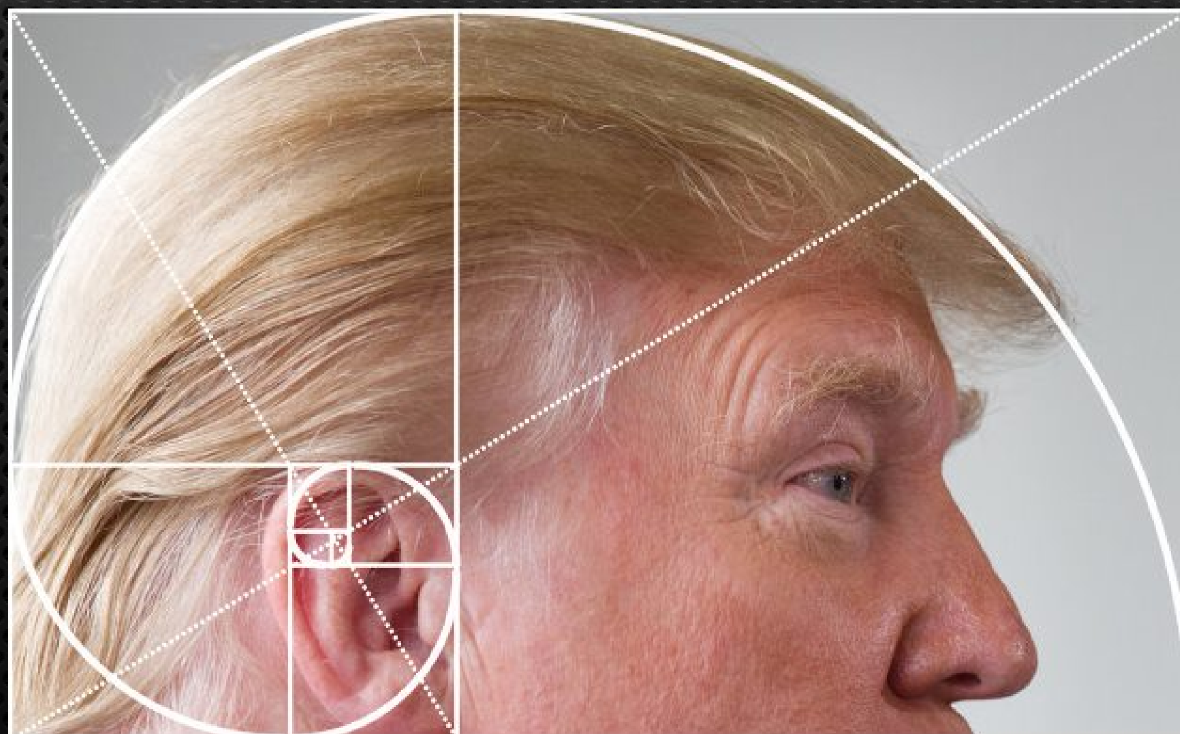
# Creating a Fibonacci Generator in Assembly



## Because ¯_(ツ)_/¯

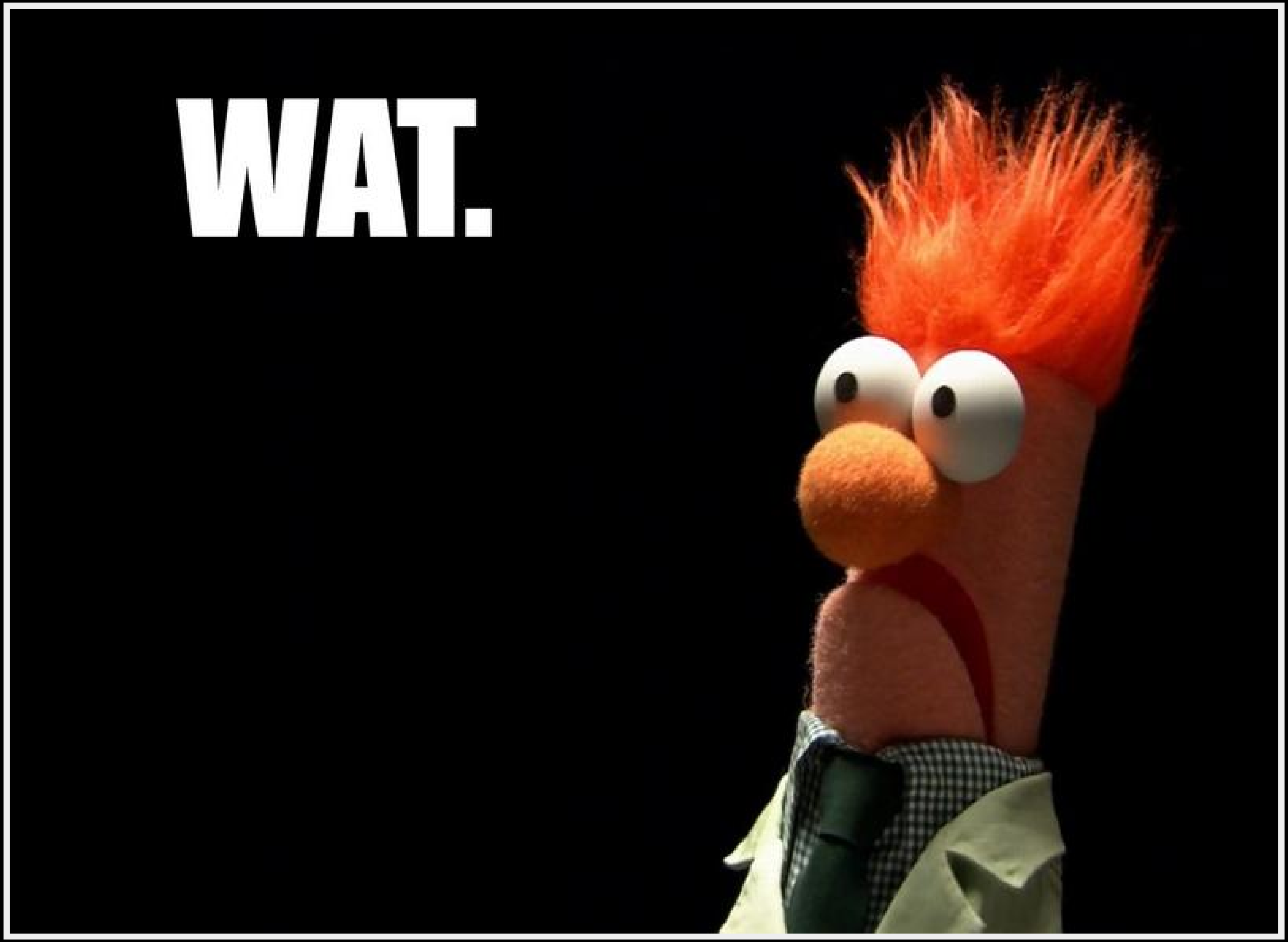
By Will van Ketwich

# @wilvk

Just a dude that does stuff... mostly with computers.

Systems Engineer at REA Group
Group Delivery Engineering, Group Technology

# Assembly



# Why???

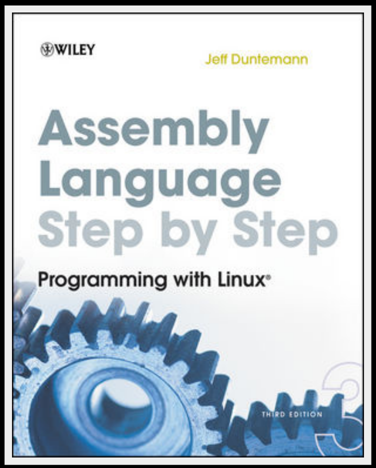
## Seriously dude, why Assembly?

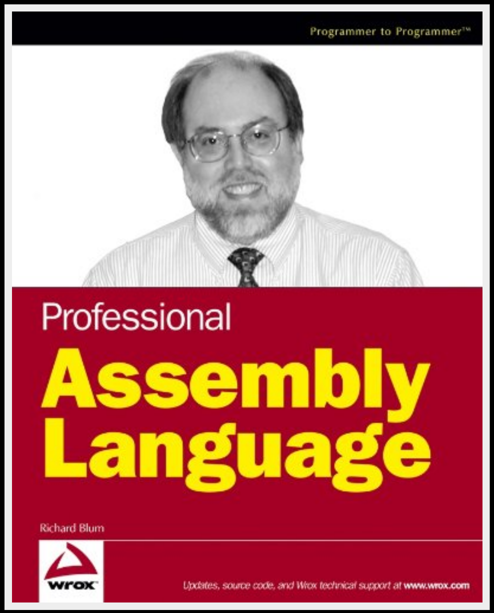
- The building blocks of all languages
- How processes work
- How to reverse engineer
- How the Linux Kernel works
- How device drivers work
- Improve troubleshooting skills
- Curiosity

# How?

## How to get started?

- Read a few books
- Do some examples
- Define a project
- Look online for help



# A Small Project

1. Reading input from the command line
2. Get length of command line argument
3. Converting input to a number
4. Generating the Fibonacci number
5. Printing the output to screen

```
$ ./fib7 46
2971215073
```

# Decisions to be made

- Should standard libraries be used? (glibc)
- What flavour of assembly to use?

NASM, GAS, MASM, YASM?

- 64 bit or 32 bit?

# The Fibonacci Sequence

# The Fibonacci Algorithm

## From Wikipedia:

The sequence $F_n$ of Fibonacci numbers is defined by the recurrence relation:

$F_n = F_{n-1} + F_{n-2}$

with seed values:

$F_1 = 1, F_2 = 1$

or:

$F_0 = 0, F_1 = 1$

# A resultant sequence

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | ... |
| Position | - | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |

# The Fibonacci Algorithm

## As C code

```
int fib(int n)
{
  int f[n+2];
  int i;
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
    f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

# Beginning our implementation

## And some starting knowledge

# GAS Files and Language Syntax

## fib1.s - doing nothing takes something

```
.section .text
.globl _start
_start:
    nop
    # the rest of our program goes here
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

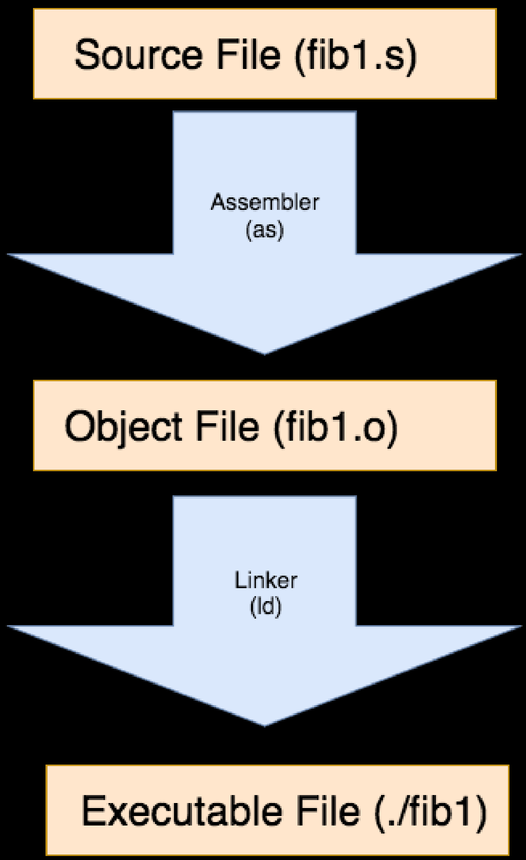
# The building blocks

## Opcodes and Operands

| Line | Opcode | Operand 1 | Operand 2 | Description |
|---|---|---|---|---|
| nop | nop | - | - | no-operation |
| movl $1, %eax | movl | $1 | %eax | copy 1 into register eax |
| movl $0, %ebx | movl | $0 | %ebx | copy 0 into register ebx |
| int $0x80 | int | $0x80 | - | call interrupt number 0x80 |

# The tools of the trade

## Assemblers and Linkers




```
as --32 -gstabs -o fib1.o fib1.s

ld -m elf_i386 -o fib1 fib1.o
```

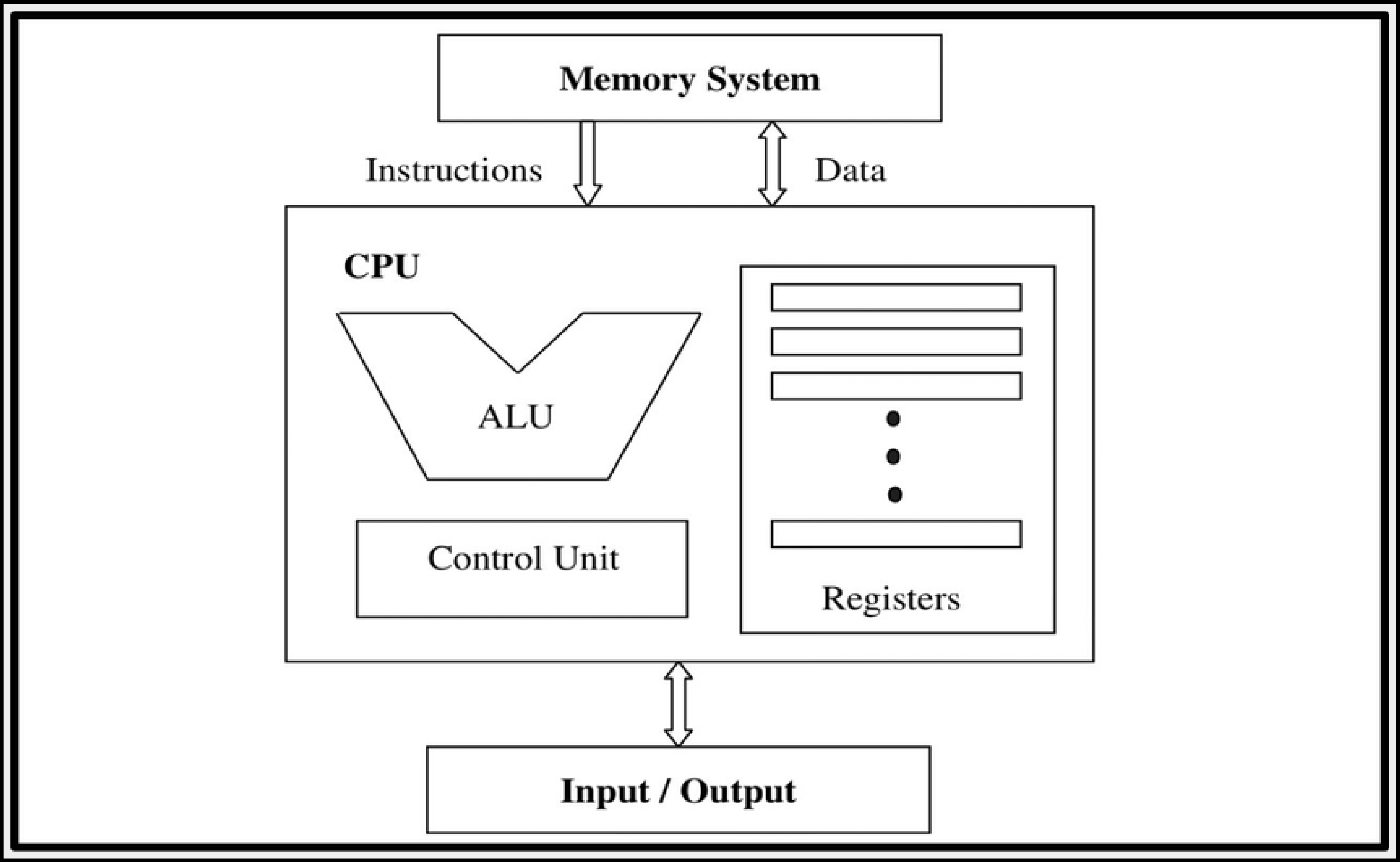
# The tools of the trade

## Object file utilities

- nm - lists the symbols from object files
- objdump - displays more information about object files
- elfdump - similar to objdump
- gdb - GNU debugger
- gcc - GNU compiler collection
- make - used for build toolchains

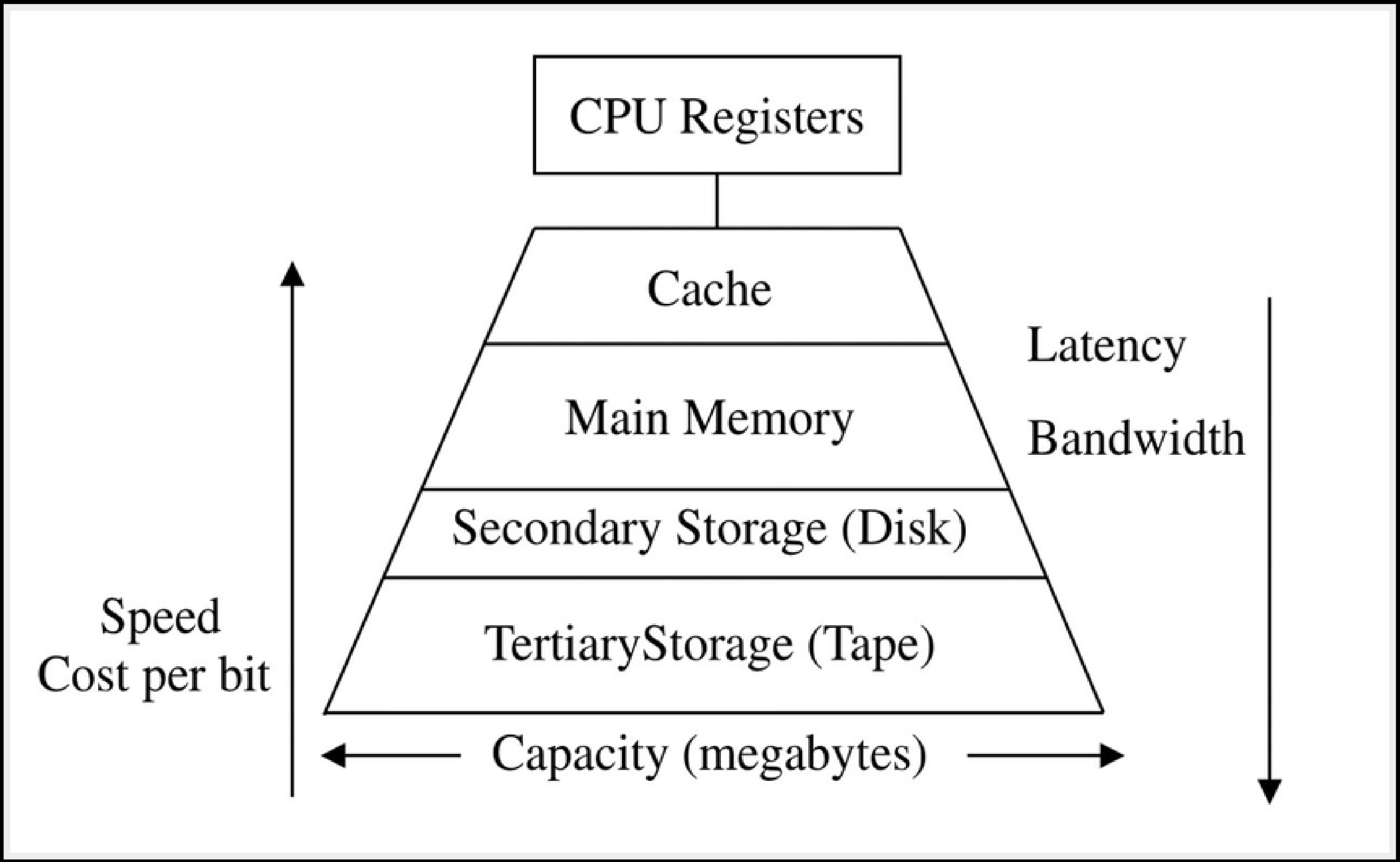
# The canvas

## CPU and Registers



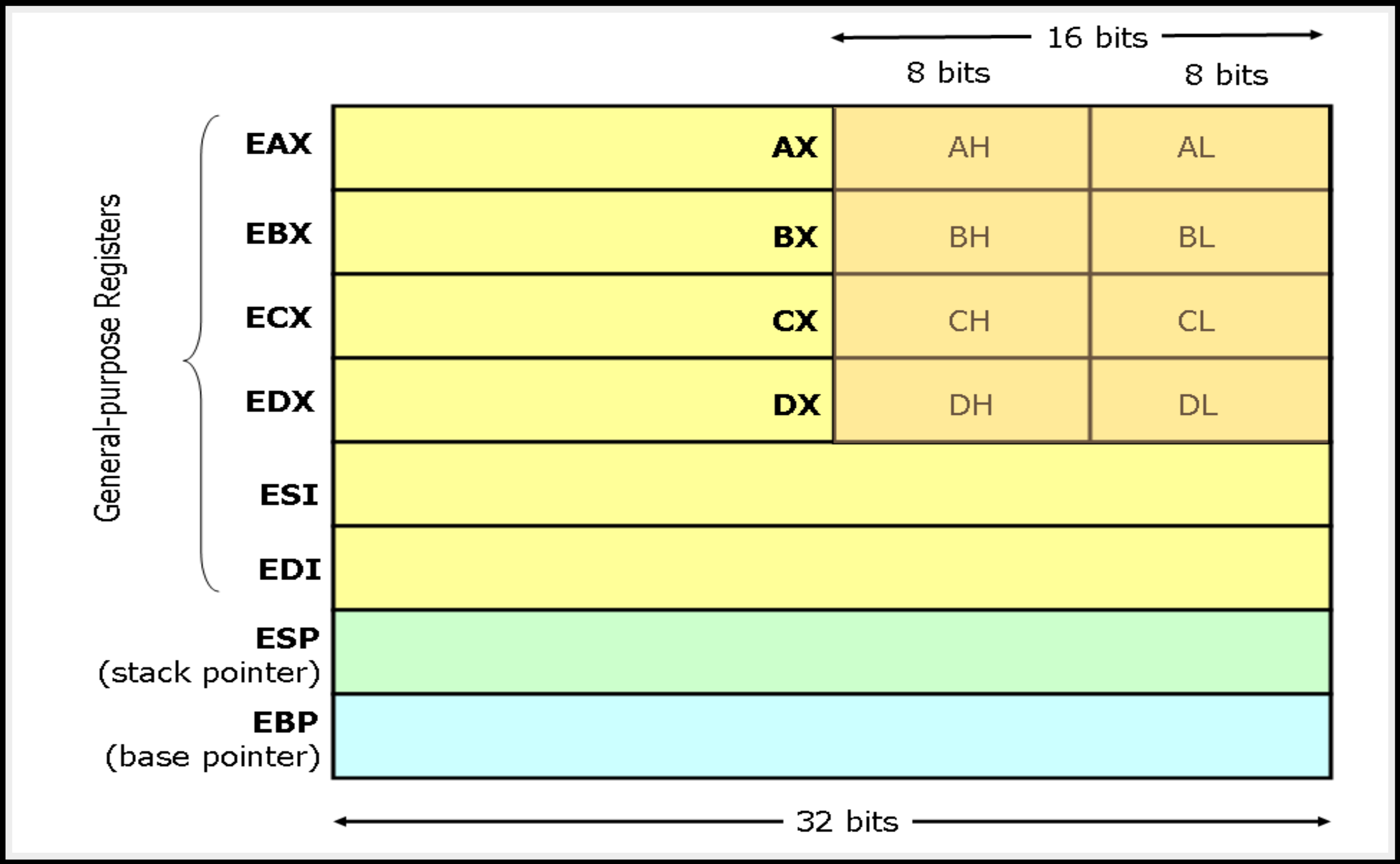
# The canvas

## Memory Hierarchy



# The canvas

## Registers



# NOP

- Does nothing functional
- Takes up a clock cycle
- Can be used to fill space

```
nop
```

# MOV

- MOVes (or copies) data from one location to another
- Can copy to/from memory, register and immediates
- Cannot copy from memory to memory

Some examples:

```
mov (%ebx), %eax # copy address pointed to by ebx to register eax
mov %ebx, (%eax) # copy value in ebx to address pointed to by eax
mov $1, %eax     # place value 1 into register eax
mov $5, (%eax)   # place value 5 into address pointed to by eax
mov %ebp, %esp   # copy value in register ebp into register esp
```

# MOV - Operation Suffixes

- Different variations based on size to copy
- Used for many instructions (MOVL, PUSHL, POPL, etc.)
- Floating point versions used with the FPU and SIMD

| Suffix | Type | Bits (Integer) | Bits (Floating Point) |
|---|---|---|---|
| b | byte | 8 | - |
| s | short | 16 | 32 |
| w | word | 16 | - |
| l | long | 32 | 64 |
| q | quad | 64 | - |
| t | ten bytes | - | 80 |

Some examples:

```
movb $2, (%ebx)  # moves 0x02 into ebx
movw $2, (%ebx)  # moves 0x0002 into ebx
movl $2, (%ebx)  # moves 0x00000002 into ebx
```

# INT

- Interrupts the CPU from scheduled processing
- Most interrupts are hardware-based
- Software-based interrupts are called 'soft-interrupts'
- Requesting actions from User Space in Kernel Space
  - Linux uses INT 0x80
  - MacOS uses INT 0x60
  - Windows uses INT 0x21

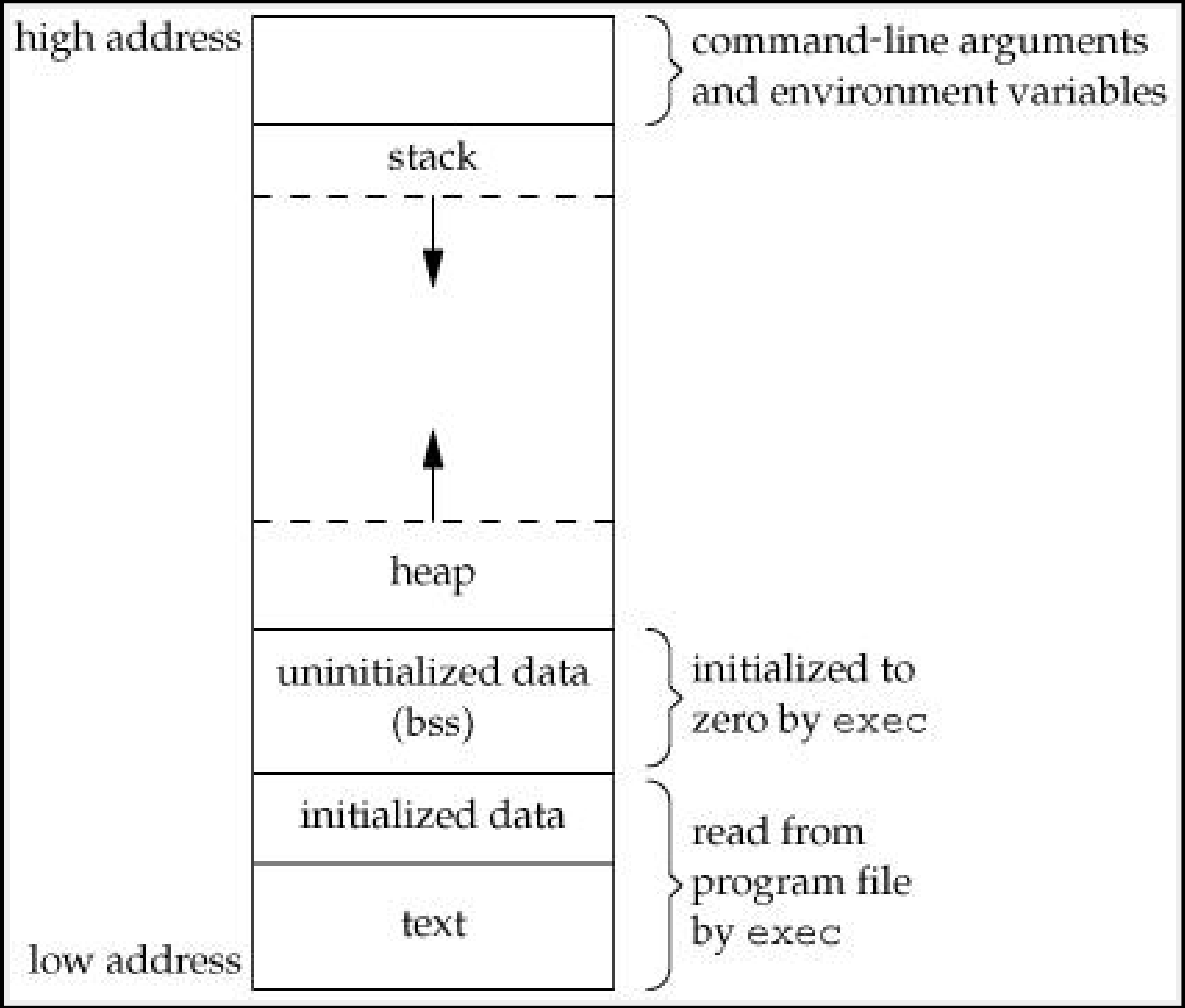
```
# exit with return code 0
movl $1, %eax
movl $0, %ebx
int 0x80
```

- Modern CPUs/OSes use SYSCALL but INT is still valid

# 1. Reading input from the command line

1. Reading input from the command line
2. Get length of command line argument
3. Converting input to a number
4. Generating the Fibonacci number
5. Printing the output to screen

# Memory Layout



# Command Line Arguments in the Stack

| Stack | Data |
|---|---|
| ... | |
| ESP + n+8 | < pointer to second environment variable > |
| ESP + n+4 | < pointer to first environment variable > |
| ESP + n | < NULL for end of command line arguments > |
| ESP + n-4 | < pointer to last string argument > |
| ... | |
| ESP + 12 | < pointer to second string argument > |
| ESP + 8 | < pointer to a string that contains the first argument > |
| ESP + 4 | < pointer to a string containing the name of the application > |
| ESP + 0 | < number of arguments on command line > |
| ... | |

# A Simple Print Utility

## fib2.s

```
# stack args example
.section .text
.globl _start
  _start:

    nop
    movl %esp, %ebp     # take a copy of the stack pointer esp into ebp
    addl $8, %ebp       # address of first arg in stack
    movl (%ebp), %ecx   # move the address of the first arg into ecx

    movl $4, %edx       # set the length of our string to 4
    movl $4, %eax       # indicate to int 0x80 that we are doing a write
    movl $0, %ebx       # indicate to int 0x80 that we are writing to file desc
    int $0x80           # call int 0x80 for write

    movl $1, %eax       # exit gracefully
    movl $0, %ebx       # with return code 0
    int $0x80           # call int 0x80 for exit
```

# Viewing the Stack in GDB

## A demo

# 2. Get length of command line argument

1. Reading input from the command line
2. Get length of command line argument
3. Converting input to a number
4. Generating the Fibonacci number
5. Printing the output to screen

# Approach

- Find address of first command line argument string
- Get length of string
- Print string to stdout

# Source

## fib3.s

```
# framework - get first argument from the command line and print to stdout
.section .text
.globl _start
_start:
    nop
    movl %esp, %ebp     # take a copy of esp to use
    addl $8, %ebp       # address of first arg in stack
    movl (%ebp), %edi   # move arg address into esi for scasb
    push %edi           # store the string address as edi gets clobbered

    movl $50, %ecx      # set ecx counter to a high value
    movl $0, %eax       # zero al search char
    movl %ecx, %ebx     # copy our max counter value to edx
    cld                 # set direction down
    repne scasb         # iterate until we find the al char

    subl %ecx, %ebx     # subtract from our original ecx value
    dec %ebx            # remove null byte at the end of the string from the count
    pop %ecx            # restore our string address into ecx
    mov %ebx, %edx      # move our count value to edx for the int 80 call

    movl $4, %eax       # set eax to 4 for int 80 to write to file
    movl $0, %ebx       # set ebx for file to write to as stdoout (file descriptor 0)
    int $0x80           # make it so
    movl $1, %eax       # set eax for int 80 for system exit
    movl $0, %ebx       # set ebx for return code 0
    int $0x80           # make it so again
```

# Source

## New Code

```
movl (%ebp), %edi     # move arg address into esi for scasb
push %edi             # store the string address as edi gets clobbered

movl $50, %ecx        # set ecx counter to a high value
movl $0, %eax         # zero al search char
movl %ecx, %ebx       # copy our max counter value to edx
cld                   # set direction down
repne scasb           # iterate until we find the al char

subl %ecx, %ebx       # subtract from our original ecx value
dec %ebx              # remove null byte at the end of the string from the co
pop %ecx              # restore our string address into ecx
mov %ebx, %edx        # move our count value to edx for the int 80 call
```

# repne scasb

## Scanning Strings

- edi - points to address of string to scan
- al - stores the byte to scan until
- ecx - stores the result of the scan count

- requires registers set up before running (like INT)

- the cld opcode clears the direction flag
- each iteration increases edi and decreases ecx

# Some Arithmetic

ecx = 50 - ( len(string) + 1 )

but we want len(string)

len(string) + 1 = 50 - ecx
len(string) = (50 - ecx) - 1

```
subl %ecx, %ebx      # subtract from our original ecx value
dec %ebx             # remove null byte at the end of the string from the
```

# Labels, CALL, RET and the Stack

## labels

- points to an addresses in memory
- label addresses resolved with ld

Standard labels:

```
my_function:
```

Local labels:

```
.my_local_function:
```

# Labels, CALL, RET and the Stack

## CALL and RET

CALL:

- places next instruction address on the stack
- jumps execution to address of label

RET:

- RET jumps execution back to the next address on the stack
- functions are just a logical placing of CALL and RET
- no set order to location of labels

# Functions

## A Simple Example

```
call get_string_length
mov %eax, %ebx

...

get_string_length:
mov $1, %eax
ret
```

# The implementation

## fib4.s

```
# framework - refactor into separate functions
.section .text
.globl _start

# entrypoint of application
_start:
    nop
    movl %esp, %ebp     # take a copy of esp to use
    addl $8, %ebp       # address of first arg in stack
    movl (%ebp), %edi   # move arg address into esi for scasb
    push %edi           # store the string address as edi gets clobbered
    call get_string_length
    pop %ecx            # restore our string address into ecx
    call print_string
    call exit

# get length of string pointed to by edi and place result in ebx
get_string_length:
    movl $50, %ecx      # set ecx counter to a high value
    movl $0, %eax       # zero al search char
    movl %ecx, %ebx     # copy our max counter value to edx
    cld                 # set direction down
    repne scasb         # iterate until we find the al char
    movl %ecx, %edx     # move count into edx
    subl %ecx, %ebx     # subtract from our original ecx value
    dec %ebx            # remove null byte at the end of the string from the count
    ret

# print the string in ecx to the length of ebx
print_string:
    mov %ebx, %edx      # move our count value to edx for the int 80 call
    movl $4, %eax       # set eax to 4 for int 80 to write to file
    movl $0, %ebx       # set ebx for file to write to as stdoout (file descriptor 0)
    int $0x80           # make it so
    ret

# exit the application
exit:
    movl $1, %eax       # set eax for int 80 for system exit
    movl $0, %ebx       # set ebx for return code 0
```

# 3. Converting input to a number

1. Reading input from the command line
2. Get length of command line argument
3. Converting input to a number
4. Generating the Fibonacci number
5. Printing the output to screen

# long_from_string

## As (pseudo) C code

```
long long_from_string(*char number_string)
{
  int i=0;
  long return_value=0;
  int temp_value=0;
  char digit = number_string[i];

  while(digit >= ‘0’ && digit <= ‘9’)
  {
    temp_value = digit - 48;
    return_value *= 10;
    return_value += temp_value;
    digit = number_string[++i];
  }

  return return_value;
```

# long_from_string

## fib5.s

```
long_from_string:
    xor %eax, %eax # set eax as our result register
    xor %ecx, %ecx # set ecx(cl) as our temporary byte register
.top:
    movb (%edi), %cl
    inc %edi
    cmpb $48, %cl  # check if value in ecx is less than ascii '0'. exit
    jl .done
    cmpb $57, %cl  # check if value in ecx is greater than ascii '9'. ex
    jg .done
    sub $48, %cl
    imul $10, %eax
    add %ecx, %eax
    jmp .top
.done:
    ret
```

# ASCII

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | Null | 32 | 20 | Space | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | Start of heading | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | Start of text | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | End of text | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | End of transmit | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | Enquiry | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | Acknowledge | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | Audible bell | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | Backspace | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | Horizontal tab | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | Line feed | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | Vertical tab | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | Form feed | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | Carriage return | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | Shift out | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | Shift in | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | Data link escape | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | Device control 1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | Device control 2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | Device control 3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | Device control 4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | Neg. acknowledge | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | Synchronous idle | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | End trans. block | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | Cancel | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | End of medium | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | Substitution | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | Escape | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | File separator | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | Group separator | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | Record separator | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | Unit separator | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | □ |

# Compare and Jump Opcodes

- CMP compares two values and sets flags based on the result
- Flags include carry, zero, overflow/underflow
- The flags are used to jump to a label/address

Many types of jump flags:

- jg = jump if greater than
- jl = jump if less than
- je = jump if equal
- jge = jump if greater than or equal
- jle = Jump if less than or equal
- jmp = jump unconditionally
- and so on.

# Relevance to Structured Programming

- Jumps, Compares and Labels
- basis of Structured Programming Constructs
- counters and comparisons

Can create:

- Selection - if..else..then
- Iteration - while, repeat, for, do..while

# 4. Generating the Fibonacci Number

1. Reading input from the command line
2. Get length of command line argument
3. Converting input to a number
4. Generating the Fibonacci number
5. Printing the output to screen

# The Fibonacci Function

## fib6.s

```
# input: eax holds our fibonacci n
# processing: iterate the fibonacci sequence n times
# output: return our fibonacci result in ebx

fibonacci:
    pushl %ebp       # preserve ebp
    mov %esp, %ebp   # copy the stack pointer to ebp for use
    mov %eax, %ebx   # make a cpoy of our fib(n) value for allocating an array on the stack
    addl $2, %ebx    # add 2 extra spaces to the array size in case n=1 or n=1
    subl %ebx, %esp  # add the size of our array to the stack to allocate the required space
    xor %ecx, %ecx   # set our counter to zero
    movl %ecx, (%esp, %ecx, 4)  # initialise our array with 0 for esp[0]
    incl %ecx                   # increase our counter
    movl %ecx, (%esp, %ecx, 4)  # initialise our array with 1 for esp[1]
    incl %ecx                   # our counter/iterator should be at 2 now

.fib_loop:
    cmp %eax, %ecx                # compare our counter (ecx) to n (eax)
    jge .fib_done                 # if it's greater or equal, we're done
    movl -4(%esp, %ecx, 4), %ebx  # get the value in the stack at esp-1 from our current stack pointer
    movl -8(%esp, %ecx, 4), %edx  # get the value in the stack esp-2 from our current stack pointer loc
    addl %edx, %ebx               # add the values esp-1 and esp-2 together
    movl %ebx, (%esp, %ecx, 4)    # place the result in the current stack location
    incl %ecx                     # bump our counter
    jmp .fib_loop                 # loop again

.fib_done:
    movl %ebp, %esp               # move our copy of the stack pointer back to esp
    popl %ebp                     # retrieve the original copy of ebp from the stack
```

# Coming back to our C algorithm

```
int fib(int n)
{
  int f[n+2];
  int i;
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
    f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

# Creating stack space

## fib6.s

```
fibonacci:
    pushl %ebp        # preserve ebp
    mov %esp, %ebp   # copy the stack pointer to ebp for use

    mov %eax, %ebx   # make a cpoy of our fib(n) value for allocating an array on the sta
    addl $2, %ebx    # add 2 extra spaces to the array size in case n=1 or n=1
    shl $2, %ebx     # multiply by 4 as we are using longs (32 bits)
    subl %ebx, %esp # add the size of our array to the stack to allocate the required sp

    xor %ecx, %ecx               # set our counter to zero
    movl %ecx, (%esp, %ecx, 4)  # initialise our array with 0 for esp[0]
    incl %ecx                    # increase our counter
    movl %ecx, (%esp, %ecx, 4)  # initialise our array with 1 for esp[1]
    incl %ecx                    # our counter/iterator should be at 2 now
```

# Saving the stack pointer (esp)

## When modifying ESP in a function

```
fibonacci:
    pushl %ebp      # preserve ebp as we are going to use it to store our stack pointer
    mov %esp, %ebp  # copy the stack pointer to ebp for use

...

.fib_done:
    movl %ebp, %esp # move our copy of the stack pointer back to esp
    popl %ebp       # retrieve the original copy of ebp from the stack
    ret
```

# Variable initialisation

- using longs (32 bit values)
- each long is 4 bytes

- we shift left twice to multiply by 4

```
shl $2, %ebx
```

- our stack grows downward, hence subtract for more space

```
subl %ebx, %esp
```

# Indexed memory MOV formats

## ABSOLUTE INDEXED MEMORY LOCATION

copy ecx into the memory address esp + ( 4 * ecx)

```
movl %ecx, (%esp, %ecx, 4)
```

## ABSOLUTE AND RELATIVE INDEXED MEMORY

copy the memory location at ( esp + ( 4 * ecx) - 4 ) into ebx

```
movl -4(%esp, %ecx, 4), %ebx
```

## THE GENERAL RULE IS:

```
relative_offset(absolute_offset, index, size)
```

# The core of the Fibonacci implementation

```
.fib_loop:                        # we begin our for loop here
    cmp %eax, %ecx                # compare our counter (ecx) to n (eax) if it's greater
    jge .fib_done
    movl -4(%esp, %ecx, 4), %ebx  # get the value in the stack at esp-1 from our current
    movl -8(%esp, %ecx, 4), %edx  # get the value in the stack esp-2 from our current st
    addl %edx, %ebx               # add the values esp-1 and esp-2 together
    movl %ebx, (%esp, %ecx, 4)    # place the result in the current stack location
    incl %ecx                     # bump our counter
    jmp .fib_loop                 # loop again
```

# 5. Printing the output to screen

1. Reading input from the command line
2. Get length of command line argument
3. Converting input to a number
4. Generating the Fibonacci number
5. Printing the output to screen

# print_long
## fib7.s

```
print_long:
    push %ebp
    mov %esp, %ebp              # copy the stack pointer to ebp for use
    add $10, %esp               # add 10 to esp to make space for our string
    mov $10, %ecx               # set our counter to the end of the stack space allocated (higher)
    mov %ebx, %eax              # our value ebx is placed into eax for division
.loop_pl:
    xor %edx, %edx              # clear edx for the dividend
    mov $10, %ebx               # ebx is our divisor so we set it to divide by 10
    div %ebx                    # do the division
    addb $48, %dl               # convert the quotient to ascii
    movb %dl, (%esp, %ecx, 1)   # place the string byte into memory
    dec %ecx                    # decrease our counter (as we are working backwards through the number)
    cmp $0, %ax                 # exit if the remainder is 0
    je .done_pl
    jmp .loop_pl                # loop again if necessary
.done_pl:
    addl %ecx, %esp             # shift our stack pointer up to the start of the buffer
    incl %esp                   # add 1 to the stack pointer for the actual first string byte
    sub $10, %ecx               # as we are counting down, we subtract 10 from ecx to give the actual number o
    neg %ecx                    # convert to a positive value
    mov %ecx, %edx              # move our count value to edx for the int 80 call
    mov %esp, %ecx              # move our string start address into ecx
    movl $4, %eax               # set eax to 4 for int 80 to write to file
    movl $0, %ebx               # set ebx for file to write to as stdoout (file descriptor 0)
    int $0x80                   # make it so
    movl %ebp, %esp             # move our copy of the stack pointer back to esp
    popl %ebp                   # retrieve the original copy of ebp from the stack
    ret
```

# Overview

## From comments (for brevity)

```
# allocate space on the stack for 10 characters

# start a loop and check if value is zero - jump to done if so

# divide the number by 10 and take the remainder as the first dig

# add 48 to the number to make it an ascii value

# store the byte in the address esp + ecx

# jump to start of loop
```

- Essentially the opposite of long_to_string

# DIV

## division

- a / b = y remainder z
- dividend / divisor = [ quotient, remainder ]
- eax / ebx = [ eax, edx ]

```
...

    mov %ebx, %eax                # our value ebx is placed into eax for division
.loop_pl:
    xor %edx, %edx                # clear edx for the dividend
    mov $10, %ebx                 # ebx is our divisor so we set it to divide by 10
    div %ebx                      # do the division
    addb $48, %dl                 # convert the quotient to ascii

...
```

# NEG

## multiplying by -1

```
neg %eax
```

- takes two's compliment of number
- most significant bit indicates sign
- 1 = negative, 0 = positive
- number inverted and 1 added to result
- loose absolute range (eg. 8 bits holds -128 to 127)
- NEG(-12) = 12
- 11110100 -> 00001011 + 1 = 00001100

# Printing to stdout
## and shuffling some registers

```
mov %ecx, %edx                  # move our count value to edx for the int 80 call
mov %esp, %ecx                  # move our string start address into ecx
movl $4, %eax                   # set eax to 4 for int 80 to write to file
movl $0, %ebx                   # set ebx for file to write to as stdoout (file descriptor 0
int $0x80                       # make it so
```

# Conclusion

## Results and limitations

- working program

```
_start:
    nop
    movl %esp, %ebp              # take a copy of esp to use
    addl $8, %ebp                # address of first arg in stack
    movl (%ebp), %edi            # move arg address into esi for scasb
    call get_string_length       # get the length of the argument string passed in
    call long_from_string        # get a numeric value from our argument
    call check_valid_input       # if the result is zero, exit
    call fibonacci               # run our fibonacci sequence
    call print_long              # print the last value in the sequence
    call exit                    # exit gracefully
```

- no error checking
- limited by 32-bit
- sub-optimal code

# Conclusion

Most of the work doing standard library things

# Conclusion

## Insight into language composition

- Jumps are GOTOs
- Frowned upon in higher level languages - Djikstra
- Fundamental units of operation in any application

# Conclusion

## It was fun!

I got some HackerNews rep from a blog post on it!

21. ▲ ML-powered newspaper for showing news from many political perspectives (knowherenews.com)
30 points by jadk157 8 hours ago | hide | 15 comments

22. ▲ How social-media platforms dispense justice (economist.com)
57 points by privong 13 hours ago | hide | 25 comments

23. BuildZoom (YC W13) is hiring for a new team: junior sales/ops talent
2 hours ago | hide

24. ▲ Spontaneous knotting of an agitated string (2007) (nih.gov)
53 points by ColinWright 11 hours ago | hide | 21 comments

25. ▲ Companies worry more about access to software developers than capital (cnbc.com)
111 points by ScottWRobinson 8 hours ago | hide | 119 comments

26. ▲ Beijing's Three Options: Unemployment, Debt, or Wealth Transfers (carnegieendowment.org)
112 points by mooreds 11 hours ago | hide | 99 comments

27. ▲ Creating a Fibonacci Generator in Assembly (seso.io)
11 points by willvk 5 hours ago | hide | 4 comments

28. ▲ NSA metadata program "consistent" with Fourth Amendment, Kavanaugh once argued (arstechnica.com)
159 points by colejohnson66 8 hours ago | hide | 59 comments

29. ▲ A year later, Equifax has faced little fallout from losing data (techcrunch.com)

# Conclusion
## Call to action

- Learning continuum
- Learn as part of something larger

Helps with understanding of:

- LLVM
- eBPF
- GPU Assembly
- WebAssembly

# Thank you!

Questions?

@wilvk