1. **Introduction**

Chip-8 is an interpreted programming language. It was developed in the 1970’s and became popular since it helped programmers to easily program video games for other machines. The CHIP-8 Virtual Machine has 4KB (4096 bytes) of memory. The lower 512 bytes of memory i.e. from 0x000 to 0x200 were historically used by the interpreter itself. In our project, where the interpreter is running outside virtual machine’s memory space, we will use this area to store the fonts (from 0x00). The programs will be loaded at location 0x200. The machine has 16 8-bit data registers named V0 to VF out of which 15 (V0 - VE) can used as GPRs (General Purpose Registers) for arithmetic and logical operations.

The 16th register (VF) serves as an “overflow flag” and is SET when an addition results in an overflow. In the case of subtraction, the register VF is SET when there is no carry needed. The VF register is also SET when the draw instruction results in a pixel collision. There also exists an index register (I) which is used in the case of memory operations. CHIP-8 instructions or “opcodes” are 16-bits in length. A special register PC (Program Counter) keeps track of which instruction is to be executed next. Certain instructions like the JUMP instruction (JP addr) can change PC. The stack on the CHIP-8 virtual machine is used to store return addresses when subroutines are called. Our implementation is capable of storing upto 16 return addresses. We will maintain a stack pointer (SP) that will always point to the top of the stack. Such a stack pointer is helpful when we want to return from a subroutine [1, 2].

Our implementation has a “delay timer” which also existed in the original CHIP-8 implementation. When the delay timer is SET, it counts down to zero. CHIP-8 has a 16-key input device (0 – F). Our implementation will map this input configuration to the left side of the keyboard. CHIP-8 has 35 instructions or “opcodes”. There has been much debate on whether to employ the “classic switch-case instruction decoding” or “jump tables”. Whatever the case, It becomes important to classify the opcodes into separate groups. We can classify the opcodes into 5 groups according to their first byte :

* Unique – The first byte of these opcodes are unique. For example, 1nnn (JP addr) and 2nnn (CALL addr).
* Begin with ‘0’ - The first byte of these opcodes is ‘0’. For example, 00E0 (CLS) and 00EE (RET).
* Begin with ‘8’ - The first byte of these opcodes is ‘8’. For example, 8xy0 (LD Vx, Vy) and 8xy1 (OR Vx, Vy).
* Begin with ‘E’ - The first byte of these opcodes is ‘E’. For example, ExA1 (SKNP Vx) and Ex9E (SKP Vx).
* Begin with ‘F’ - The first byte of these opcodes is ‘F’. For example, Fx15 (LD DT, Vx) and Fx0A (LD Vx, K).

Our implementation decodes the instructions using function pointers (or jump tables). In contrast to the classic switch-case instruction decoding, this simplifies the design since now, we don’t have to deal with hundreds of “case” statements. CHIP-8 can display hexadecimal digits (0 – F) as sprites on the 64 x 32 screen. Each sprite is guaranteed to be 5 bytes long. The bits which are SET, draw out a pattern that corresponds to the particular character that is to be drawn.

Let us take an example of the character ‘4’

The binary representation for the character ‘4’ in CHIP-8 is,

10010000

10010000

11110000

00010000

00010000

The bits which are set (i.e. 1) draw out a pattern similar to that of character ‘4’.

The hexadecimal representation of this binary:

0x90

0x90

0xF0

0x10

0x10

This is the exact hexadecimal representation that is to be stored in our fonts array. Since each character is guaranteed to be 5 bytes long and we need to store 16 characters, our FONTS array has to of 80 bytes long.

|  |
| --- |
| uint8\_t FONTS[80] = {0xF0, 0x90, 0x90, 0x90, 0xF0, // 0  0x20, 0x60, 0x20, 0x20, 0x70, // 1  0xF0, 0x10, 0xF0, 0x80, 0xF0, // 2  0xF0, 0x10, 0xF0, 0x10, 0xF0, // 3  0x90, 0x90, 0xF0, 0x10, 0x10, // 4  0xF0, 0x80, 0xF0, 0x10, 0xF0, // 5  0xF0, 0x80, 0xF0, 0x90, 0xF0, // 6  0xF0, 0x10, 0x20, 0x40, 0x40, // 7  0xF0, 0x90, 0xF0, 0x90, 0xF0, // 8  0xF0, 0x90, 0xF0, 0x10, 0xF0, // 9  0xF0, 0x90, 0xF0, 0x90, 0x90 ,// A  0xE0, 0x90, 0xE0, 0x90, 0xE0, // B  0xF0, 0x80, 0x80, 0x80, 0xF0, // C  0xE0, 0x90, 0x90, 0x90, 0xE0, // D  0xF0, 0x80, 0xF0, 0x80, 0xF0, // E  0xF0, 0x80, 0xF0, 0x80, 0x80}; // F |

Fig. 1. Fonts Array Representation

**2. Related Work**

The technical reference for the instructions used in our project comes from the work of Thomas P. Greene.Thomas hosts a website at [1]. His technical reference to the CHIP-8 instruction set is named “Cowgod’s Chip-8 Technical Reference” and is probably the most popular technical reference for the CHIP-8 architecture. The reference has been immensely helpful to us. Although some regard the reference as to be somewhat faulty, it is to be noted that almost all of the available programs for the CHIP-8 architecture are programmed according to this reference.

The implementation idea for the Dxyn (DRW Vx, Vy, nibble) instruction comes from Laurence Muller. Laurence maintains a website at [2] . Laurence does an excellent job of explaining the concept of sprites and how the VF register can be used to detect collisions. Further, the website also presented a beautiful method of iterating over a byte bit-by-bit by using a bitwise AND operation. Our implementation also benefited immensely from the advice of not updating the screen after every cycle and instead maintaining a draw flag which makes sure that screen only gets updated after a draw or a clear screen instruction i.e. the only instructions that are capable of drawing on the screen. The idea of implementing jump tables instead of huge switch-case statements for decoding opcodes comes from Nigel Jones. The article we referred to was posted at the website [3] . Nigel first posted this article in the May 1999 issue of Embedded Systems Programming. In this article Nigel goes over the performance benefits from using jump tables over switch-case statements. The article mentions that while the compiler may optimise the switch-case statements into jump tables however, it is not always the case. It mentions that a programmer cannot reliably predict when a switch-case gets optimized into jump tables.

It also provides security methods to apply when implementing jump tables by using the keyword static in our project which would make sure that our table would not be changed by a rogue outsider. This article helped us a lot in implementing jump tables by means of function pointers in our project. It is to be noted that jump tables are preferred over switch-case statements in emulators/interpreters that are far more complex in implementation such as the NES, the SNES etc. The concept of jump tables proved to be an important lesson for us.

**3. Work Done**

In this project, we have implemented a CHIP-8 emulator in the C programming language along with all its instructions. As a result, all the games that are available for the CHIP-8 architecture can be run on our emulator.

**3.1 Proposed Work**

1. While implementing the draw instruction, we will wrap around any pixels that are drawn outside the screen. This is in contrast to other available emulators that do not wrap around the pixels.
2. In the implementation of the cxkk instruction, that generates a pseudo random number, some games such as VBRIX do not handle the number 32 correctly. Our implementation does not have this limitation.

**3.2 Instructions**

The emulator begins by calling the reset() function which resets all the registers (including memory) to zero, seeds the pseudo-random number generator and loads the fonts at memory location 0x00. It also sets the Program Counter (PC) to memory location 0x200 at which the to-be-executed program will be loaded. It then proceeds to load the ROM file by calling the loadROM() function. Then it proceeds to create a window via the SDL library. From then, it repeatedly calls the cycle() function until the ‘Escape’ key is pressed, in which case the program exits. The cycle() function is the heart of the emulator and is implemented in the classic “Fetch, Decode and Execute” way. It fetches 2 bytes from the emulator’s memory (of the program) and joins them together so as to form an “opcode”. Then it increments the PC by 2 since 2 bytes have been fetched from the memory. It masks the opcode with 0xF000 and logically shifts it right by 12-bits so as to get it’s first byte and consults a jump-table for deciding which of the 5 groups (as discussed earlier) it belongs to.

For example, let us consider the opcode “1208”:

Since, we masked 1208 with 0xF000 (and logically shifted it right by 12-bits), we are left with ‘1’. Now, this ‘1’ is looked up on the jump-table which decides that the instruction is of the format “1nnn” and calls the SYS\_1nnn() function to execute the opcode “1208”.

However, we can’t do this in case the opcode belongs to the other four groups. This is because, unlike “1208”, we cannot immediately decide the exact instruction just by looking at the first byte.

Let us take an example, the opcode “80E0”. Looking at it’s first byte, we can tell that it belongs to the group where opcodes begin with “8”. However, we cannot tell whether the instruction belongs to 8xy0, 8xy1 .... or 8xy7 just by looking at it’s first byte. In this case we also need to look at it’s last byte.

So, our procedure for decoding an opcode becomes as follows:

1. Get the first byte (first\_byte) of the opcode and call JUMP\_TABLE[first\_byte]

Our jump table is implemented somewhat like this:

JUMP\_TABLE[first\_byte] =

{ JUMP\_TABLE\_0, SYS\_1nnn, SYS\_2nnn, SYS\_3xkk,

SYS\_4xkk, SYS\_5xy0, SYS\_6xkk, SYS\_7xkk,

JUMP\_TABLE\_8, SYS\_9xy0, SYS\_Annn, SYS\_Bnnn,

SYS\_Cxkk, SYS\_Dxyn, JUMP\_TABLE\_E, JUMP\_TABLE\_F

};

2. If the first\_byte is ‘unique’ (i.e. 1, 2, 3, 4, 5, 6, 7, 9, A, B, C, D) then JUMP\_TABLE will call the respective SYS\_xxxx() function.

3. If the first\_byte is ‘0’ then JUMP\_TABLE will call JUMP\_TABLE\_0() function which will further lookup the opcode on a jump table which will also look at it’s last byte to decode it.

4. If the first\_byte is ‘8’ then JUMP\_TABLE will call JUMP\_TABLE\_8() function which will further lookup the opcode on a jump table which will also look at it’s last byte to decode it.

5. If the first\_byte is ‘E’ then JUMP\_TABLE will call JUMP\_TABLE\_E() function which will further lookup the opcode on a jump table which will also look at it’s last byte to decode it.

6. If the first\_byte is ‘F’ then JUMP\_TABLE will call JUMP\_TABLE\_F() function which will further lookup the opcode on a jump table which will look at it’s last two bytes to decode it.

The reason for JUMP\_TABLE\_F() to look at the opcode’s last two bytes instead of just one is because unlike others (‘0’, ‘8’, ‘E’ grouped) the ‘F’ group has two instructions Fx55 and Fx65 that both have their last byte as ‘5’. Obviously, we can’t just take the last byte for deciding the exact instruction in this case. Therefore, we consider the last two bytes.

For ‘E’ and ‘F’ grouped instructions, we have a large amount of indexes in our jump-table that are unfilled. The functions at these indexes, if called, serve to inform us that an invalid function is being called. So, we decided to have a special function – SYS\_INVAL() that will print out “Invalid opcode” and show which opcode failed. It’ll then exit.

The input handling is done via the SDL2 library which provides a cross-platform graphic library. CHIP-8’s inputs and it’s keyboard mapping are as follows:

KEYPAD KEYBOARD

1 2 3 C => 1 2 3 4

4 5 6 D => Q W E R

7 8 9 E => A S D F

A 0 B F => Z X C V

Apart from these keys, the ‘Esc’ (Escape) key can be used at any time to exit the emulator.

**3.2. Instructions**

1. SYS\_0nnn() - SYS addr

It is ignored by our emulator. This instruction was originally used to call machine language subroutines.

2. SYS\_00E0() - CLS

This instruction clears the entire 64 x 32 display via a memset() call.

memset(VIDEO, 0, sizeof(VIDEO));

3. SYS\_00EE() - RET

This instruction is used to return from subroutines. This is generally used at the end of a subroutine so as to transfer the control back to the position at which the subroutine was called at.

The program counter is set to the address stored at the top of the stack and 1 is subtracted from the stack pointer.

PC = STACK[SP];

SP--;

4. SYS\_1nnn() - JP addr

This instruction jumps to address nnn. This is similar to an unconditional jump. This is usually used to repeat parts of the program one or more times. To get the address, the opcode is masked with 0x0FFF. Then, the program counter is set to that address.

uint16\_t address = opcode & 0x0FFFu;

PC = address;

5. SYS\_2nnn() - Call addr

This instruction is used to call subroutines which are used to execute parts of the program multiple times.

On the COSMAC VIP, since the stack could store upto 12 addresses, 12 subsequent subroutines could be called. In our implementation, 16 subroutines can be subsequently called.

First, the stack pointer is incremented. Then the current value of the program counter (the point at which you’re supposed to return after the subroutine ends) is stored onto the top of the stack. To get the address, the opcode is masked with 0x0FFF and then the program counter is set to the address.

SP++;

STACK[SP] = PC;

uint16\_t nnn = (OPCODE & 0x0FFFu);

PC = nnn;

6. SYS\_3xkk() - SE Vx, byte

This instruction skips the next instruction (by incrementing the program counter by 2) if the value of register x is equal to kk (a byte). This is an example of conditional branching, however, it is to be noted that only a single instruction can be skipped.

To get the x register, the opcode is masked with 0xF00 and shifted right by 8 bits so as to convert the resultant to a single byte. Then, to get kk (a byte), the opcode is masked with 0x00FF. Finally, we check if the value at register x is equal to that of kk (a byte) and if it is, then skip the next instruction.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t kk = (OPCODE & 0x00FFu);

if(V[x] == kk)

PC = PC + 2;

7. SYS\_4xkk() - SNE Vx, byte

This instruction skips the next instruction (by incrementing the program counter by 2) if the value of register x is not equal to kk (a byte). This is also an example of conditional branching but only a single instruction can be skipped.

To get the x register, the opcode is masked with 0xF00 and shifted right by 8 bits so as to convert the resultant to a single byte. Then, to get kk (a byte), the opcode is masked with 0x00FF. Finally, we check if the value at register x is not equal to that of kk (a byte) and if it is not, then skip the next instruction.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t kk = (OPCODE & 0x00FFu);

if(V[x] != kk)

PC = PC + 2;

8. SYS\_5xy0() - SE Vx, Vy

This instruction skips the next instruction (by incrementing the program counter by 2) if the value of register x is equal to the value of register y. This is also an example of conditional branching but only a single instruction can be skipped.

To get the first register (x) to be compared, we mask the opcode with 0x0F00 and bitshift it right by 8 bits to convert the resultant into a byte. We mask the opcode with 0x00F0 and bitshift it right by 4 bits to get the second register (y) that is to be compared (x) and convert the resultant to a byte. Then, we compare the values at the two registers (x and y) and if they are equal, we skip the next instruction (by incrementing the program counter by 2).

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

if(V[x] == V[y])

PC = PC + 2;

9. SYS\_6xkk() - LD Vx, byte

This instruction stores the value of kk (a byte) into the register x.

To get the register x, we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. Then, we get the value of kk (a byte) that is to be stored by masking the opcode with 0x00FF. We then store the value kk in the register x.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t kk = (OPCODE & 0x00FFu);

V[x] = kk;

10. SYS\_7xkk() - ADD Vx, byte

This instruction adds kk (a byte) to the already existing byte in the register x and stores it in the register x.

To get the register x, we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a single byte. To get kk (a byte) we mask the opcode with 0x00FF. Then, we add kk (a byte) to the value in the register x and store the result in the register x.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t kk = (OPCODE & 0x00FFu);

V[x] = V[x] + kk;

11. SYS\_8xy0() - LD Vx, Vy

This instruction stores the value in the register y into the register x.

To get the register x, we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant to a byte. To get the register y, we mask the opcode with 0x00F0 and bitshift it right by 4 bits so as to convert it to a byte. Then we store the value of the register y into the register x.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

V[x] = V[y];

12. SYS\_8xy1 – OR Vx, Vy

This instruction performs a bitwise OR operation on the contents of the x register and the y register. It then stores the resultant in the x register.

To get the first register (x) we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. To get the y register we mask the opcode with 0x00F0 and bitshift it right by 4 bits so as to convert the resultant into a byte. Then we perform a bitwise OR operation on the contents of these two registers and store them in the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

V[x] = (V[x] | V[y]);

13. SYS\_8xy2 – AND Vx, Vy

This instruction performs a bitwise AND operation on the contents of the x register and the y register. It then stores the resultant in the x register.

To get the first register (x) we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. To get the y register we mask the opcode with 0x00F0 and bitshift it right by 4 bits soa as to convert the resultant into a byte. Then we perform a bitwise AND operation on the contents of these two registers and store the result in the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

V[x] = (V[x] & V[y]);

14. SYS\_8xy3() - XOR Vx, Vy

This instruction performs a bitwise exlusive OR (XOR) operation on the contents of the x register and the y register. It then stores the resultant in the x register.

To get the first register (x) we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. To get the ye register we mask the opcode with 0x00F0 and bitshift it right by 4 bits so as to convert the resultant into a byte. Then we perform a bitwise exclusive on the contents of these two registers and store the result in the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

V[x] = (V[x] ^ V[y]);

15. SYS\_8xy4() - ADD Vx, Vy

This instrcution adds the content of the x register with the content of the y register and stores them in the register x. If the additive result is greater than 8 bits (i.e. 255) then the carry flag (the VF register) is SET otherwise it is CLEARED. Only the lowest 8 bits of the additive result is stored in the x register.

To get the x register we mask the opcode with 0x0F00u and bitshift it right by 8 bits to convert the result into a byte. To get the y register we mask the opcode with 0x00F0 and bitshift it right by 4 bits to convert the result into a byte. We check if the additive result of the contents of the x register and the y register is greater than 8 bits (i.e. 255). The carry flag (VF register) is SET if this condition is true and it is CLEARED if this condition is false. We then store the additive result in the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

if( (V[x] + V[y]) > 255u)

V[0xF] = 1;

else

V[0xF] = 0;

V[x] = (V[x] + V[y]);

16. SYS\_8xy5() - SUB Vx, Vy

This instruction checks if the content of the x register is greater than the content of the y register. If the condition is true then the carry flag (VF register) is SET otherwise it is CLEARED.

To get the x register we mask the opcode with 0x0F00 and bitshift it by 8 bits so as to convert the resultant into a byte. To get the y register we mask the opcode with 0x00F0 and bitshift it right by 4 bits so as to convert the resultant into a byte. We then check if the content of the x register is greater than the y register. If the condition is true, we SET the carry flag (VF register) otherwise it is CLEARED. We then subtract the content of they y register from the content of the x register (i.e. V[x] – V[y]) and store the result in the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

if(V[x] > V[y])

V[0xF] = 1;

else

V[0xF] = 0;

V[x] = (V[x] – V[y]);

17. SYS\_8xy6() - SHR Vx {, 2Vy}

This instruction checks if the least significant bit of the content of the x register is 1. If the condition is true, VF register is SET otherwise it is cleared.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant to a single byte. We then perform a bitwise AND operation on the content of the x register and the number 0x1 and store the result in the VF register (this checks if the least signicifant bit is 1 or not). We then bitshift the content of the x register by 1 bit so as to divide it by 2.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

V[0xF] = (V[x] & 0x1u);

V[x] = (V[x] >> 1);

18. SYS\_8xy7 – SUBN Vx, Vy

This instruction subtracts the content of the register x from that of y and checks if the value in register y is greater than that of x. If this condition is true, then the VF register is SET otherwise it is CLEARED.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits. To get the y register we mask the opcode with 0x00F0 and bitshift it right by 4 bits. We then check if the content of the y register is greater than that of the x register. If the condition is true we SET the VF register and CLEAR the VF register if it is not. We then subtract the content of the y register from the content of the x register and store the resultant in the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

if(V[y] > V[x])

V[0xF] = 1;

else

V[0xF] = 0;

V[x] = (V[y] – V[x]);

19. 8xyE – SHL Vx, {, Vy

This instruction checks if the most significant bit of the content of the x register is 1. If the condition is true then the VF register is SET otherwise it is CLEARED. Then, the content of the x register is multiplied by 2. To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert it into a single byte. We then bitshift the content of the x register by 7 bits right (to get the most significant bit) and store it in the VF register. This makes sure that the VF register is SET if the most significant bit is 1 (i.e. the value is greater than 128 since it is the only way for the MSB to be 1 for a 8 bit value) and is CLEARED otherwise. We then bitshift the content of the x register left by 1 bit so as to multiply it by 2.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

V[0xF] = V[x] >> 7u;

V[x] = (V[x] << 1);

20. 9xy0 – SNE Vx, Vy

This instruction checks if the content of the x register is not equal to that of the y register. If the condition is true then the program counter is incremented by 2 (effectively skipping the next instruction).

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert it into a single byte. To get the y register we mask the opcode with 0x00F0 and bitshift it right by 4 bits so as to convert it into a single byte. We then check if the content of the x register is not equal to the content of the y register. If the condition is true then the program counter is incremented by 2.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

if(V[x] != V[y])

PC = PC+2;

21. Annn – LD I, addr

This instruction sets the value of the register I to nnn (an address).

To get the address we mask the opcode with 0x0FFF and store it in the variable named nnn. We then set the I register to the variable nnn.

uint16\_t nnn = (OPCODE & 0x0FFFu);

I = nnn;

22. Bnnn – JP V0, addr

This instruction sets the program counter to V0 + addr (i.e. the content of the 0th register + the address nnn) effectively jumping to that address. This is also an example of an unconditional jump.

To get the address (that is to be added to the content of the 0th register) we mask the opcode with 0x0FFF and store it in the variable named nnn. We then set the program counter to the additive result of that of the content of the 0th register and the address nnn.

uint16\_t nnn = (OPCODE & 0x0FFFu);

PC = (V[0] + nnn);

23. Cxkk – RND Vx, byte

This instruction performs a bitwise AND operation between the contents of the x register and kk (a byte). It then stores the result in the x register.

To find out the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits. To find kk (the byte) we mask the opcode with 0x00FF. We then perform a bitwise and between a random number (that is generated by our PRNG – pseudo random number generator) and kk (the byte). The result of this operation is stored in the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t kk = (OPCODE & 0x00FFu);

uint8\_t prng\_b = rand() & kk;

V[x] = prng\_b;

24. Dxyn – DRW Vx, Vy, nibble

This instruction displays (on the 64 x 32 screen) a sprite that starts at the memory location pointed to by the I register. The sprite is displayed at the position pointed by the x and the y register. Sprites are XORed onto the screen and if this causes any of the previous sprites on the screen to be turned off then the VF register is SET.

This is the only instruction that’s capable on drawing on the screen and hence, the most important instruction of this whole project.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. To get the y register we mask the opcode with 0x00F0 and bitshift it right by 4 bits so as to convert the resultant into a byte.

The value ‘n’ represents the height of the sprite. To get n we mask the opcode with 0x000F.

We then declare two variables xc and yc. The content of the x register is stored in xc and the content of the y register is stored in yc. This pair xc, yc denotes the position at which the sprite is to be drawn.

In our first loop (named outside\_loop), we iterate from ‘0’ to ‘n’ which is the number of rows to be drawn. With each iteration, we load the sprite byte from memory[ I + outside\_loop\_counter] into the variable sprite\_byte.

For each such row, we also need to iterate over the byte, bit by bit. This becomes our second loop (named inside\_loop). Since, a sprite is 8 bits long we need 8 iterations to iterate over an entire row. So, we iterate from ‘0’ to ‘8’ for each bit and check if the bit is SET (i.e. 1). If it is SET then we need to draw a pixel at the position yyPos=(yc + outside\_loop\_counter) and xxPos=(xc + inside\_loop\_counter). yyPos is named so because it represents the y-cordinate at which the pixel of the sprite is going to get drawn. The same goes for xxPos.

Since, pixels are XORed onto the screen, we must check for any existing pixel at the point where we are about to draw (because for an XOR operation, the result is 0 if both the operands are 1). If there is such an existing pixel, then we SET the VF register. Here, the VF register acts as a collision detector. Finally, we draw at the position (xxPos, yyPos).

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t y = (OPCODE & 0x00F0u) >> 4u;

uint8\_t n = (OPCODE & 0x000Fu);

uint8\_t xc = V[x];

uint8\_t yc = V[y];

V[0xF] = 0;

for(uint8\_t i=0 ; i<n ; i++)

{

uint8\_t sprite\_byte = MEMORY[I + i];

for(uint8\_t col=0 ; col<8 ; col++)

{

uint8\_t sprite\_bit = sprite\_byte & (0x80u >> col);

uint8\_t yyPos = yc + i;

uint8\_t xxPos = xc + col;

xxPos = xxPos%64;

yyPos = yyPos%32;

uint32\_t \*spp = &VIDEO[(yyPos) \* 64 + (xxPos)];

if(sprite\_bit)

{

if(\*spp == 0xFFFFFFFF)

V[0xF] = 1;

\*spp ^= 0xFFFFFFFF;

}

}

}

Notice how be iterate over an entire byte, bit-by-bit:

for(uint8\_t col=0 ; col<8 ; col++)

{

uint8\_t sprite\_bit = sprite\_byte & (0x80u >> col);

Let’s say our sprite\_byte is 01011010

When col = 0, sprite\_bit = 01011010 & (0x80u >> 0)

i.e.

0 1 0 1 1 0 1 0

& 1 0 0 0 0 0 0 0 (0x80u >> 0 = 10000000 >> 0 = 10000000)

--------------------

0 0 0 0 0 0 0 0

--------------------

Similary,

When col = 4, sprite\_bit = 01011010 & (0x80 >> 4)

i.e.

0 1 0 1 1 0 1 0

0 0 0 0 1 0 0 0 (0x80u >> 4 = 10000000 >> 4 = 00001000)

& --------------------

0 0 0 0 1 0 0 0

--------------------

25. Ex9E – SKP Vx

This instruction increments the program country by 2 if the key with the value represented by the content of the x register is pressed.

To get the x register we bitmask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. Then we check if the key with the value of the x register is pressed. If the condition is true then the program counter is incremented by 2 (effectively skipping the next instruction).

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t k = V[x];

if(KEYPAD[k])

PC = PC + 2;

26. ExA1 – SKNP Vx

This instruction increments the program country by 2 if the key with the value represented by the content of the x register is not pressed.

To get the x register we bitmask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. Then we check if the key with the value of the x register is not pressed. If the condition is true then the program counter is incremented by 2 (effectively skipping the next instruction).

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t k = V[x];

if(!KEYPAD[k])

PC = PC + 2;

27. Fx07 – LD Vx, DT

This instruction loads the the x register with the value of the delay timer.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. Then we place the value of the delay timer register into the register x.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

V[x] = DT;

28. Fx0A – LD Vx, K

This instruction waits for a key press. Until a key is pressed, all execution stops.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. We then check if any of the 16 keys is pressed. If the condition is true then the normal execution continues. If this condition is false then we decrement the program counter by 2. This effectively means that until a key is pressed our program will not continue forward.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

bool flag = false;

for(int k=0 ; k<16; k++)

if(KEYPAD[k])

{

V[x] = k;

flag = true;

break;

}

if(flag == false)

PC = PC – 2;

29. Fx15 – LD DT, Vx

This instruction loads the delay timer register with value present in the x register.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. We then get the value present in the x register and load it into the delay timer register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

DT = V[x];

30. Fx18 – LD ST, Vx

This instruction loads the sound timer register with the value present in the x register.

This instruction is ignored by our interpreter.

31. Fx1E – ADD I, Vx

This instruction adds the value of the I register and the value of the x register. It then places the result into the I register.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. We then add the value in the I register and the value present in the x register. This additive result is placed in the I register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

I = I + V[x]

32. Fx29 – LD F, Vx

This instruction places the memory address of the sprite of the digit (present in the x register) into the I register.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. We then find the digit (0x0 – 0x16 i.e. 0 – F) of the sprite whose position is to be loaded in the I register. We then multiply the digit by 5 (to get the offset at which the said digit resides in memory) and place this result in the I register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t digit = V[x];

I = (5 \* digit);

33. Fx33 – LD B, Vx

This instruction stores the BCD (Binary Coded Decimal) value of the content of the x register into the memory locations I, I + 1 and I + 2.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a single byte. Since an unsigned 8-bit number can only go upto 255, we find out the last digit and place it into MEMORY[I + 2]. Then we find out the second-last digit and place it into MEMORY[I + 1]. And finally we find out the third-last digit i.e. the first-digit and place it into MEMORY[I]. This is done by a series of repeated divisions and modulo operations.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

uint8\_t number = V[x];

MEMORY[I + 2] = number % 10;

number = number / 10;

MEMORY[I + 1] = number % 10;

number = number / 10;

MEMORY[I + 0] = number % 10;

34. Fx55 – LD [I], Vx

This instruction places the contents of the the registers beginning from the 0 register to the x register into the memory location starting at the value of the I register.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. We then place the contents of the registers beginning from the 0 register to the x register starting at memory location I.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

for(uint8\_t r=0 ; r<=x ; r++)

MEMORY[I + r] = V[r]

35. Fx65 – LD Vx, [I]

This instruction places the contents of the memory location starting at the value of the I register into the registers beginning from the 0 register to the x register.

To get the x register we mask the opcode with 0x0F00 and bitshift it right by 8 bits so as to convert the resultant into a byte. We then place the contents of the memory location starting at the value of the I register into the registers beginning from the 0 register to the x register.

uint8\_t x = (OPCODE & 0x0F00u) >> 8u;

for(uint8\_t r=0 ; r<=x ; r++)

V[r] = MEMORY[I + r]

**4. Conclusion**

Our primary motivation behind choosing to build a basic emulator was to understand the fundamentals of how CPUs worked. We were always fascinated by microprocessors especially the older ones such as the 6502, Z80, 8086, m68k etc. These were very successful CPUs which were used in many video game consoles. For example, the 6502 was used on the NES and the Z80 on the Pacman Arcade Machine. Also, the legendary personal computer, the Commodore Amiga was powered by one of these processors as well – the m68k. This project gave us an insight into writing emulators and we hope the knowledge gained from this project will help us writing more complex emulators in the future.

We would like to thank our supervisor, Dr.Sushree B. B. Priyadarshini, for her guidance which helped us immensely in our project. We would also like to thank our professor, Dr.Smita Prava Mishra, for helping us as well.

**References**

1. Greene, Thomas, “Cowgod’s CHIP-8 Technical Reference” <http://devernay.free.fr/hacks/chip8/C8TECH10.HTM>

2. Muller, Laurence, “How to write an emulator (CHIP-8 emulator)“ <http://www.multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/>

3. Jones, Nigel, "Arrays of Pointers to Functions" Embedded Systems Programming, May 1999.

**Motivation**

As architectures get old, the programs written using CHIP-8 architecture get forgotten. Since, CHIP-8 was developed in the 1970’s, almost everyone has no access to the original machines on which the programs ran. W