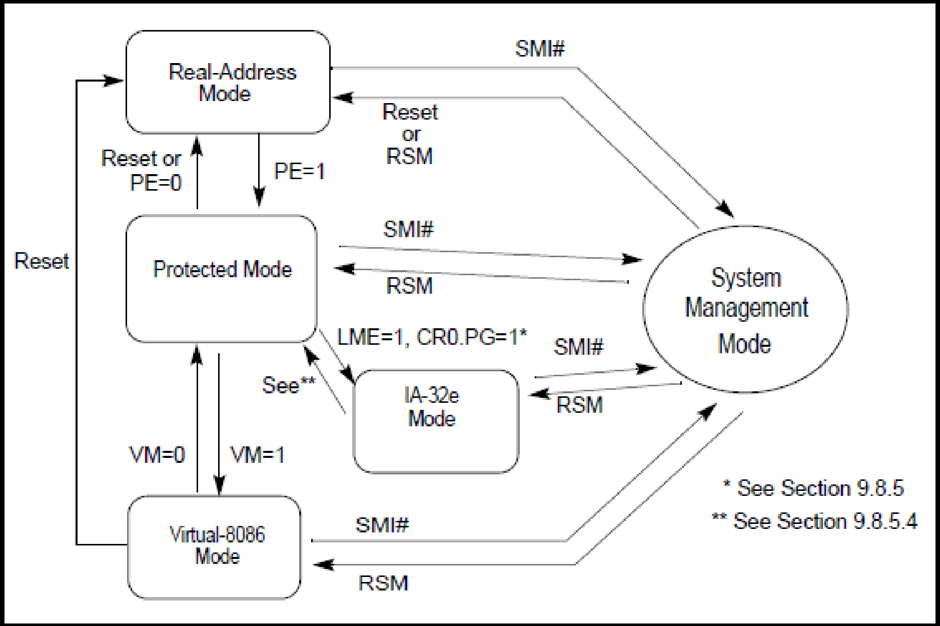
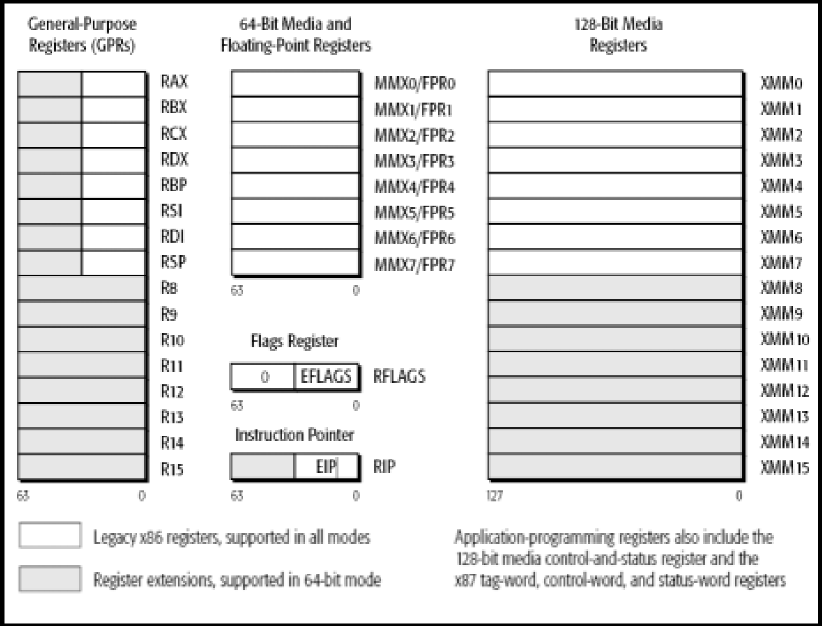
Assembly Language

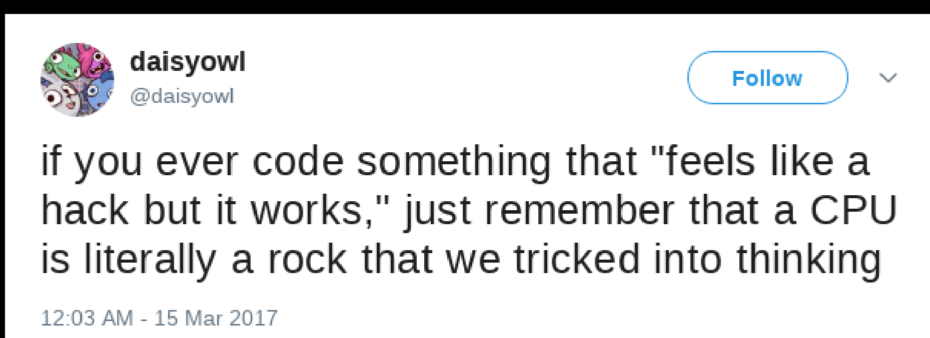
What people see

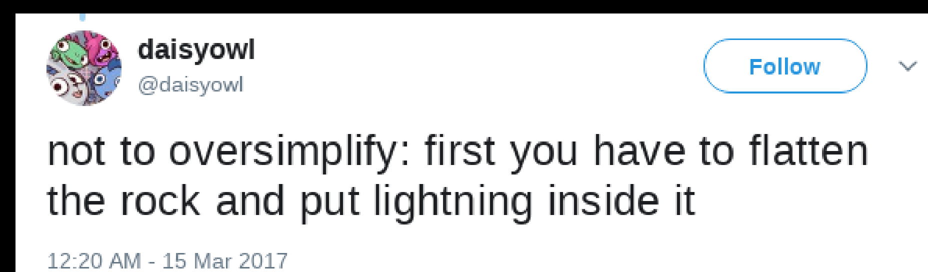




What it is

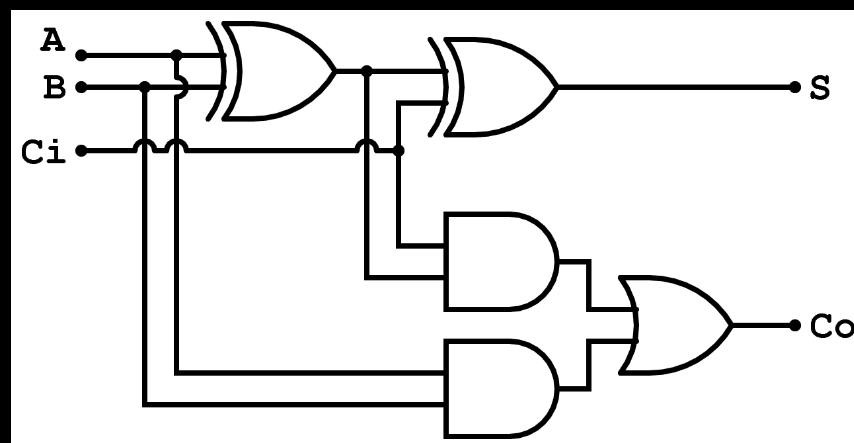
- It’s just machine language
  - A language with instructions that allows you to manipulate physical electrons to make computations happen
- Essentially, it allows you to take a circuit in the CPU and manipulate its inputs to get a desired output





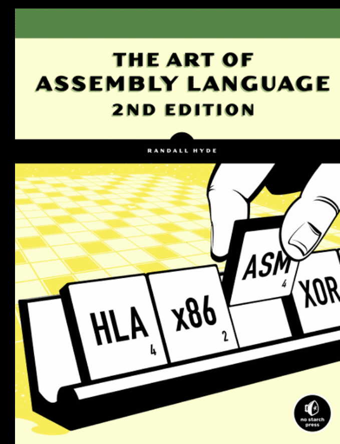
Lets code a rock so

- All thinking rocks have the following circuit



- It’s a full adder, it allows you to add a whole 8 bits together and carry a 9th bit if required
- Essentially, we can manually vary, the inputs to A and B as fast as is possible by the CPU and count it in single cycles

Choosing our environment



- Inline asm
- it's ok, but you need to know C and some complexities

- HLA is nice and simple
- A pain to install...

- The real deal
- Though also the most complex

What we need first

- A text editor
  - I’m using Atom with the *language-x86-64-assembly* for syntax highlighting
- NASM*
  - Because you can write IA-32 and AMD64 assembly and it can cross compile to work on several other arch’s

*MASM for peasants on Windows

Hello World

- Just a handy framework
- `cd` to the directory where the program is
- Assemble with:
  - `nasm -f elf hello.asm`
- Create executable
  - `ld -m elf_i386 -s -o hello hello.o`
- Execute with
  - `./hello`

```
section .text
   global _start              ;must be declared for ld linker

_start:                       ;tells linker entry point
   mov  edx,len               ;message length
   mov  ecx,msg               ;message to write
   mov  ebx,1                 ;file descriptor (stdout)
   mov  eax,4                 ;system call number (sys_write)
   int  0x80                  ;call kernel

   mov  eax,1                 ;system call number (sys_exit)
   int  0x80                  ;call kernel

section .data
msg db 'Hello, world!', 0xa   ;string to be printed
len equ $ - msg               ;length of the string
```

So lets do the circuit we looked at!

- Inputs;

  a = 2, b = 3

- Expected result;

  sum = 5

```
legendarypatman@charmander:~
nguage$ ./add
The sum is:
5legendarypatman@charmander:
```

```
section .text
   global _start                        ;must be declared for ld linker

_start:                                 ;tell linker entry point

  mov eax,'2'                           ;move 2 to eax register
  sub eax, '0'                          ;set 2 as a signed integer
  mov ebx, '3'                          ;move 3 to ebx register
  sub ebx, '0'                          ;set 3 as a signed integer
  add eax, ebx                          ;add ebx to eax
  add eax, '0'                          ;convert from decimal to ASCII to print
  ;
  mov [sum], eax                        ;set sum variable i.e. sum = eax
  mov ecx,msg                           ;delacre msg as the message to write
  mov edx, len                          ;set the message lenght
  mov ebx,1                             ;file descriptor (stdout)
  mov eax,4                             ;system call number (sys_write)
  int 0x80                              ;call kernel
  ;
  mov ecx,sum                           ;set sum as the message
  mov edx, 1                            ;set lenght 1 for len when printing
  mov ebx,1                             ;file descriptor (stdout)
  mov eax,4                             ;system call number (sys_write)
  int 0x80                              ;call kernel
  ;
  mov eax,1                             ;system call number (sys_exit)
  int 0x80                              ;call kernel

section .data
   msg db "The sum is:", 0xA,0xD        ;print msg to line(0xA) & carrige return(0xD)
   len equ $ - msg                      ;the length of msg equates(equ) to len

segment .bss
   sum resb 1                           ;declare variable sum, reserving 4 byte's
```

So why is any of this relevant?!

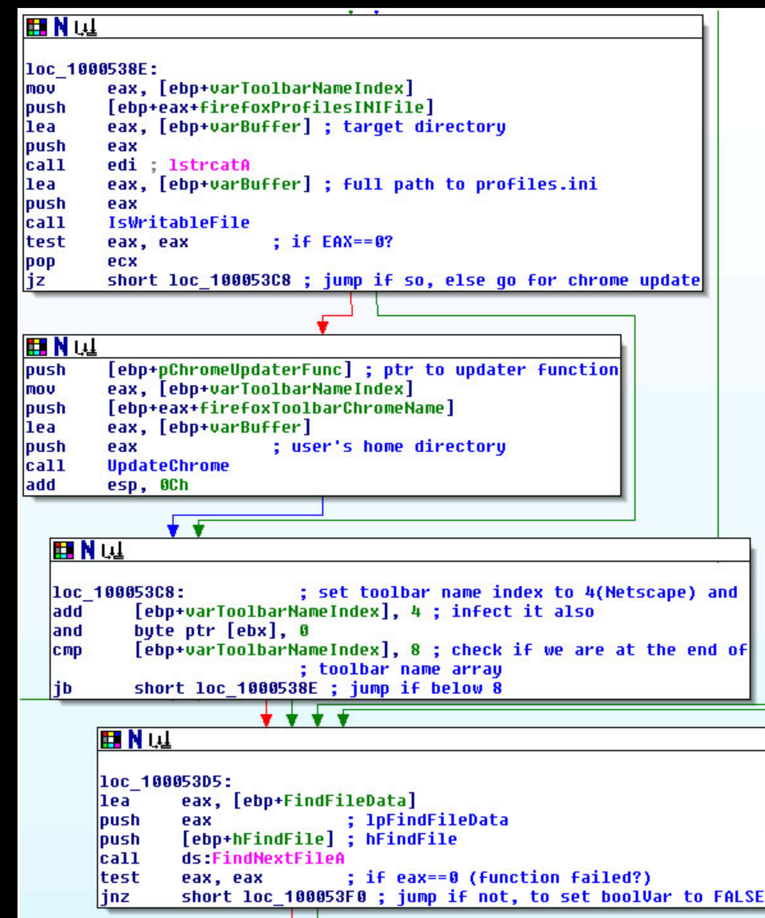
It’s the foundation of programming languages

- When a compiled program is assembled, it’s converted to assembly
- `println` ->

```
;;; ; ------------------------------------------------------------
;;; ; _println            put the cursor on the next line.
;;; ;
;;; ; Example:
;;; ;         call      _println
;;; ;
;;; ; REGISTERS MODIFIED:  NONE
;;; ; ------------------------------------------------------------
_println:
                section .data
.nl             db              10

                section .text
                push            ecx
                push            edx

                mov             ecx, .nl
                mov             edx, 1
                call            _printString

                pop             edx
                pop             ecx
                ret
```

Reverse Engineering!

- You can do RE in both asm and higher langs but finding exploits is generally seen though asm
- When you break down a file, you get a diagram in IDA like →
- If you don't know asm, you're going to have a harder time finding the exploit



Helpful to understand exploits

- SpectreV2 allows an attacker to read the data of other processes by jumping to different locations in memory via speculative execution

```
call *%r11        jmp set_up_return;
                inner_indirect_branch:
                  call set_up_target; }
                capture_spec:         }
                  pause;              }
                  jmp capture_spec;   } Indirect branch
                set_up_target:        } sequence.
                  mov %r11, (%rsp);   }
                  ret;                }
                set_up_return:
                  call inner_indirect_branch; (1)
```

- Retpoline is the fix
- It allows indirect branch's to be isolated from speculative execution effectively stopping SpectreV2

```
call *%r11        jmp set_up_return;
                .align 16;
                inner_indirect_branch:
                  call set_up_target;
                capture_spec:
                  pause;
                  jmp capture_spec;
                .align 16;
                set_up_target:
                  mov %r11, (%rsp);
                  Ret
                .align 16;
                set_up_return:
                  call inner_indirect_branch;
```