A simple emulation project: Chip 8

First of all I would just like to say that this guide was originaly by the author of
http://www.goldroad.co.uk
but as this site is becoming out of date and with broken links appearing everywhere
I have decided to no longer link to the site. Instead I (TJA) have copied parts of the guide
and re-written it fixing the mistakes I have found and also I have added parts, to make
it more complete.

What is chip 8?

Chip 8 was never a system as such, instead it was a virtual machine in the 1970s and resurrected in the 1990s on those powerful graphic calculators. Games could be written easily in the Chip 8 langauge, and then executed on any computer than had a Chip 8 interpreter. Therefore, it was kind of like a primitive Java.

The games

Chip 8 games are simple but there are many interpretations of classic games; pong, joust, breakout, space invaders and so on. There are also some more modern games such as tetris, and that game everyone has on their Nokia, with the worm chasing its tail.

How can Chip 8 help me learn to emulate?

Writing a Chip 8 emulator is one of the simplest emulation projects you could undertake. The instruction set is very small (about 30 instructions) and include many that you will find when you emulate CPUs for system emulators, such as:

- load & store
- arithmetic shift
- bitwise instructions
- jumps and subroutines

When you have written a chip 8 emulator, you could upgrage it to a SCHIP system, which has an extended instruction set and introduces more CPU architecure, such as a stack

A word of caution

Despite what i have said, there are some things to bear in mind. When you write a Chip 8 emulator, you are really writing an interpreter. The Chip 8 langauge has, for instance, an instruction that will draw a spite to the screen, and another which will point an addressing register to a built in font. These kind of commands you will not find in any other emulation (I think) but you can safely think of them as if they were just functions provided by a systems bios or graphics chips which is more realistic. There are some also slightly strange features of writing a Chip 8 emulator. For instance, the basic Chip 8 instruction set includes calls and returns, but there is no stack. This means you have to implement a stack as part of the interpreter to store return addresses. Well, its strange, but all good practice.

Some Chip 8 Roms (freeware)

Coming soon......

Getting Started

Typically starting to write an emulator, this is the sort of information you should seek to find:

- how much memory does the system address? - Chip 8 address 4k, 0x000 to 0x200 are reserved for the interpreter, so will be empty in an emulator
- the CPU registers - Chip 8 has 15 8-bit general purpose registers with equal status, named V0,V1...VE. How about making emulation simpler by using an array such as char V[16] to emulate these?
- The 16th register VF is equal to the carry flag in other systems
- There is also a memory address register, I, and a program counter. Both can be emulated as 16-bit, though they only 0 - 0xFFF.
- The systems memory map; It is extremely simple for Chip 8:
    - 0xF?? - 0xFFF built in 4x5 pixel font set, A-F, 1-9.
    - 0x200 - 0xF?? Program Rom and work RAM
    - 0x000 - 0x200 Chip 8 interpreter (see note above)
- The graphics system. Chip 8 has 1 instruction which draws sprites to the screen Drawing is done is XOR mode and if a pixel is turned off as a result of drawing, the VF register is set, and this way the game knows there has been a collision on screen. How are you going to get this feedback? how about holding an array of all screen pixels, since the screen is only 64x32 pixels.
- chip 8 graphics are black and white and 1-bit encoded.
- interrupts and hardware registers. Chip 8 has none, but it does have two timer registers called delay and sound timers which count about 60 times a second when set above zero until they reach zero.
- The sound delay register form the basis for a very simple sound system. The system's buzzer sounds whenever the timer is zero.

Stage 1

Ok, so chip 8 addresses 4k of memory. In a more advanced emulator, where you may have mirroring, hardware registers, banked memory and so on, the most practical solution may be to split the different types of memory (ie. rom, ram, i/o, etc) into separately emulated memory regions and use memory read and write handlers (routines) to let the CPU access these areas as a continual memory space. You will also find it faster to allocate memory and reference it with pointers than to emulate memory with arrays.
However, with the simplicity of Chip 8, and its age (believe me, you will have to TRY to slow this thing down!), the simplest solution is to use a single array.

```
unsigned char memory[0xFFF];
```

I have set up the memory as the char variable type, that is, 1 byte. Why have I done this when all chip 8 instructions are two bytes long?? Well, if part of the code is data (ie. sprites) and there is not an even number of bytes, then instructions will become unaligned and you will not be able to read them properly if you allocate memory in two byte portions. This is a nice introduction to memory alignment; in 16/32/64...bit emulation having to accomodate unaligned memory accessing can have serious implications for emulation speed.

Stage 2

Now, you are essentially ready to write 'CPU' emulaton! Refer to the end of this page for information on where you can find the full Chip 8 instruction set. Lets take the instruction:

6XKK - register[X] = KK

now this is an opcode you will come accross in any CPU you emulate, it would normally be called something like 'load register with 8 bit immediate'. Since we are not concerned about speed in this emulator, how about the following to execute this opcode;

```
opcode = ((memory[PC]<<8) + memory[PC+1]); (this is just to form the full opcode)

V[((opcode&0x0F00)>>8)]=opcode&0x00FF;
```

Remember that we have emulated all the general register in an array, so we can write to the correct register by manipulating the opcode which contains information about the register affected. In this way, all the 6--- opcodes can be emulated in one line instead of 15. For a more demanding emulator you will want to trade compactness for speed, so this would not be a very good idea. Just to clarify the method used to find the registers concerned in an operation, say the opcode is 6XKK, X denotes the register, from 0 to E (maybe F, but I don't think writes to the that register are done). If, say, X=A, so the opcode is 6AKK, we must isolate X with a logical AND; opcode&0x0F00 gives us 0A00. But this would equal 2560, so we shift the figure 8 positions to the right to give us 000A, and thus V[000A] is the register we want.

The general layout of your cpu could take the following form:

```
cpu(){
opcode = ((memory[PC]<<8) + memory[PC+1]);
switch (opcode&0xF000){
case 0x6000: ...code as above...PC+=2;break;
case:...break;
case:...break;
}
}
```

PC is a variable defined to emulate the system's program counter. As i said before, each chip 8 instruction is two bytes long, so that is why it is incremented by 2 after the opcode.

Ok, lets take another instruction.
8XY6 - register[X] = register[X] shifted right 1 position, VF= carry
This is another extremely common CPU instruction. Binary shifting is used as a means of multiplication and division, especially before the modern CPUs which have specific instructions for multiplication and division. Shifting 1 position to the right is the same as dividing by 2, shifting 1 position to the left is the same as multiplying by 2.

```
V[F]=(V[((opcode&0x0F00)>>8)]&0x1);
V[((opcode&0x0F00)>>8)]>>=1;
```

Since the VF register is to take the carry, which is the displaced bit shifted out of the right hand side of the operand register, the most obvious thing to do is to find out what this bit is before the shift is done, otherwise it will be lost! This is what the first line above is doing. Then the register (again, we identify the correct register by shifting the opcode), is shifted 1 position right using the C >> syntax. simple huh? If you are lost at this point it is probably because a) i am bad at explaining myself, or b) you do not understand shifting and logic. Not only are these things essential to CPUs, a firm grasp of them is also essential for programming, especially emulator programming!

Ok, one last instruction.

5XY0 - skip next instruction if register[X] = register[Y]
Not a totally authodox instruction, but similar to a conditional branch which you will find all the time. Again, both registers concerned are contained in the opcode, so;

```
if (V[((opcode&0x0F00)>>8)]==V[((opcode&0x00F0)>>4)])PC+=4;else PC+=2;
```

got that?. If the condition is true, we skip the next instruction, which means skipping 4 bytes, otherwise, we move two bytes forward to the next instruction as usual.

Stage 3

A small note before we preceed: normally, you would try to emulate the timings of your CPU, usually in terms of machine cycles. This is generally so that you can draw the screen and emulate sound at the correct relative intervals. However, there is no information on the timings for the Chip 8 instruction set (and remember, it was never a real machine!). Therefore, don't worry about timings, In my emulator i just ignored timing altogether with no obvious ill effects.

So you now have you complete CPU emulation, with perhaps the exception of the sprite drawing opcode. Usually it is possible to completely separate the emulation of the CPU from the emulation of the system, and combine the two when you feel ready, though in the case of Chip 8 they are inter-connected.
So we need to think about the chip 8 graphics. The display is 64x32, so you will probably not want to plot pixels at

this resolution becuase the display will look tiny. In addition to this you need to think about how you will emulation the XORing nature of the graphics. If you do not know what XOR means, you will need to find out about this and other logic instructions. However, to explain here, XOR means exclusively or. Thus, take the following 8 bit binary numbers;

```
10011011
01111100 XOR =
--------
11100111
--------
```

Thus, for a given position in the result, if either or both binary digits are a 1, then the result will have a 1, but if both are 1, the result will be zero. In terms of the screen, this means that pixels are toggled on and off by drawing to the screen. That is, writing to a pixel that is off will turn it on, but writing to a pixel that is already on will turn it off.
Before going on to explain the screen emulation, we need to think about how screen information is encoded. Chip 8 graphics are 1 bit encoded, which means each pixel is represented by one bit in a byte. Bit encoding is extremely common, but typically, at least two bits (as in gameboy) will be associated with each pixel to achieve a greater colour range.
With one bit encoding, take the byte
10011110
Each bit that is 1 represents a pixel turned on, (ie. white), and each bit that is 0 represents a pixel that is off (black).
Several bytes are used to diplay an image; e.g.
1111000,0001000,0001111 would result in a spite which looks like this:

```
****
   *
   ****
```

Which you may recognise as a tetris shape (sort of).
I am not going to spend a lot of time on Chip 8 graphics because they are not particularly representitive of general emulation, but as I will eleborate a litlle.

If we set up an array to represent the screen;
unsigned char screen[64*32];

Now the CPU sprite drawing opcode can be written so that the sprite is drawn into the array. This way it is easy to test for collisions on the screen and set the VF register accordingly. You can write a routine that draws the actual screen from the array, and you will probably want to expand each pixel into a primitive (ie. a square) to make the screen more visible.

Decoding bit encoded graphics

How can this be done?. The sprite opcode takes the following form

DXYN - draw sprite starting at coordinates held in register[X] (x axis), and register [Y] (y axis). The sprite data begins at location pointed to by the I register which will have been set by the game before this opcode is called. The sprite is 8 pixels by N pixels. So that N is 3, and starting at I we have the information give previously, which drew the tetris shape.Consider the following simplified routine

```
for (yline=0;yline<(opcode&0x000F);yline++){
data = memory[I+yline]; //this retreives the byte for a give line of pixels
for(xpix=0;xpix<8;xpix++){
if ((data&(0x80>>xpix))!=0){
if (screen[V[X] +(V[Y]*64)]==1) V[F]=1 //there has been a collision
screen[V[X] +(V[Y]*64)]^=1; //note: coordinate registers from opcode
}
}
}
```

With any luck this routine should draw sprites into your screen array!. You may wonder about the data&(0x80>>xpix) test. Well this tests the 'data' variable against 0x80 (bit 7) then 0x40 (bit 6), then 0x20(bit 5) and so on, so each bit is being evaluated from left to right. This happens because the AND operand, 0x80, gets shifted right on each loop.

In case you have not noticed, all chip 8 graphics are sprite based. In just about any other emulator you will also have at the very least one background layer, often more. Thinking about some of the peculiarities of Chip 8 opcodes, I believe the Atari 2600 does have some similar things. I believe there are register specifically for tracking missile spites accross the screen!!

Since graphics are only sprite based, after a sprite has been plotted to the screen array, it would be ok to then draw the whole screen. As I said before, I am not going to discuss this because it is simply a case of reading through the screen array and plotting pixels according to whatever graphics system you are using.

Step 4

We are now ready to emulate user input. The Chip 8 system uses a hex keypad, so only accepts 0 to F. In general emulation you will need to identify the registers which deal with input devices, and these will typically need to be updated constantly, such as once per screen refresh. This is not necessary for chip 8, because there are specific opcodes which wait for a screen press and store it in a given register. When you program these opcodes, create a loop which checks the keys you havce decided to map to the Chip 8 keypad. When a key is found to be pressed, store the number it represents (i.e. 0-F) in the register identified by the opcode, then break the loop.

Step 5

Finally, I think you have everything readty to make the emulators main emulation loop.
Once again, speed is not at all crucial in this project. In my emulator I have had to put in several delay mechansims to slow it down to a playable speed. However, we still want to learn some good practices, so make the main loop along these lines;

```
for (;;){
//call cpu to execute instructions, etc
}
```

Although this is an infinite loop and might generate compiler warnings because any code following the loop is unreachable, but this is the most effective thing to do. Why? Well this unconditional loop does not have to perform checks on each pass of the loop, so in theory is faster than say

```
do{
//call cpu to execute instructions, etc
}while(exit==0);
```

because 'exit' must be tested on every loop. The main emulation loop is running thousand of times a second and unneccesary tests on each pass are not efficient. Instead, when we want to exit the loop we will explicitly breakout of it rather than officially terminate the loop.
What goes in the loop? Well since we are not really bothering with proper timing, you will have to play around a little, but in general, an inbeded loop should run the CPU for a given number of executions, then the keyboard can be checked for exit commands, and a function to draw the screen can be called (although you may decide to draw the screen after each sprite is drawn). Experiment with the size of this imbeded loop untill the emulator is sufficiently responsive to the exit command.

One other thing to take into account is the delay and sound registers which I mentioned earlier. These need to be counted down if non zero, so this can be done in the main loop, and again the inner cpu loop should be made the correct size so that the timers count down roughly at the right speed, which is 60 times a second.

```
for(;;){
```

```
for(cpu=0;cpu<5000;cpu++){ //some arbitrary number of loops
cpu();
}
if (delay_register>0)delay_register-=1;
if (sound_register>0)sound-register-=1;
if exit key pressed, exit(0);
}
```

One thing to bear in mind for a larger emulation project; in the case above, the CPU function executes only one instruction. However, you could minimise the number of function calls by programming the CPU function to emulate a given number of instructions itself. The number will be based on the timing of the emulator which is not being discussed hear, but typically it may be equivalent to the number of machine cycles the real machine takes to draw one line to the screen, because this is typically the smallest common divisor and all other counters and timed events can be based around multiplies of this amount of time.

Step 6

Perhaps this stage should have come much earlier, as it regards loading a Chip 8 rom. As we know, the first 0x200 bytes of the Chip 8 memory map would contain the interpreter, so the game should be loaded starting from 0x200 onwards. Simply open a rom file in binary reading mode and copy the whole file into the memory array. Be very careful the rom is not loaded even 1 byte out of line, because then all jump instructions will land at the wrong place.

Conclusion

Hopefully you have everything you need here, and in the document available at the link below, to program your own Chip 8 emulator, and get playing Pong!. If you manage to do this, you will be equiped to start a bigger project, though you will find lots of complexities to deal with! Good luck.

Confession Time

By orginal author: Ok, despite everything I have said, there is one opcode in Chip 8 which I have not been able to get right. It is meant to store a binary coded decimal representation of a value held in a specified register, at memory locations I, I+1, and I+2. I take this to mean a non packed binary decimal representation, in which three bytes would be needed to represent a number above hex 99. Well, this does not seem to work. Don't worry too much about this though, you will find it is generally only used to calculate the scores in the games, and is not very crucial.

Edit bt TJA: This is actually quite simple, you simply need to convert the decimal contained in the register into a BCD, and store it back it memory this is done with the help of the mod and divide operators, for simplicty here is the ocode.

```
RAM[I] = (V[((opcode&0x0F00)>>8)]/100);
RAM[I+1] = ((V[((opcode&0x0F00)>>8)]/10)%10);
RAM[I+2] = ((V[((opcode&0x0F00)>>8)]%100)%10);
PC+=2;
```

Just a little confession of my own it you have looked at my emulator source I actually stored the BCD backwards in the first release of my emulator whoops!!!

Documentation

For details on the Chip8 instruction set and how to emulate the Chip8 'system'Click here.