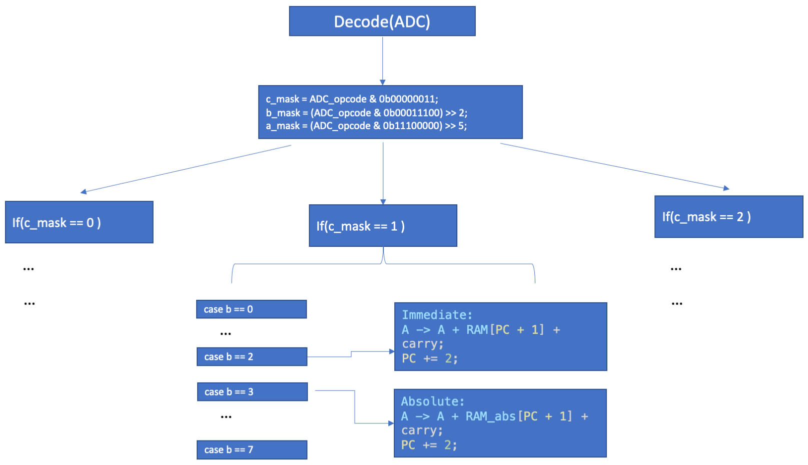
6. In the execute call we look up the function pointer pointed to by each specific opcode byte, and call that function in void_functions.c with the parameters passed into execute. The

functions themselves as well as decode are expanded on in the next section.

```c
void execute(byte opcode, address pc) {
 struct opcode_table *s; // used in execute(byte, address)
 HASH_FIND_BYTE(OurComputer->opcodes, &opcode, s);
 if(s == NULL){
   printf("Byte not in table \n");
   printf("opcode = %x \n", opcode);
   exit(-1);
 }
 (*s->opcode_function) (opcode, pc);
 return;
}
```

Decode() Proof of Concept:

1. Fetch is technically absent, but the idea is that an execute function contains all CPU capabilities of the cycle.
2. “Execute” arguments: Opcode(byte) in RAM at address held in program counter and the address(2 bytes) held in program counter itself.
3. Decode arguments: Opcode(byte)


Decode(ADC)
c_mask = ADC_opcode & 0b00000011;
b_mask = (ADC_opcode & 0b00011100) >> 2;
a_mask = (ADC_opcode & 0b11100000) >> 5;
If(c_mask == 0)
If(c_mask == 1)
If(c_mask == 2)
…
…
case b == 0
…
case b == 2
case b == 3
…
case b == 7
…
…

```
Immediate:
A -> A + RAM[PC + 1] +
carry;
PC += 2;
```

```
Absolute:
A -> A + RAM_abs[PC + 1] +
carry;
PC += 2;
```

4. Through the hash table, opcode functions are called with their corresponding opcode.
    a. Depending on the opcode byte itself, decode will find and/or set up the following:
        i. Addressing mode for argument and the argument itself
        ii. The PC address after the current working op code is executed
    b. There are 3 bit masks: a_mask, b_mask, c_mask to find bits of an opcode byte.
    c. Using these masks, we are able to effectively differentiate addressing modes, and write specific edge cases where not applicable.
    d. The return_ ret struct acts like a data bus of sorts
        i. holds ret.arg argument after using addressing modes to navigate memory
        ii. holds ret.pc counter/address to increment/set the pc to the next/right opcode
5. Opcode functions will continue, complete the operation, and iterate the program counter to the next address in RAM with opcode.

Testing, accuracy and effectiveness of model:

We built our tester as follows:

The first line of http://www.qmtpro.com/~nes/misc/nestest.log, reads,

C000 4C F5 C5 JMP $C5F5   A:00 X:00 Y:00 P:24 SP:FD PPU: 0, 21 CYC:7

Where the left half denotes the PC address, the opcode in bytes, and its name, and on the right, the correct respective registers, stack_pointer, and accumulator values that correspond to the opcodes in that line.

Hence, for testing purposes, our start_cpu() function works as follows:

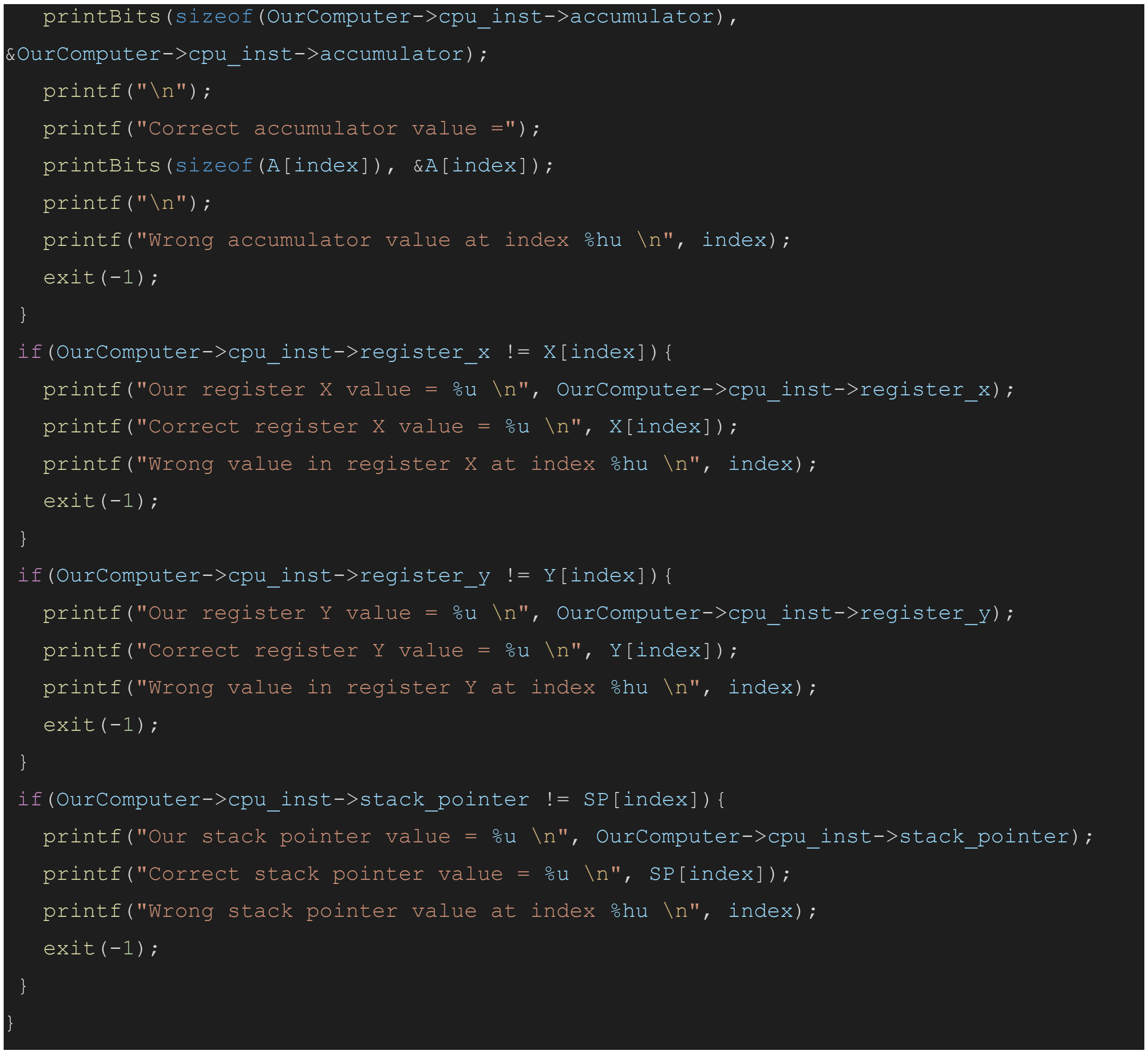
```
void start_cpu(){
 OurComputer->cpu_inst->pc = 0xC000; // starting address of the test opcodes
 OurComputer->cpu_inst->stack_pointer = 0xFD;
 for(address i = 0; i < 8991; i++){ // 8991 is the amount of test opcodes
   test_registers(i);
   execute(OurComputer->RAM[OurComputer->cpu_inst->pc], OurComputer->cpu_inst->pc);
 }
}
```

We initialize the program counter to start where the first instruction does in that line, and the stack pointer per the first line as well. (8991 is the amount of lines in the log test file).

Before each execute() invocation, we call test_regsiters with the current line block we are at per the for loops counter i, which tests whether our accumulator matches the one specified by A:00, our register X matches register X specified by X:00, and so on and so forth, for the stack pointer, and the program counter address as well. The details of how we parse the file to have a 1-1 mapping of each instruction to the correct values per call are specified in the Initialization and CPU execution, and can be found in the file "make_test_files/6502_test_opcodes_scraper.ipynb'.

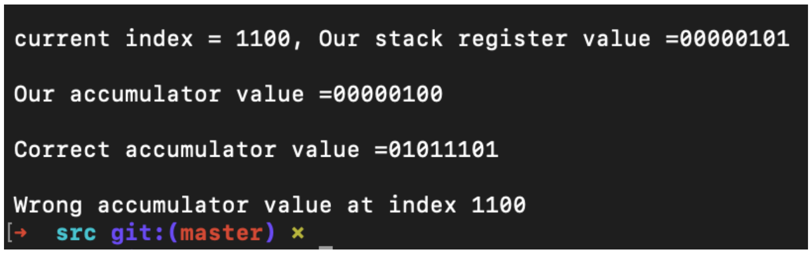
If at any point during the execution of this for loop, we get that one of the values have not matched, we throw an error, specifying the mismatch value and type, as shown below:

```
void test_registers(address index){
 if(OurComputer->cpu_inst->pc != PCs[index]){
     printf("Our PC = %x \n", OurComputer->cpu_inst->pc);
     printf("Correct PC = %x \n", PCs[index]);
     printf("Wrong pc address at index %hu \n", index);
     exit(-1);
   }
 if(OurComputer->cpu_inst->accumulator != A[index]){
   printf("Our accumulator value =");
```



```
   printBits(sizeof(OurComputer->cpu_inst->accumulator),
&OurComputer->cpu_inst->accumulator);
   printf("\n");
   printf("Correct accumulator value =");
   printBits(sizeof(A[index]), &A[index]);
   printf("\n");
   printf("Wrong accumulator value at index %hu \n", index);
   exit(-1);
 }
 if(OurComputer->cpu_inst->register_x != X[index]){
   printf("Our register X value = %u \n", OurComputer->cpu_inst->register_x);
   printf("Correct register X value = %u \n", X[index]);
   printf("Wrong value in register X at index %hu \n", index);
   exit(-1);
 }
 if(OurComputer->cpu_inst->register_y != Y[index]){
   printf("Our register Y value = %u \n", OurComputer->cpu_inst->register_y);
   printf("Correct register Y value = %u \n", Y[index]);
   printf("Wrong value in register Y at index %hu \n", index);
   exit(-1);
 }
 if(OurComputer->cpu_inst->stack_pointer != SP[index]){
   printf("Our stack pointer value = %u \n", OurComputer->cpu_inst->stack_pointer);
   printf("Correct stack pointer value = %u \n", SP[index]);
   printf("Wrong stack pointer value at index %hu \n", index);
   exit(-1);
 }
}
```

(PrintBits is just a custom function to print the value in question as a binary array). Having continually run the executable testprogram (6502 emulator) on the binary image made from these opcodes (test_opcodes.img), we were able to effectively debug faulty instruction name functions and decode for addressing mode faults in void_functions.c. Doing so, we were able to get up to a testcase number of 1100 consecutive instructions out of 8991 successfully.



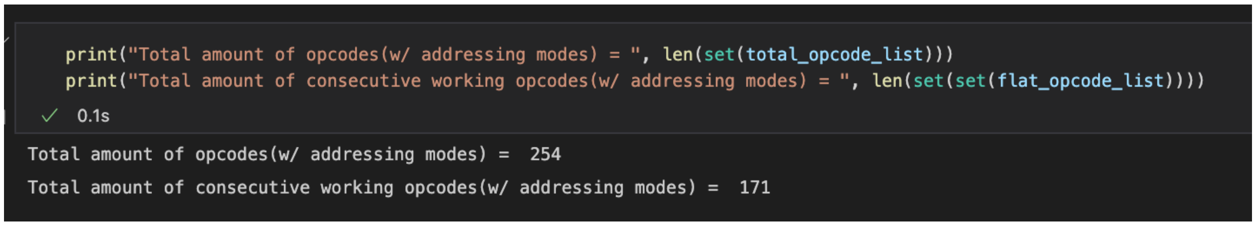
```
current index = 1100, Our stack register value =00000101

Our accumulator value =00000100

Correct accumulator value =01011101

Wrong accumulator value at index 1100
[→  src git:(master) ✗ 
```

Further, as found in "6502_test_opcodes_scraper.ipynb", this represents the use of a staggering 67% of the individual byte opcodes used consecutively over 1000 times.



```
print("Total amount of opcodes(w/ addressing modes) = ", len(set(total_opcode_list)))
print("Total amount of consecutive working opcodes(w/ addressing modes) = ", len(set(set(flat_opcode_list))))
```

```
✓ 0.1s
```

```
Total amount of opcodes(w/ addressing modes) =  254
Total amount of consecutive working opcodes(w/ addressing modes) =  171
```

What could have been improved:

We could have tried to attend more office hours and take advantage of pre-existing resources rather than trying to reinvent the wheel.

The rate at which we met as a team could have been more consistent. Although, we did meet a lot and work together almost every weekend, Sunday.

We could have been more proactive about integrating with web-assembly. We instead decided to first try to get the C executable running on all test cases, and then worry about translating to web-assembly; although, it may have proved better to tackle them simultaneously.

Ultimately, however, we set out to make consistent steady steps weekwise throughout the semester, and slowly but surely, arrived at a running simulator that works on nearly 70% of the different byte instructions, all without ever hardcoding each specific one.

Bibliography/Sources:

1. https://www.atariarchives.org/2bml/chapter_10.php
2. https://www.atariarchives.org/mlb/chapter6.php
3. https://jayscholar.etown.edu/cgi/viewcontent.cgi?article=1000&context=comscistu
4. http://users.telenet.be/kim1-6502/6502/proman.html#41
5. https://www.wdc65xx.com/wdc/documentation/w65c02s.pdf
6. https://piazza.com/class_profile/get_resource/ktajlid2ixe2nr/kubjlxqv8lu55d
7. https://piazza.com/class_profile/get_resource/ktajlid2ixe2nr/kubjlxfwb6b539
8. https://piazza.com/class_profile/get_resource/ktajlid2ixe2nr/kubjlx8j93a526
9. https://www.masswerk.at/6502/6502_instruction_set.html