

6502 Simulator

Collaborated on by:
Ben Badnani, William Wei, Ryan Velez, James Ngo

Recap from last presentation

- Introduced in 1975
- Well known consoles used either the 6502 or one of its variants
 - Atari 2600
 - Nintendo Entertainment System (NES)



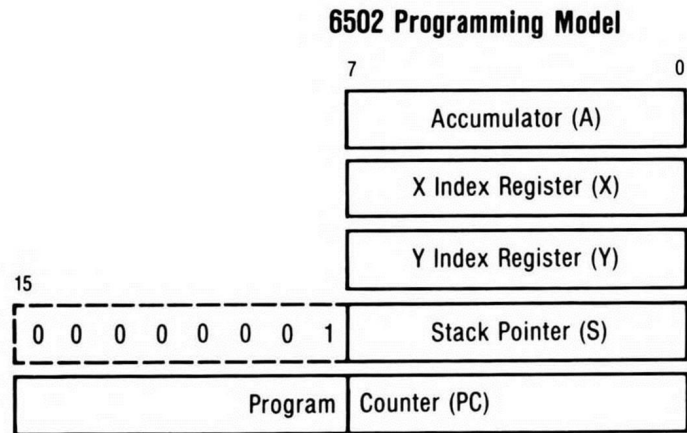
Used the 6507.



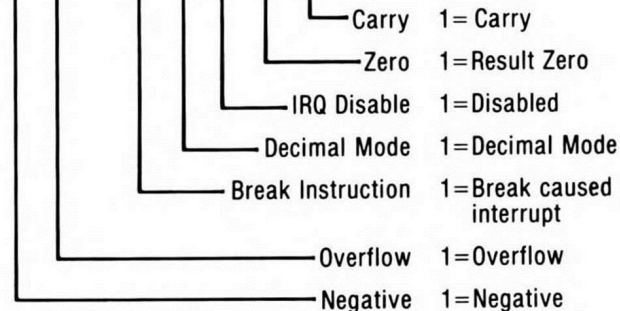
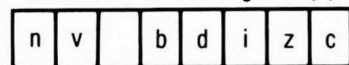
Used the 6502.

Hardware Recap

- Very few registers
 - One 8-bit accumulator
 - Two 8-bit index registers, X and Y
 - One 8 bit status register
 - One 8 bit stack pointer
 - One 16 bit program counter



Processor Status Register (P)



Simulator Setup

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

```
3390 2003F8C898AA940857820DF8C47820  
3391 E2F8E8A93F857820EEF8C47820F0F8E8  
3392 A9418578C47820FCF8E8A90085782006  
3393 F9C4782009F9E8A9808578C4782013F9  
3394 E8A9818578C478201DF9E8A97F8578C4  
3395 782027F9E88AA82090F985784678A578  
3396 209DF9C885784678A57820ADF9C820BD  
3397 F985780678A57820C3F9C885780678A5  
3398 7820D4F9C820E4F985786678A57820EA  
3399 F9C885786678A57820FBF9C8200AFA85  
3400 782678A5782010FAC885782678A57820  
3401 21FAA9FF85788501240138E678D00C30  
3402 0A50089006A578C900F0040000000A9  
3403 7F8578B818E678F00C100A7008B006A5  
3404 78C980F00400000000A9008578240138  
3405 C678F00C100A50089006A578C9FFF004  
3406 00000000A9808578B818C678F00C300A  
3407 7008B006A578C97FF0040000000A901
```

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

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

Simulator Setup

```
OurComputer->cpu_inst->stack_pointer = 0xFF;
```

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

```
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];  
}
```

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

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

```
3390 20D3F8C898AA940857820D0F8C47820  
3391 E2F8E8A93F857820EEF8C47820F0F8E8  
3392 A9418578C47820FCF8E8A90085782006  
3393 F9C4782009F9E8A9808578C4782013F9  
3394 E8A9818578C478201DF9E8A97F8578C4  
3395 782027F9E88AA82090F985784678A578  
3396 209DF9C885784678A57820ADF9C820BD  
3397 F985780678A57820C3F9C885780678A5  
3398 7820D4F9C820E4F985786678A57820EA  
3399 F9C885786678A57820FBF9C8200AFA85  
3400 782678A5782010FAC885782678A57820  
3401 21FAA9FF85788501240138E678D00C30  
3402 0A50089006A578C900F00400000000A9  
3403 7F8578B818E678F00C100A7008B006A5  
3404 78C980F00400000000A9008578240138  
3405 C678F00C100A50089006A578C9FFF004  
3406 00000000A9808578B818C678F00C300A  
3407 7008B006A578C97FF00400000000A901
```

Method Architecture Recap

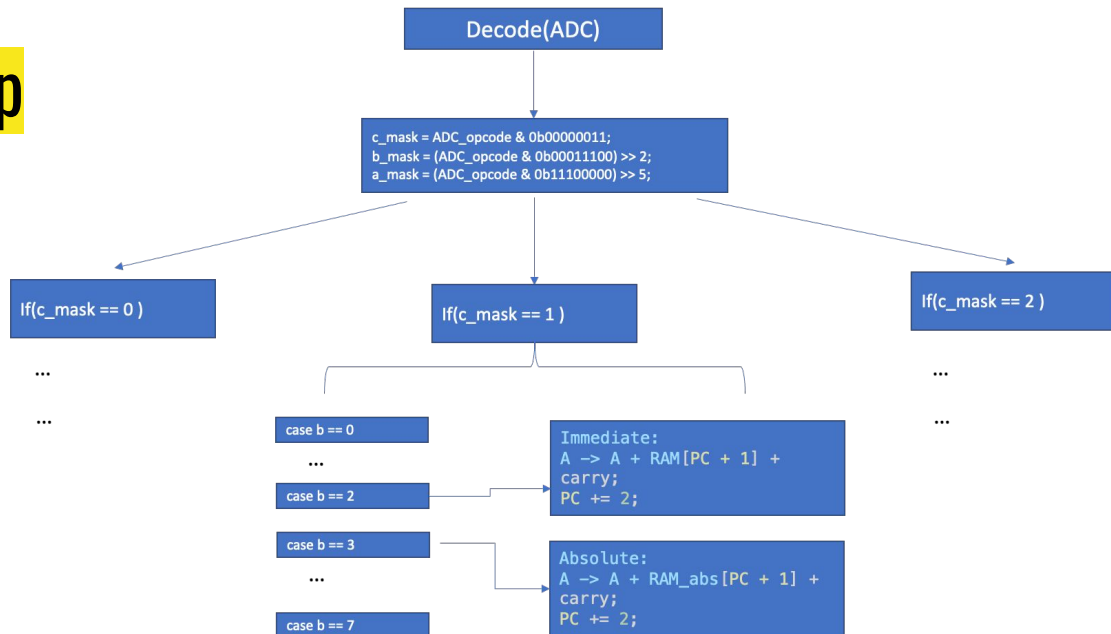
Fetch, Decode, Execute!

Fetch

4C F5 C5 A2 00 ..

Decode

Execute!!
Jmp \$C5 F5



Fetch using UTHash

- Build the hashtable to store function pointers

```
// Build opcode table
void build_opcode_table(){
    int n, fd;
    byte* opcodes_keys;
    OurComputer->opcodes = NULL;

    // need to read in the opcodes
    if((opcodes_keys = (byte*) malloc((opcode_size) * sizeof(byte))) == NULL){
        exit(-1);
    }
    if((fd=open("opcode_values", O_RDONLY)) < 0){
        exit(-1);
    }
    if((n = read(fd, opcodes_keys, opcode_size)) != opcode_size){
        exit(-1);
    }
    close(fd);

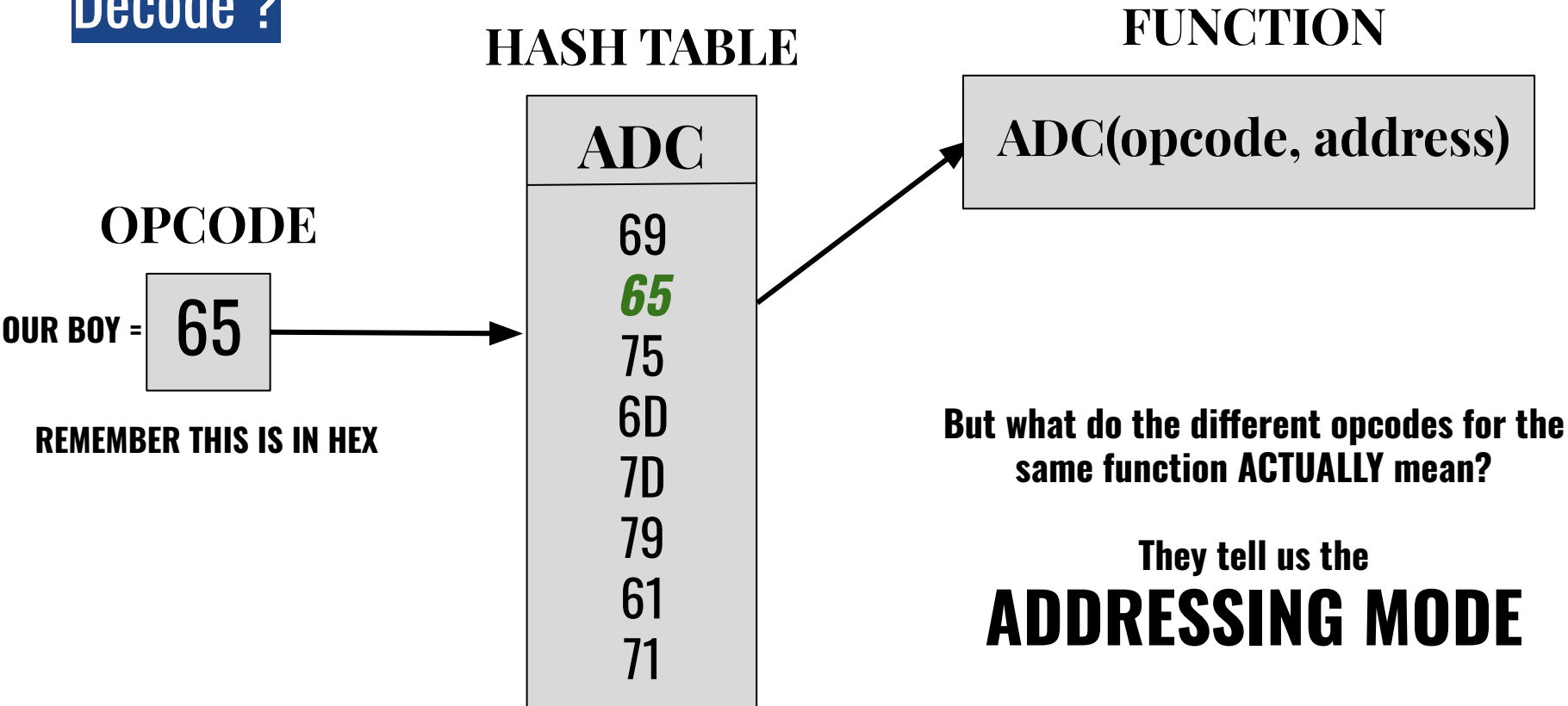
    struct opcode_table* s = NULL;
    for (int i = 0; i < opcode_size; i++){
        s = (struct opcode_table*) malloc(sizeof(*s)); // check if NULL?
        s->opcodes_key = opcodes_keys[i]; // initializing key for s
        s->opcode_function = functions[i]; // initializing the value for s
        HASH_ADD(hh,OurComputer->opcodes,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

UTHash Continued

- We can now invoke the correct function by reading in the opcode at the program counter

```
void find_user(byte opcode, address pc) {  
  
    struct opcode_table *s;  
    HASH_FIND_BYTE(OurComputer->opcodes, &opcode, s);  
  
    if(s == NULL){  
        printf("err");  
        return;  
    }  
  
    (*s->opcode_function) (opcode, pc);  
  
    return;  
}
```


Decode ?



Addressing Modes, a breakdown

Group 00

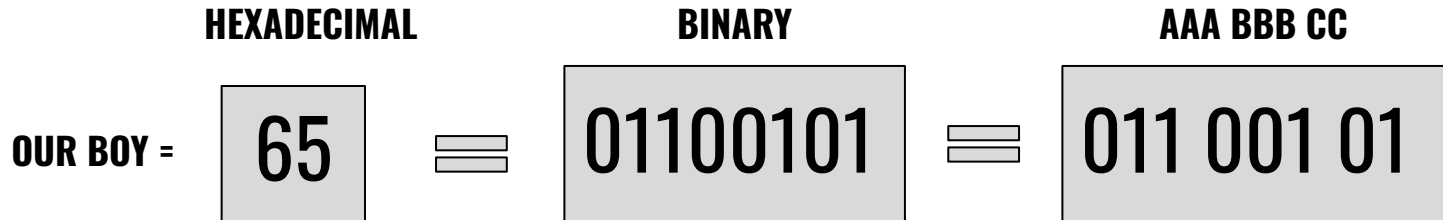
- 000 -> Immediate
- 001 -> Zero page
- 010 -> Absolute
- 101 -> Zero page, x
- 111 -> Absolute, x

Group 01

- 000 -> (Zero page, x)
- 001 - Zero page
- 010 -> Immediate
- 011 -> Absolute
- 100 -> (Zero page), Y
- 101 -> Absolute, Y
- 111 -> Absolute, X

Group 10

- 000 -> Immediate
- 001 -> Zero page
- 010 -> Accumulator
- 101 -> Zero page, X
- 111 -> Absolute, X



6502 Instructions in Detail

ADC Add Memory to Accumulator with Carry

$A + M + C \rightarrow A, C$

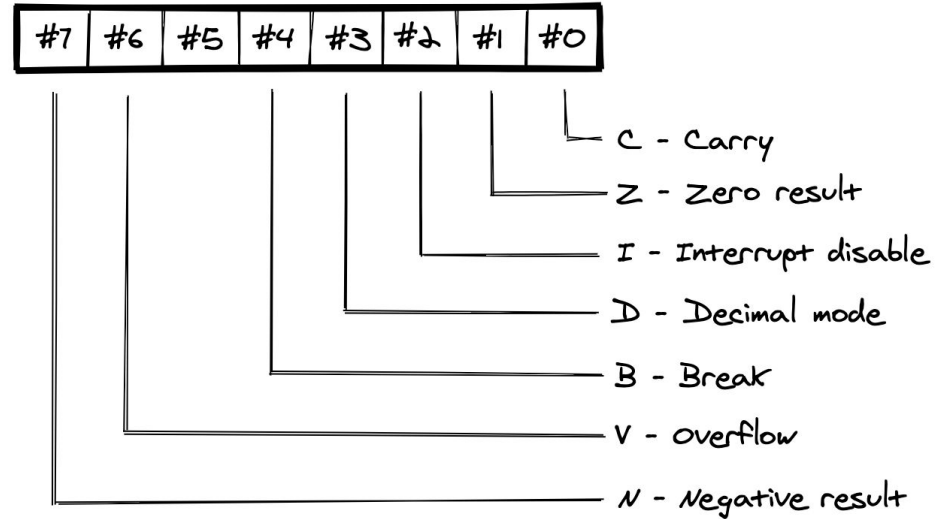
N Z C I D V
+ + + - - +

OUR BOY

addressing	assembler	opc	bytes	cycles
immediate	ADC #oper	69	2	2
zeropage	ADC oper	65	2	3
zeropage,X	ADC oper,X	75	2	4
absolute	ADC oper	6D	3	4
absolute,X	ADC oper,X	7D	3	4*
absolute,Y	ADC oper,Y	79	3	4*
(indirect,X)	ADC (oper,X)	61	2	6
(indirect),Y	ADC (oper),Y	71	2	5*

Status Register

- Many of the opcodes set different flags in the status register
- Functions can behave differently based on these flags



Functions setting flags

- Rotate operand one bit to the right
 - 10100010 -> [c]010001
- Updates the carry flag!
- But also, updates the negative flag on input carry
- Updates zero flag on operand == 0

```
void ROR(byte opcode, address pc) {
    decode(opcode);
    byte carry = getCarryFlag(); //get the carry flag
    byte bitZero = (0b1 & OurComputer->RAM[ret.pc]); // bit 0 is shifted into carry
    OurComputer->RAM[ret.pc] = OurComputer->RAM[ret.pc] >> 1; //shift ret.arg over one bit
    OurComputer->RAM[ret.pc] = carry | ret.arg; //move carry into the 7th bit
    //Set flags
    if (carry) {
        setNegativeFlag();
    }
    else {
        clearNegativeFlag();
    }
    if (bitZero) {
        setCarryFlag();
    }
    else {
        clearCarryFlag();
    }
    if (OurComputer->RAM[ret.pc] == 0) {
        setZeroFlag();
    }
    else {
        clearZeroFlag();
    }
    OurComputer->RAM[ret.pc] += ret.arg;
}
```

Branch Commands

```
// BNE - branch on Zero Flag = 0
void BNE (byte opcode, address pc) {
    if (!getZeroFlag()) {
        OurComputer->cpu_inst->pc += OurComputer->RAM[pc + 1];
    }
    else {
        OurComputer->cpu_inst->pc += 2;
    }
    return;
}
```

Conditional jump

More Branch Commands

```
// BEQ - branch on Zero Flag = 1
void BEQ(byte opcode, address pc) {
    if (getZeroFlag()) {
        OurComputer->cpu_inst->pc += OurComputer->RAM[pc + 1];
    }
    OurComputer->cpu_inst->pc += 2;
    return;
}
```

Stack and Stack Pointer Register

- Created at addresses 0x100 - 0x1FF
- Memory Map:
- 0x0000 - 0x00FF: Zero page
- 0x0100 - 0x01FF: Stack

```
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];
}
```

0x0000-
0x00FF

Zero - Page

0x0100-
0x01FF

Stack

0x0200 -

RAM

Running Instance of Simulator

→ `src git:(master) × ./testprogram test_opcodes.img`

```
int main(int argc, char* argv[]) {
    // user must pass in binary image to simulate RAM
    if (argc != 2){
        printf("%s outfile", argv[0]);
        return 1;
    }

    char* file_name = argv[1];
    // allocate memory for Computer Structure
    if((OurComputer = (struct Computer*) malloc(sizeof(struct Computer))) == NULL){
        exit(-1);
    }

    // initializing size of the RAM to 2^16
    if ((OurComputer->RAM = (byte*) malloc(RAMSIZE * sizeof(byte))) == NULL){
        exit(-1);
    }

    // initializing cpu structure inside of computer
    if((OurComputer->cpu_inst = (struct cpu*) malloc(sizeof(struct cpu))) == NULL){
        exit(-1);
    }

    read_in_binary_image(file_name); // fill struct->RAM with file_name
    build_opcode_table(); // link opcodes to functions in void_functions.c
    initialize_registers();
    start_cpu();

    free(OurComputer->RAM);
    free(OurComputer->cpu_inst);
    free(OurComputer);
    return 0;
}
```

```
void read_in_binary_image(char* image_name){
    int n, fd;
    if((fd=open(image_name, O_RDONLY)) < 0){
        exit(-1);
    }
    if((n = read(fd, OurComputer->RAM, RAMSIZE)) != RAMSIZE){
        exit(-1);
    }
    close(fd);
}
```

Running Instance of Simulator

→ `src git:(master) × ./testprogram test_opcodes.img`

```
int main(int argc, char* argv[]) {
    // user must pass in binary image to simulate RAM
    if (argc != 2){
        printf("%s outfile", argv[0]);
        return 1;
    }

    char* file_name = argv[1];
    // allocate memory for Computer Structure
    if((OurComputer = (struct Computer*) malloc(sizeof(struct Computer))) == NULL){
        exit(-1);
    }

    // initializing size of the RAM to 2^16
    if ((OurComputer->RAM = (byte*) malloc(RAMSIZE * sizeof(byte))) == NULL){
        exit(-1);
    }

    // initializing cpu structure inside of computer
    if((OurComputer->cpu_inst = (struct cpu*) malloc(sizeof(struct cpu))) == NULL){
        exit(-1);
    }

    read_in_binary_image(file_name); // fill struct->RAM with file_name
    build_opcode_table(); // link opcodes to functions in void_functions.c
    initialize_registers();
    start_cpu();

    free(OurComputer->RAM);
    free(OurComputer->cpu_inst);
    free(OurComputer);
    return 0;
}
```

```
// Builds opcode table
void build_opcode_table(){
    int n, fd;
    byte* opcodes_keys;
    OurComputer->opcodes = NULL;
    // need to read in the opcodes
    if((opcodes_keys = (byte*) malloc(opcode_size * sizeof(byte))) == NULL){
        exit(-1);
    }

    if((fd=open("opcode_values", O_RDONLY)) < 0){
        exit(-1);
    }

    if((n = read(fd, opcodes_keys, opcode_size)) != opcode_size){
        exit(-1);
    }

    close(fd);

    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);
    }

    return;
}
```

Running Instance of Simulator

→ `src git:(master) × ./testprogram test_opcodes.img`

```
int main(int argc, char* argv[]) {
    // user must pass in binary image to simulate RAM
    if (argc != 2){
        printf("%s outfile", argv[0]);
        return 1;
    }

    char* file_name = argv[1];
    // allocate memory for Computer Structure
    if((OurComputer = (struct Computer*) malloc(sizeof(struct Computer))) == NULL){
        exit(-1);
    }

    // initializing size of the RAM to 2^16
    if ((OurComputer->RAM = (byte*) malloc(RAMSIZE * sizeof(byte))) == NULL){
        exit(-1);
    }

    // initializing cpu structure inside of computer
    if((OurComputer->cpu_inst = (struct cpu*) malloc(sizeof(struct cpu))) == NULL){
        exit(-1);
    }

    read_in_binary_image(file_name); // fill struct->RAM with file_name
    build_opcode_table(); // link opcodes to functions in void_functions.c
    initialize_registers();
    start_cpu();

    free(OurComputer->RAM);
    free(OurComputer->cpu_inst);
    free(OurComputer);
    return 0;
}
```

```
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);
    }
}
```

Running Instance of Simulator

```
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);
    }
}
```

```
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);
    }
}
```

Running Instance of Simulator

```
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);
    }
}
```

```
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;
}
```

```
void ADC(byte opcode, address pc) {
    decode(opcode);
    // pull high bits to test for overflow later
    byte acc_hi = ((getAccumulator() & 0x80) >> 7);
    byte arg_hi = ((ret.arg & 0x80) >> 7);
    // perform addition, cull result to 2 bytes
    int16_t res = (int16_t) (getAccumulator() + ret.arg);

    printf("res = %d \n", res);
    printf("ret.arg = %d \n", ret.arg);
    printf("Accumulator = %u \n", getAccumulator());

    OurComputer->cpu_inst->accumulator = (byte) (res & 0x00ff);
    // add 1 if carry flag set
    if(getCarryFlag()){
        OurComputer->cpu_inst->accumulator += 1;
        res += 1;
    }

    if(res > 255){
        setCarryFlag();
    }else{
        clearCarryFlag();
    }

    // pull high bits of result to test for overflow
    byte res_hi = ((getAccumulator() & 0x80) >> 7);
    if(acc_hi == arg_hi && acc_hi != res_hi){
        setOverflowFlag();
    }else{
        clearOverflowFlag();
    }

    // test high bit of result to see if negative
    if(res_hi){
        setNegativeFlag();
    }else{
        clearNegativeFlag();
    }

    // if result is 0, set zero flag
    if(getAccumulator() == 0x00){
        setZeroFlag();
    }else{
        clearZeroFlag();
    }

    update_PC();
}
```

Running Instance of Simulator

```
void ADC(byte opcode, address pc) {
    decode(opcode);
    // pull high bits to test for overflow later
    byte acc_hi = ((getAccumulator() & 0x80) >> 7);
    byte arg_hi = ((ret.arg & 0x80) >> 7);
    // perform addition, cull result to 2 bytes
    int16_t res = (int16_t) (getAccumulator() + ret.arg);

    printf("res = %d \n", res);
    printf("ret.arg = %d \n", ret.arg);
    printf("Accumulator = %u \n", getAccumulator());

    OurComputer->cpu_inst->accumulator = (byte) (res & 0x00ff);
    // add 1 if carry flag set
    if(getCarryFlag()){
        OurComputer->cpu_inst->accumulator += 1;
        res += 1;
    }

    if(res > 255){
        setCarryFlag();
    }else{
        clearCarryFlag();
    }

    // pull high bits of result to test for overflow
    byte res_hi = ((getAccumulator() & 0x80) >> 7);
    if(acc_hi == arg_hi && acc_hi != res_hi){
        setOverflowFlag();
    }else{
        clearOverflowFlag();
    }

    // test high bit of result to see if negative
    if(res_hi){
        setNegativeFlag();
    }else{
        clearNegativeFlag();
    }

    // if result is 0, set zero flag
    if(getAccumulator() == 0x00){
        setZeroFlag();
    }else{
        clearZeroFlag();
    }

    update_PC();
}
```

```
void decode(byte opcode) {
    int a_mask, b_mask, c_mask;
    a_mask = (opcode & 0b11100000) >> 5;
    b_mask = (opcode & 0b00011100) >> 2;
    c_mask = opcode & 0b00000011;
    if (c_mask == 0b00) {
        switch(b_mask) {
            // immediate
            case 0:
                ret.arg = OurComputer->RAM[getProgramCounter() + 1];
                ret.pc = getProgramCounter() + 2;
                break;
            // zeropage
            case 1:
                ret.pc = OurComputer->RAM[getProgramCounter() + 1];
                if (a_mask == 0b100) { // STY
                    ret.arg = 2;
                    break;
                }
                ret.arg = OurComputer->RAM[ret.pc];
                ret.pc = getProgramCounter() + 2;
                break;
            // absolute
            case 3:
                ret.pc = read_16(getProgramCounter() + 1);
                if (a_mask == 0b010) { // JMP
                    break;
                }
                if (a_mask == 0b011) { // JMP (abs)
                    ret.pc = read_16(ret.pc);
                    break;
                }
                if (a_mask == 0b100) { // STY
                    ret.arg = 3;
                    break;
                }
                ret.arg = OurComputer->RAM[ret.pc];
                ret.pc = getProgramCounter() + 3;
                break;
            // zeropage, x
            case 5:
                ret.pc = OurComputer->RAM[getProgramCounter() + 1];
                ret.pc += getRegisterX();
                if (a_mask == 0b100) { // STY
                    ret.arg = 2;
                }
            }
        }
    }
}
```


Running Instance of Simulator

```
void ADC(byte opcode, address pc) {
    decode(opcode);
    // pull high bits to test for overflow later
    byte acc_hi = ((getAccumulator() & 0x80) >> 7);
    byte arg_hi = ((ret.arg & 0x80) >> 7);
    // perform addition, cull result to 2 bytes
    int16_t res = (int16_t) (getAccumulator() + ret.arg);

    printf("res = %d \n", res);
    printf("ret.arg = %d \n", ret.arg);
    printf("Accumulator = %u \n", getAccumulator());

    OurComputer->cpu_inst->accumulator = (byte) (res & 0x00ff);
    // add 1 if carry flag set
    if(getCarryFlag()){
        OurComputer->cpu_inst->accumulator += 1;
        res += 1;
    }

    if(res > 255){
        setCarryFlag();
    }else{
        clearCarryFlag();
    }

    // pull high bits of result to test for overflow
    byte res_hi = ((getAccumulator() & 0x80) >> 7);
    if(acc_hi == arg_hi && acc_hi != res_hi){
        setOverflowFlag();
    }else{
        clearOverflowFlag();
    }

    // test high bit of result to see if negative
    if(res_hi){
        setNegativeFlag();
    }else{
        clearNegativeFlag();
    }

    // if result is 0, set zero flag
    if(getAccumulator() == 0x00){
        setZeroFlag();
    }else{
        clearZeroFlag();
    }

    update_PC();
}
```

```
void decode(byte opcode) {
    int a_mask, b_mask, c_mask;
    a_mask = (opcode & 0b11100000) >> 5;
    b_mask = (opcode & 0b00011100) >> 2;
    c_mask = opcode & 0b00000011;
    if (c_mask == 0b00) {
        switch(b_mask) {
            // immediate
            case 0:
                ret.arg = OurComputer->RAM[getProgramCounter() + 1];
                ret.pc = getProgramCounter() + 2;
                break;
            // zeropage
            case 1:
                ret.pc = OurComputer->RAM[getProgramCounter() + 1];
                if (a_mask == 0b100) { // STY
                    ret.arg = 2;
                    break;
                }
                ret.arg = OurComputer->RAM[ret.pc];
                ret.pc = getProgramCounter() + 2;
                break;
            // absolute
            case 3:
                ret.pc = read_16(getProgramCounter() + 1);
                if (a_mask == 0b01) { // LD
                    break;
                }
                if (a_mask == 0b01) { // LD
                    ret.pc = read_16(getProgramCounter() + 1);
                    break;
                }
                if (a_mask == 0b10) { // LD
                    ret.arg = 3;
                    break;
                }
                ret.arg = OurComputer->RAM[ret.pc];
                ret.pc = getProgramCounter() + 2;
                break;
            // zeropage, x
            case 5:
                ret.pc = OurComputer->RAM[getProgramCounter() + 1];
                ret.pc += getRegisterX();
                if (a_mask == 0b100) { // STY
                    ret.arg = 2;
                }
            }
        }
    }
}
```

```
struct return_{
    address pc;
    byte arg;
};

struct return_ ret;
```

Running Instance of Simulator

```
void ADC(byte opcode, address pc) {
    decode(opcode);
    // pull high bits to test for overflow later
    byte acc_hi = ((getAccumulator() & 0x80) >> 7);
    byte arg_hi = ((ret.arg & 0x80) >> 7);
    // perform addition, cull result to 2 bytes
    int16_t res = (int16_t) (getAccumulator() + ret.arg);

    printf("res = %d \n", res);
    printf("ret.arg = %d \n", ret.arg);
    printf("Accumulator = %u \n", getAccumulator());

    OurComputer->cpu_inst->accumulator = (byte) (res & 0x00ff);
    // add 1 if carry flag set
    if(getCarryFlag()){
        OurComputer->cpu_inst->accumulator += 1;
        res += 1;
    }

    if(res > 255){
        setCarryFlag();
    }else{
        clearCarryFlag();
    }

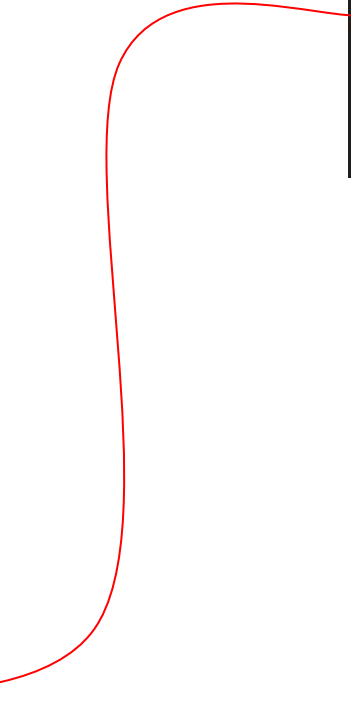
    // pull high bits of result to test for overflow
    byte res_hi = ((getAccumulator() & 0x80) >> 7);
    if(acc_hi == arg_hi && acc_hi != res_hi){
        setOverflowFlag();
    }else{
        clearOverflowFlag();
    }

    // test high bit of result to see if negative
    if(res_hi){
        setNegativeFlag();
    }else{
        clearNegativeFlag();
    }

    // if result is 0, set zero flag
    if(getAccumulator() == 0x00){
        setZeroFlag();
    }else{
        clearZeroFlag();
    }

    update_PC();
}
```

```
void update_PC(){
    OurComputer->cpu_inst->pc = ret.pc;
}
```



Running Instance of Simulator

```
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);
    }
}
```

Our accumulator value =00000100

Correct accumulator value =01011101

Wrong accumulator value at index 1100

→ [src git:\(master\)](#) ✖

```
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);
    }
}
```

Improvements

- Could've attended more office hours
- Could've had better time management and more consistent meetings
- Could've been more proactive about integrating the webassembly feature