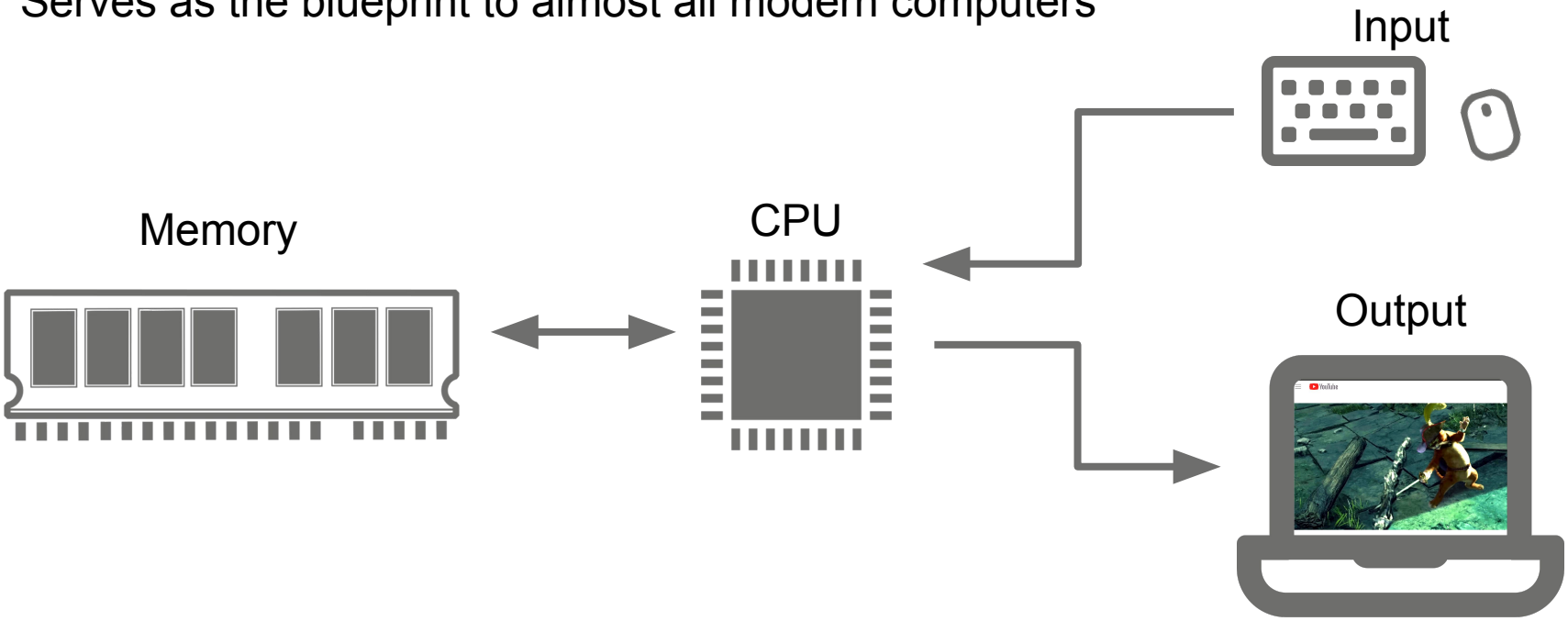
CSC 405
Assembly

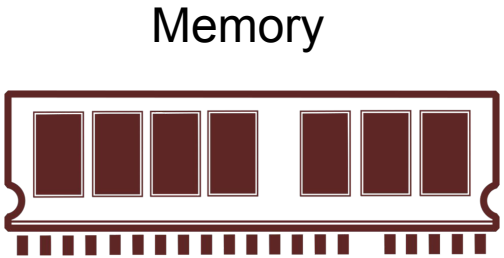
Aleksandr Nahapetyan
anahape@ncsu.edu
(Slides adapted from Dr. Kapravelos)

The von Neumann Architecture

Serves as the blueprint to almost all modern computers



The von Neumann Architecture



Memory holds two types of information:

Data Items

- variables, objects, etc.
- Read **from** or written **to**

Program Instructions

- machine code
- Code, but converted into 'binary words'

Both are stored in memory as binary numbers in a continuous array of fixed width (also known as **words**) and have a unique **address**

Compiling Programs

Let's take a look at a simple C program

```c
#include <stdio.h>

int main() {
  // Create an integer with the initial value of 42
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Compiling Programs

We can compile C programs using **`gcc`** to generate a **binary** executable

```c
#include <stdio.h>

int main() {
  // Create an integer with the initial value of 42
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

```
gcc simple.c -o simple
```

Using **`gcc`**, compile **`simple.c`** and output its binary as **`simple`**

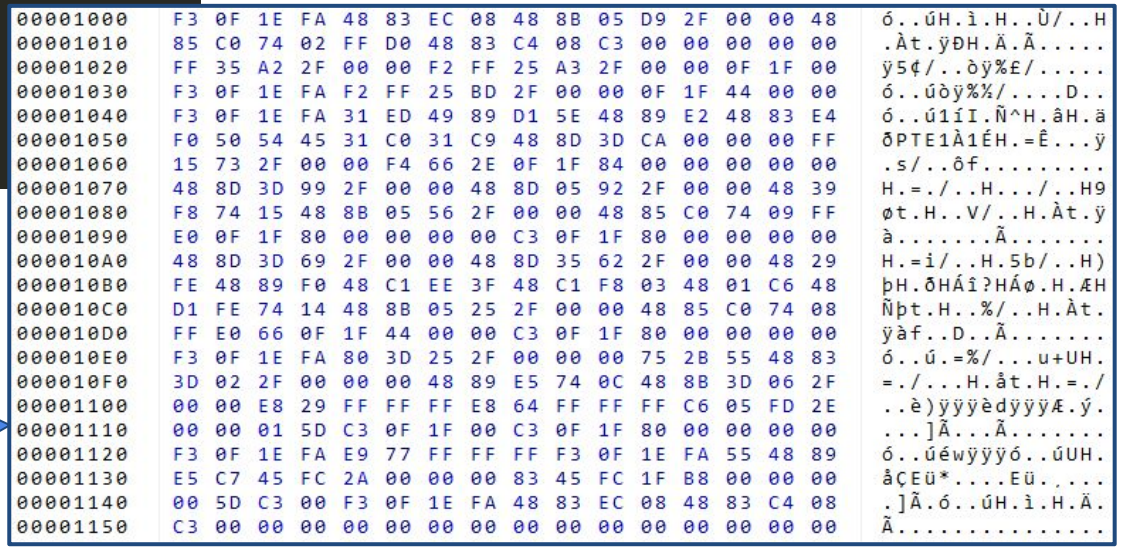
Compiling Programs

We can compile C programs using **gcc** to generate a **binary** executable

```c
#include <stdio.h>

int main() {
  // Create an integer with the initial value of 42
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

```
gcc simple.c -o simple
```

It will translate things into binary!



```
00001000  F3 0F 1E FA 48 83 EC 08 48 8B 05 D9 2F 00 00 48  ó..úH.ì.H..Ù/..H
00001010  85 C0 74 02 FF D0 48 83 C4 08 C3 00 00 00 00 00  .Àt.ÿÐH.Ä.Ã.....
00001020  FF 35 A2 2F 00 00 F2 FF 25 A3 2F 00 00 0F 1F 00  ÿ5¢/..òÿ%£/.....
00001030  F3 0F 1E FA F2 FF 25 BD 2F 00 00 0F 1F 44 00 00  ó..úòÿ%½/....D..
00001040  F3 0F 1E FA 31 ED 49 89 D1 5E 48 89 E2 48 83 E4  ó..ú1íI.Ñ^H.âH.ä
00001050  F0 50 54 45 31 C0 31 C9 48 8D 3D CA 00 00 00 FF  ðPTE1À1ÉH.=Ê...ÿ
00001060  15 73 2F 00 00 F4 66 2E 0F 1F 84 00 00 00 00 00  .s/..ôf.........
00001070  48 8D 3D 99 2F 00 00 48 8D 05 92 2F 00 00 48 39  H.=./..H.../..H9
00001080  F8 74 15 48 8B 05 56 2F 00 00 48 85 C0 74 09 FF  øt.H..V/..H.Àt.ÿ
00001090  E0 0F 1F 80 00 00 00 00 C3 0F 1F 80 00 00 00 00  à.......Ã.......
000010A0  48 8D 3D 69 2F 00 00 48 8D 35 62 2F 00 00 48 29  H.=i/..H.5b/..H)
000010B0  FE 48 89 F0 48 C1 EE 3F 48 C1 F8 03 48 01 C6 48  þH.ðHÁî?HÁø.H.ÆH
000010C0  D1 FE 74 14 48 8B 05 25 2F 00 00 48 85 C0 74 08  Ñþt.H..%/..H.Àt.
000010D0  FF E0 66 0F 1F 44 00 00 C3 0F 1F 80 00 00 00 00  ÿàf..D..Ã.......
000010E0  F3 0F 1E FA 80 3D 25 2F 00 00 00 75 2B 55 48 83  ó..ú.=%/...u+UH.
000010F0  3D 02 2F 00 00 00 48 89 E5 74 0C 48 8B 3D 06 2F  =./...H.åt.H.=./
00001100  00 00 E8 29 FF FF FF E8 64 FF FF FF C6 05 FD 2E  ..è)ÿÿÿèdÿÿÿÆ.ý.
00001110  00 00 01 5D C3 0F 1F 00 C3 0F 1F 80 00 00 00 00  ...]Ã...Ã.......
00001120  F3 0F 1E FA E9 77 FF FF FF F3 0F 1E FA 55 48 89  ó..úéwÿÿÿó..úUH.
00001130  E5 C7 45 FC 2A 00 00 00 83 45 FC 1F B8 00 00 00  åÇEü*....Eü.¸...
00001140  00 5D C3 00 F3 0F 1E FA 48 83 EC 08 48 83 C4 08  .]Ã.ó..úH.ì.H.Ä.
00001150  C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  Ã...............
```

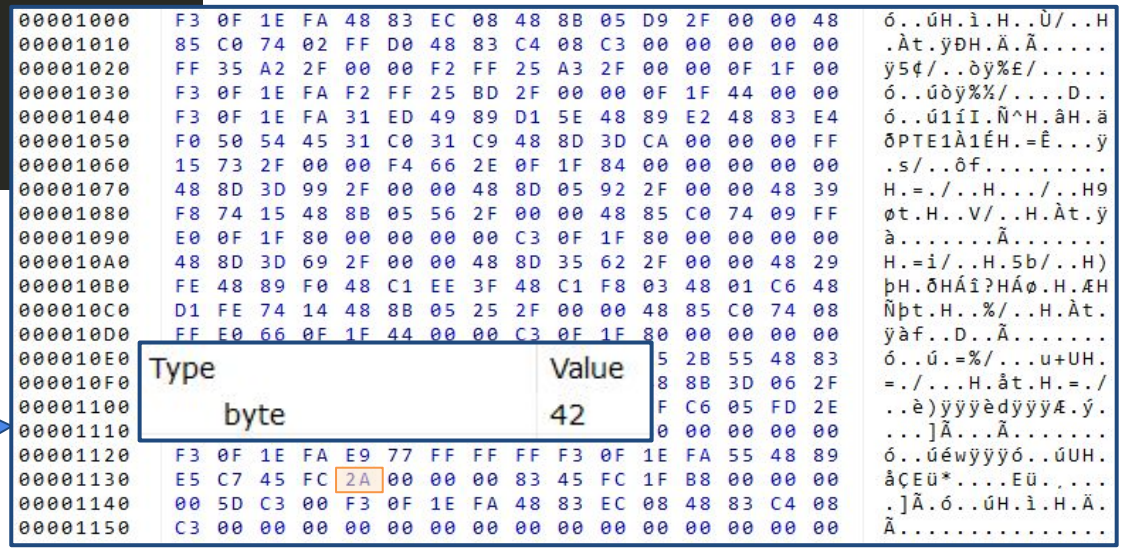
Compiling Programs

We can compile C programs using **gcc** to generate a **binary** executable

```c
#include <stdio.h>

int main() {
  // Create an integer with the initial value of 42
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

```
gcc simple.c -o simple
```

It will translate things into binary!



```
00001000  F3 0F 1E FA 48 83 EC 08 48 8B 05 D9 2F 00 00 48  ó..úH.ì.H..Ù/..H
00001010  85 C0 74 02 FF D0 48 83 C4 08 C3 00 00 00 00 00  .Àt.ÿÐH.Ä.Ã.....
00001020  FF 35 A2 2F 00 00 F2 FF 25 A3 2F 00 00 0F 1F 00  ÿ5¢/..òÿ%£/.....
00001030  F3 0F 1E FA F2 FF 25 BD 2F 00 00 0F 1F 44 00 00  ó..úòÿ%½/....D..
00001040  F3 0F 1E FA 31 ED 49 89 D1 5E 48 89 E2 48 83 E4  ó..ú1íI.Ñ^H.âH.ä
00001050  F0 50 54 45 31 C0 31 C9 48 8D 3D CA 00 00 00 FF  ðPTE1À1ÉH.=Ê...ÿ
00001060  15 73 2F 00 00 F4 66 2E 0F 1F 84 00 00 00 00 00  .s/..ôf.........
00001070  48 8D 3D 99 2F 00 00 48 8D 05 92 2F 00 00 48 39  H.=./..H.../..H9
00001080  F8 74 15 48 8B 05 56 2F 00 00 48 85 C0 74 09 FF  øt.H..V/..H.Àt.ÿ
00001090  E0 0F 1F 80 00 00 00 00 C3 0F 1F 80 00 00 00 00  à.......Ã.......
000010A0  48 8D 3D 69 2F 00 00 48 8D 35 62 2F 00 00 48 29  H.=i/..H.5b/..H)
000010B0  FE 48 89 F0 48 C1 EE 3F 48 C1 F8 03 48 01 C6 48  þH.ðHÁî?HÁø.H.ÆH
000010C0  D1 FE 74 14 48 8B 05 25 2F 00 00 48 85 C0 74 08  Ñþt.H..%/..H.Àt.
000010D0  FF E0 66 0F 1F 44 00 00 C3 0F 1F 80 00 00 00 00  ÿàf..D..Ã.......
000010E0  [illegible]                                      [illegible]
000010F0  [illegible]                                      [illegible]
00001100  [illegible]                                      [illegible]
00001110  [illegible]                                      [illegible]
00001120  F3 0F 1E FA E9 77 FF FF FF F3 0F 1E FA 55 48 89  ó..úéwÿÿÿó..úUH.
00001130  E5 C7 45 FC 2A 00 00 00 83 45 FC 1F B8 00 00 00  åÇEü*....Eü.¸...
00001140  00 5D C3 00 F3 0F 1E FA 48 83 EC 08 48 83 C4 08  .]Ã.ó..úH.ì.H.Ä.
00001150  C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  Ã...............
```

Type	Value
byte	42

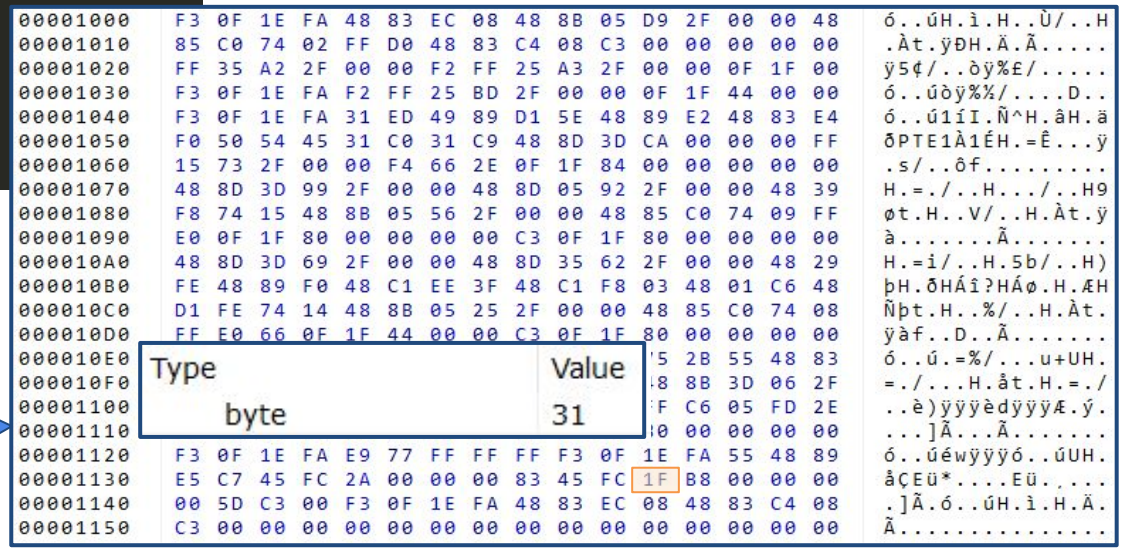
Compiling Programs

We can compile C programs using **gcc** to generate a **binary** executable

```c
#include <stdio.h>

int main() {
  // Create an integer with the initial value of 42
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

```
gcc simple.c -o simple
```

It will translate things into binary!



Type	Value
byte	31

Compiling Programs

We can compile C programs using **gcc** to generate a **binary** executable

```c
#include <stdio.h>

int main() {
  // Create an integer with the initial value of 42
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

```
gcc -nostdlib simple.c -o simple
```

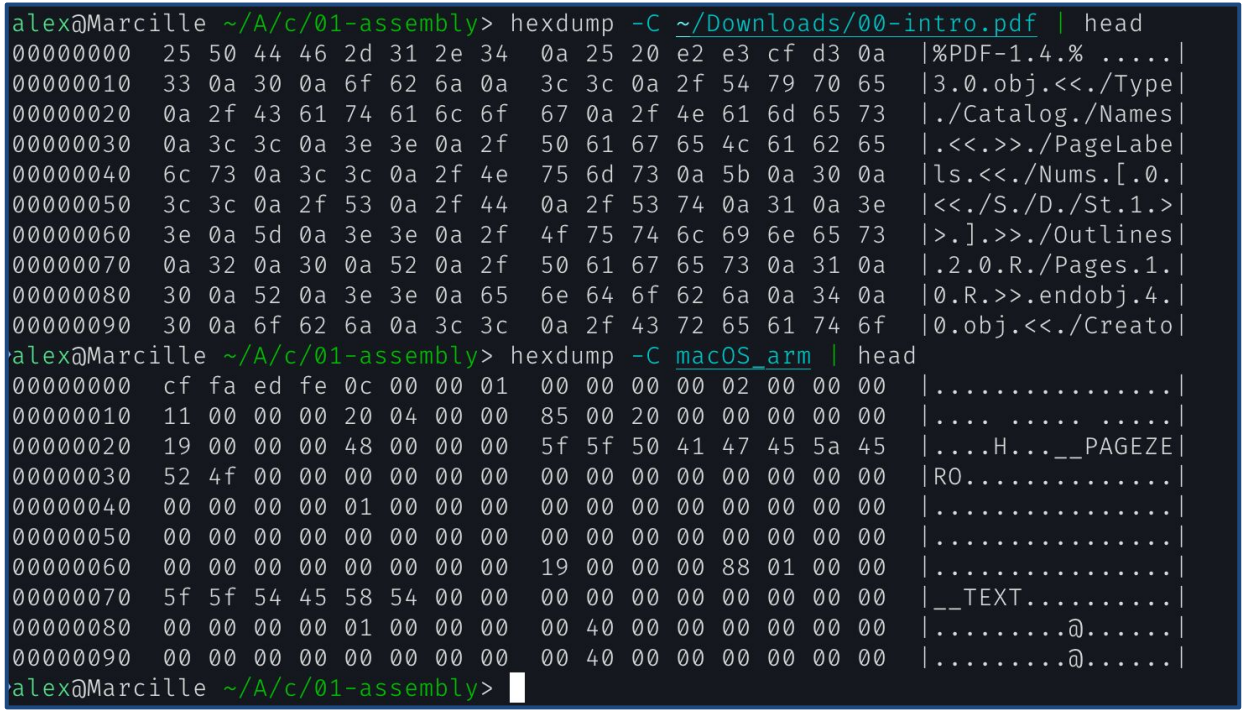
We can also exclude the standard library with -nostdlib to reduce "the code"

```
00001000   F30F 1EFA 5548 89E5 C745 FC2A 0000 0083   ó..úUH.åÇEü*....
00001010   45FC 1FB8 0000 0000 5DC3 0000 0000 0000   Eü.¸....]Ã......
```

Same code, but **only** `simple.c` and nothing else

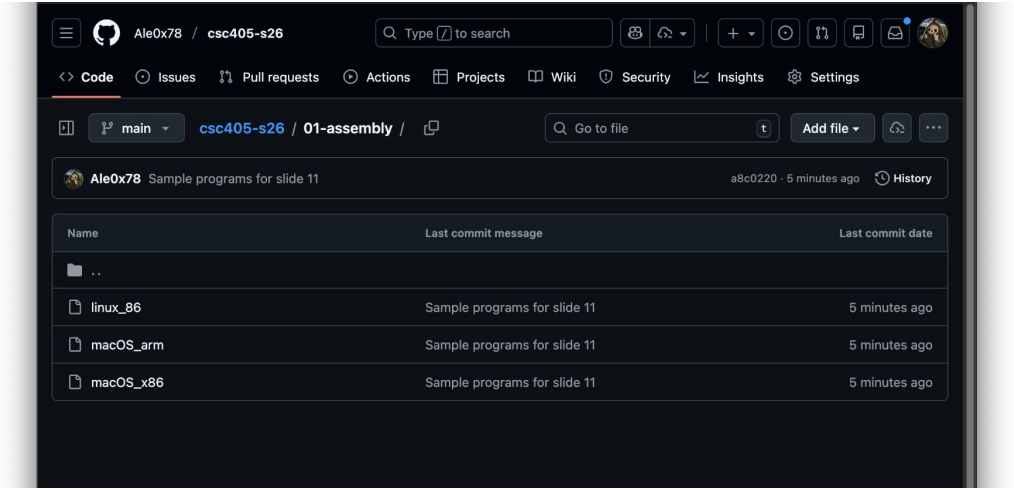
Tools to Become Familiar With

`xxd/hexdump -C <filename>` - Unix tool to viewing any binary file in hex format

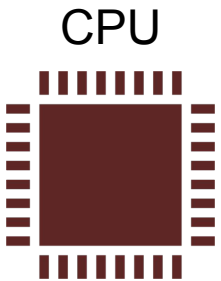


Try it yourself: find the password!

https://go.ncsu.edu/86udx45



The von Neumann Architecture



The CPU is in charge of executing the currently load program's instructions

Executes three primary tasks:

- **Arithmetic Logic Unit** (ALU)
  - Make some calculation
  - Do some comparison
- **Registers**
  - Read/Write values from/to memory
  - Stores values on the CPU rather than pushing to memory for efficiency
- **Control Unit**
  - Conditionally **jump** to execute other instructions

Memory is Slow

When the CPU retrieves contents from memory address **`i`**

- **`i`** travels from the **`CPU`** to **`RAM`**
- **`RAM`**'s logic selects the memory register whose address is **`i`**
- contents of **`RAM[i]`** travels back to the **`CPU`**

Level	Access Time	Typical Size	Technology	Managed By
Registers	`1-3 ns`	`1 KB`	CMOS	Compiler
L1 Cache	`2-8 ns`	`8KB - 128KB`	SRAM	Hardware
L2 Cache	`5-12 ns`	`0.5MB - 8MB`	SRAM	Hardware
Main Memory	`10-60 ns`	`64MB - 1GB`	DRAM	OS
Hard Disk	`0.3-1 ms`	`20GB - 100GB`	Magnetic	OS / User

Registers

Registers provide the same service but without travel and search expenses

This is because the reside inside the CPU and are much more limited in supply (allowing for shorter instructions)

Serves three purposes:

- **Data** - stores values for short term calculations
- **Addressing** - stores memory addresses for various functions
- **Program Counter** - keeps track of the next instruction to be fetched

Registers

Registers provide the same service but without travel and search expenses

This is because the reside inside the CPU and are much more limited in supply (allowing for shorter instructions)

Serves three purposes:

- **Data** - stores values for short term calculations
- **Addressing** - stores memory addresses for various functions
- **Program Counter** - [illegible] on to be fetched

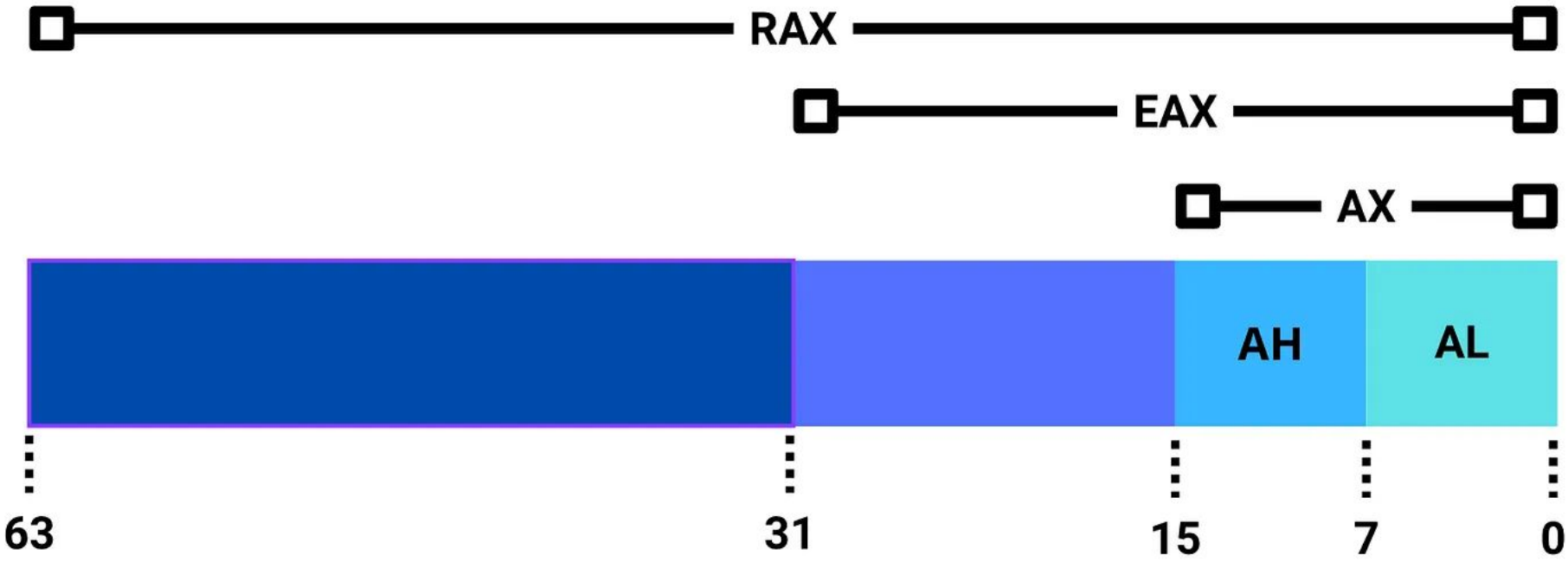
As we'll see next week, this is how we can cause some damage

Registers

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits
%rax	Return value, callee-owned	%eax	%ax	%al
%rdi	1st argument, callee-owned	%edi	%di	%dil
%rsi	2nd argument, callee-owned	%esi	%si	%sil
%rdx	3rd argument, callee-owned	%edx	%dx	%dl
%rcx	4th argument, callee-owned	%ecx	%cx	%cl
%r8	5th argument, callee-owned	%r8d	%r8w	%r8b
%r9	6th argument, callee-owned	%r9d	%r9w	%r9b
%r10	Scratch/temporary, callee-owned	%r10d	%r10w	%r10b
%r11	Scratch/temporary, callee-owned	%r11d	%r11w	%r11b
%rsp	Stack pointer, caller-owned	%esp	%sp	%spl
%rbx	Local variable, caller-owned	%ebx	%bx	%bl
%rbp	Local variable, caller-owned	%ebp	%bp	%bpl
%r12	Local variable, caller-owned	%r12d	%r12w	%r12b
%r13	Local variable, caller-owned	%r13d	%r13w	%r13b
%r14	Local variable, caller-owned	%r14d	%r14w	%r14b
%r15	Local variable, caller-owned	%r15d	%r15w	%r15b
%rip	Instruction pointer			
%eflags	Status/condition code bits			

From the reading

What's up with *RAX, EAX, AX, AL*?



From: The Accumulator register in x86–64 assembly: understanding RAX, EAX, AX, AH, and AL

Machine Code

Machine code can be broken down into two categories: **binary** and **symbolic**

```
C7 45 FC 2A 00 00 00
```

```
00001000    F30F 1EFA 5548 89E5 C745 FC2A 0000 0083    ó..úUH.åÇEü*....
00001010    45FC 1FB8 0000 0000 5DC3 0000 0000 0000    Eü.¸....]Ã......
```

Machine Code

Machine code can be broken down into two categories: **binary** and **symbolic**

```
C7 45 FC 2A 00 00 00
```

"binary"

```
00001000   F30F 1EFA 5548 89E5 C745 FC2A 0000 0083   ó..úUH.åÇEü*....
00001010   45FC 1FB8 0000 0000 5DC3 0000 0000 0000   Eü.¸....]Ã......
```

Machine Code

Machine code can be broken down into two categories:

```
C7 45 FC 2A 00 00 00
```

Instead of
1100 0111 0100 0101 1111
1100 0010 1010 0000 0000
0000 0000 0000 0000,
we commonly condense it down to hexadecimal for "easier reading"

```
00001000    F30F 1EFA 5548 89E5 C745 FC2A 0000 0083    ó..úUH.åÇEü*....
00001010    45FC 1FB8 0000 0000 5DC3 0000 0000 0000    Eü.¸....]Ã......
```

Machine Code

Machine code can be broken down into two categories: **binary** and **symbolic**

```
C7 45 FC 2A 00 00 00
```

```
00001000    F30F 1EFA 5548 89E5 C745 FC2A 0000
00001010    45FC 1FB8 0000 0000 5DC3 0000 0000
```

We can also use a symbolic assembly language that converts these 1's and 0's into something actually readable

```
1    main:
2            pushq   %rbp
3            movq    %rsp, %rbp
4            movl    $42, -4(%rbp)
5            addl    $31, -4(%rbp)
6            movl    $0, %eax
7            popq    %rbp
8            ret
```

Assembly Flavors

There are several Assembly languages, each written for a specific processor

In accordance with the processor's Instruction Set Architecture, or **ISA**

Three Primary Architectures

- **x86 ← We will be working with this one!**
- ARM
- MIPS
- plus **many** more…

x86 Syntax Branches - Intel and AT&T

Intel

- Windows and DOS programs
- Operations follow the format
  `mnemonic destination, source`
- `mov ebx, 42`

AT&T

- Unix programs
- Operations follows the format
  `mnemonic source, destination`
- `mov $42, %ebx`

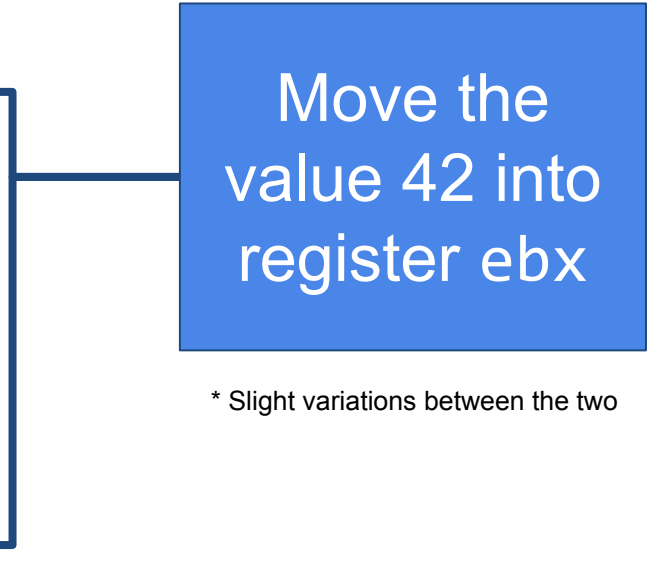
Syntax Branches - Intel and AT&T

Intel

- Windows and DOS programs
- Operations follow the format
  `mnemonic destination, source`
- `mov ebx, 42`

AT&T

- Unix programs
- Operations follows the format
  `mnemonic source, destination`
- `mov $42, %ebx`



* Slight variations between the two

x86 Assembly Syntax - Reserved Keywords

- lds
- les
- lfs
- lgs
- lss
- pop
- push
- in
- ins
- out
- outs
- lahf
- sahf
- popf
- pushf
- cmc
- clc
- stc
- cli
- sti
- cld
- std
- add
- adc
- sub
- sbb
- cmp
- inc
- dec
- test
- sal
- shl
- sar
- shr
- shld
- shrd
- not
- neg
- bound
- and
- or
- xor
- imul
- mul
- div
- idiv
- cbtw
- cwtl
- cwtd
- cltd
- daa
- das
- aaa
- aas
- aam
- aad
- wait
- fwait
- movs
- cmps
- stos
- lods
- scas
- xlat
- rep
- repnz
- repz
- lcall
- call
- ret
- lret
- enter
- leave
- jcxz
- loop
- loopnz
- loopz
- jmp
- ljmp
- int
- into
- iret
- sldt
- str
- lldt
- ltr
- verr
- verw
- sgdt
- sidt
- lgdt
- lidt
- smsw
- lmsw
- lar
- lsl
- clts
- arpl
- bsf
- bsr
- bt
- btc
- btr
- bts
- cmpxchg
- fsin
- fcos
- fsincos
- fld
- fldcw
- fldenv
- fprem
- fucom
- fucomp
- fucompp
- lea
- mov
- movw
- movsx
- movzb
- popa
- pusha
- rcl
- rcr
- rol
- ror
- setcc
- bswap
- xadd
- xchg
- wbinvd
- invd
- invlpg
- lock
- nop
- hlt
- fld
- fst
- fstp
- fxch
- fild
- fist
- fistp
- fbld
- fbstp
- fadd
- faddp
- fiadd
- fsub
- fsubp
- fsubr
- fsubrp
- fisubrp
- fisubr
- fmul
- fmulp
- fimul
- fdiv
- fdivp
- fdivr
- fdivrp
- fidiv
- fidivr
- fsqrt
- fscale
- fprem
- frndint
- fxtract
- fabs
- fchs
- fcom
- fcomp
- fcompp
- ficom
- ficomp
- ftst
- fxam
- fptan
- fpatan
- f2xm1
- fyl2x
- fyl2xp1
- fldl2e
- fldl2t
- fldlg2
- fldln2
- fldpi
- fldz
- finit
- fnint
- fnop
- fsave
- fnsave
- fstew
- fnstew
- fstenv
- fnstenv
- fstsw
- fnstsw
- frstor
- fclex
- fnclex
- fdecstp
- ffree
- fincstp

https://en.wikipedia.org/wiki/X86_instruction_listings

x86 Assembly Syntax - Reserved Keywords

- lds
- les
- lfs
- lgs
- lss
- po
- pu
- in
- ins
- ou
- ou
- la
- sa
- po
- pu
- cr
- clc
- stc
- cli
- sti
- cld
- std
- shrd
- not
- neg
- bound
- and
- or
- aam
- aad
- wait
- fwait
- movs
- cmps
- jcxz
- loop
- loopnz
- loopz
- jmp
- ljmp
- smsw
- lmsw
- lar
- lsl
- clts
- arpl
- fprem
- fucom
- fucomp
- fucompp
- lea
- mov
- wbinvd
- invd
- invlpg
- lock
- nop
- hlt
- fsubp
- fsubr
- fsubrp
- fisubrp
- fisubr
- fmul
- fabs
- fchs
- fcom
- fcomp
- fcompp
- ficom
- fldz
- finit
- fnint
- fnop
- fsave
- fnsave
- ew
- tew
- env
- tenv
- sw
- tsw
- tor
- ex
- lex
- cstp
- e
- cstp

You don't need to memorize them, but be aware they all exist, have corresponding hexadecimal values, and **some** of them will be needed for this class

https://en.wikipedia.org/wiki/X86_instruction_listings

Executing Programs

When a program is executed, various elements of the program are loaded into memory

Information from the program is then loaded from the address space in memory

Three Segments:

`.text` - holds program instructions (read-only)

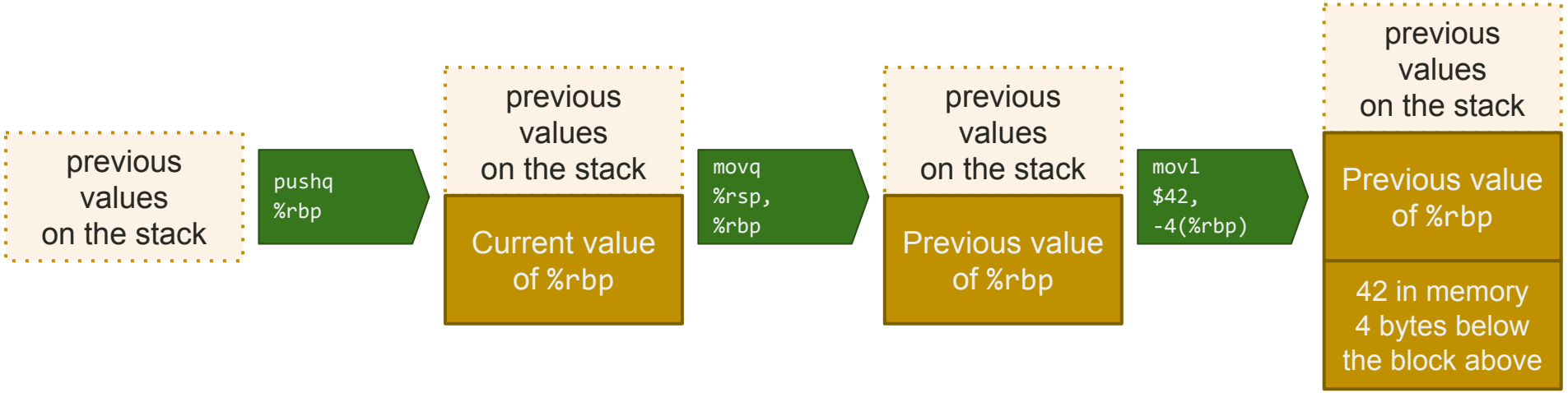
`.bss` - reserved for global variables, contains uninitialized data

`.data` - reserved for global variables, contains initialized data

Stack Machine Model

Arithmetic commands pop their operands from the top of the stack and push their results back to the stack

Since stacks are LIFO (last in first out), a stack pointer (sp) tracks the location just above the topmost element



Programs in Memory

```
↑ Lower Memory Addresses (0x08000000)
  Shared Libraries
  .text
  .bss
  Heap  (grows ↓)
  Stack (grows ↑)
  env pointer
  argc
↓ Higher Memory Addresses (0xbfffffff)
```

Machine Code

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

```
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

These first two instructions serve as the "function prologue"

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

First, we **push** the **base pointer** (**%rbp**) onto the stack for later

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

Next, we **move** (really copy) the **stack pointer** (`%rsp`) to the **base pointer** (`%rbp`)

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

These two instructions establish the **stack frame** of the program

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

Next, we're storing the constant 42 (**$42**) into a memory location

-4(%rbp) is pointing to a memory address that is 4 bytes before **%rbp**

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

Next, add the constant 31 (**$31**) that same memory address

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

C programs need to return a value, so here we are copying the return value (`0`) to a general purpose register (`%eax`)

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

General purpose register (**%eax**)
Register relative to stack (**%rbp**)

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

We **pop** the **base pointer** (**%rbp**) off the stack to return it to its original value

```c
int main() {
  // Create an integer with
  int num = 42;

  // Add 31 to the integer
  num += 31;

  return 0;
}
```

Machine Code

Let's break down the machine code of simple.c

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         movl    $42, -4(%rbp)
5         addl    $31, -4(%rbp)
6         movl    $0, %eax
7         popq    %rbp
8         ret
```

Finally, we **return** from the function, where the return value (**0**) is expected to be stored in **%eax**

Let's read some assembly

```
main:
  push $1
  push $2
  push $3
  push $4
  pop %rax
```

- *push: pushes the values onto the stack*
- *pop: pops the last value off of the stack into the given register*

What is RAX?

Let's emulate it

Let's read some assembly

```
.global main
.section      .text

main:
  movq %rsp, %rbp
  push $1
  push $2
  movq %rbp, %rax
  movq %rsp, %rcx
  sub    %rcx, %rax
  ret
```

- *push: pushes the values onto the stack*
- *pop: pops the last value off of the stack into the given register*
- *mov(q): copies the values from src→dest. The (q) means 64-bit values*
- *sub: dest = dest - src*

What is RAX?

Step-by-step

- Let's step through it with ASM-Visualizer
- What if this was a 32-bit system?
  - You can do this in GDB!

```
Breakpoint 2, 0x0804900c in _start ()
(gdb) info registers
eax            0x8                 8
ecx            0xffffa6f8          -22792
edx            0x0                 0
ebx            0x0                 0
esp            0xffffa6f8          0xffffa6f8
ebp            0xffffa700          0xffffa700
esi            0x0                 0
edi            0x0                 0
eip            0x804900c           0x804900c <_start+12>
eflags         0x212               [ AF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x0                 0
```

Assemble the file

Breakpoint at _start

There is our code!

Run until the ret instruction

```
alex@mithrun ~/A/tmp> cc -m32 -static -nostdlib sample.s -o a.out
alex@mithrun ~/A/tmp> gdb -q a.out
Reading symbols from a.out...
(No debugging symbols found in a.out)
(gdb) b _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/alex/Ambrosia/tmp/a.out

Breakpoint 1, 0x08049000 in _start ()
(gdb) disassemble
Dump of assembler code for function _start:
=> 0x08049000 <+0>:     mov    %esp,%ebp
   0x08049002 <+2>:     push   $0x1
   0x08049004 <+4>:     push   $0x2
   0x08049006 <+6>:     mov    %ebp,%eax
   0x08049008 <+8>:     mov    %esp,%ecx
   0x0804900a <+10>:    sub    %ecx,%eax
   0x0804900c <+12>:    ret
End of assembler dump.
(gdb) b *0x0804900c
Breakpoint 2 at 0x804900c
(gdb) c
Continuing.

Breakpoint 2, 0x0804900c in _start ()
(gdb) info registers
eax            0x8                 8
ecx            0xffffa6f8          -22792
edx            0x0                 0
ebx            0x0                 0
esp            0xffffa6f8          0xffffa6f8
ebp            0xffffa700          0xffffa700
esi            0x0                 0
edi            0x0                 0
eip            0x804900c           0x804900c <_start+12>
eflags         0x212               [ AF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x0                 0
```

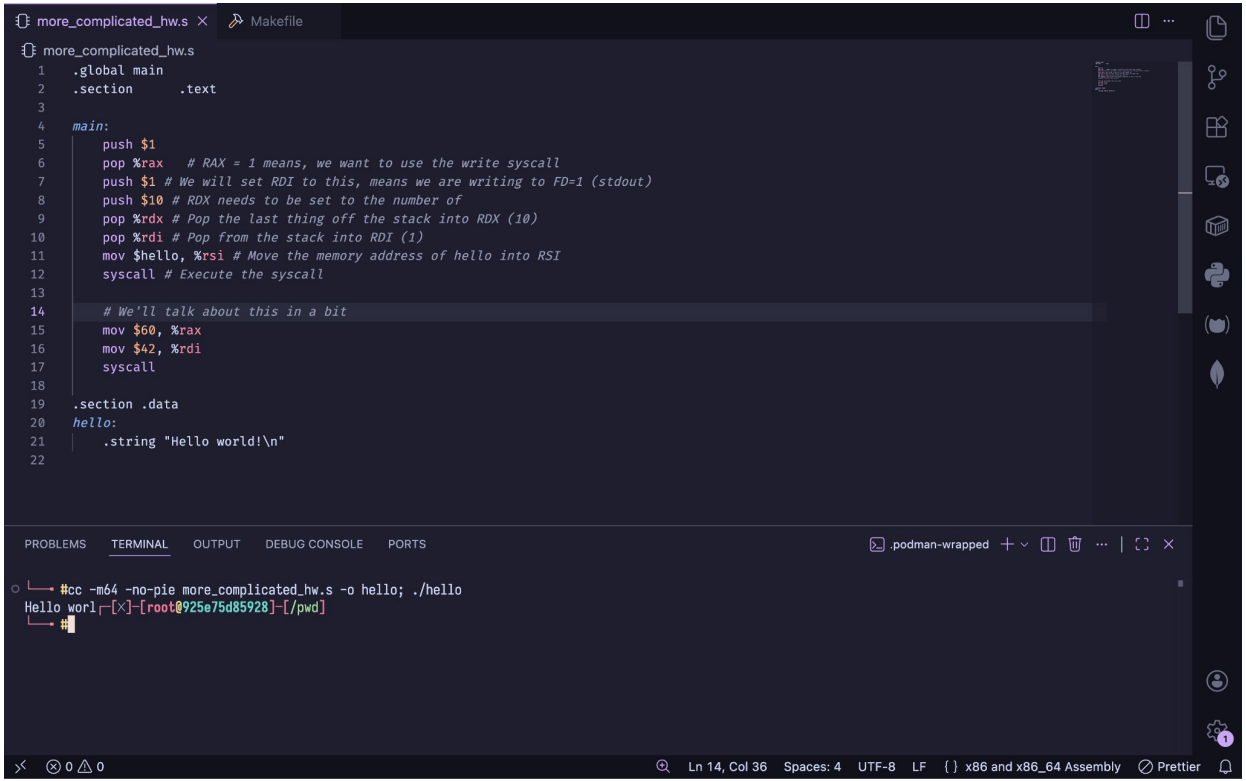
A more complicated than needed Hello World

more_complicated_hw.s × Makefile

more_complicated_hw.s

```
1   .global main
2   .section      .text
3
4   main:
5       push $1
6       pop %rax   # RAX = 1 means, we want to use the write syscall
7       push $1 # We will set RDI to this, means we are writing to FD=1 (stdout)
8       push $10 # RDX needs to be set to the number of
9       pop %rdx # Pop the last thing off the stack into RDX (10)
10      pop %rdi # Pop from the stack into RDI (1)
11      mov $hello, %rsi # Move the memory address of hello into RSI
12      syscall # Execute the syscall
13
14      # We'll talk about this in a bit
15      mov $60, %rax
16      mov $42, %rdi
17      syscall
18
19  .section .data
20  hello:
21      .string "Hello world!\n"
```

Did you spot the bug?



Let's step through it

- ASM Visualizer
- How you know how to set the registers? Use a syscall table

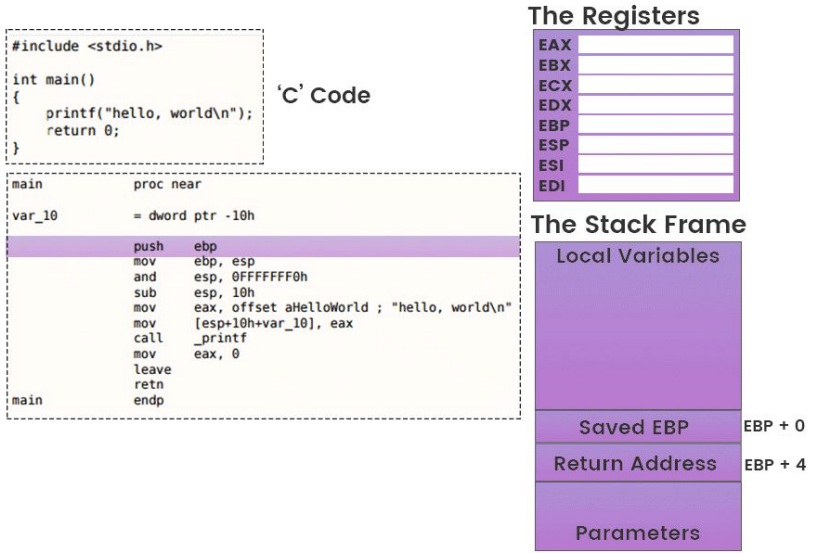
```
# What does this do?
mov $60, %rax
mov $42, %rdi
syscall
```

x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
59	execve	man/ cs/	0x3b	const char *filename	const char *const *argv	const char *const *envp	-	-	-
60	exit	man/ cs/	0x3c	int error_code	-	-	-	-	-
61	wait4	man/ cs/	0x3d	pid_t pid	int *stat_addr	int options	struct rusage *ru	-	-
62	kill	man/ cs/	0x3e	pid_t pid	int sig	-	-	-	-
63	uname	man/ cs/	0x3f	struct old_utsname *	-	-	-	-	-
64	semget	man/ cs/	0x40	key_t key	int nsems	int semflg	-	-	-

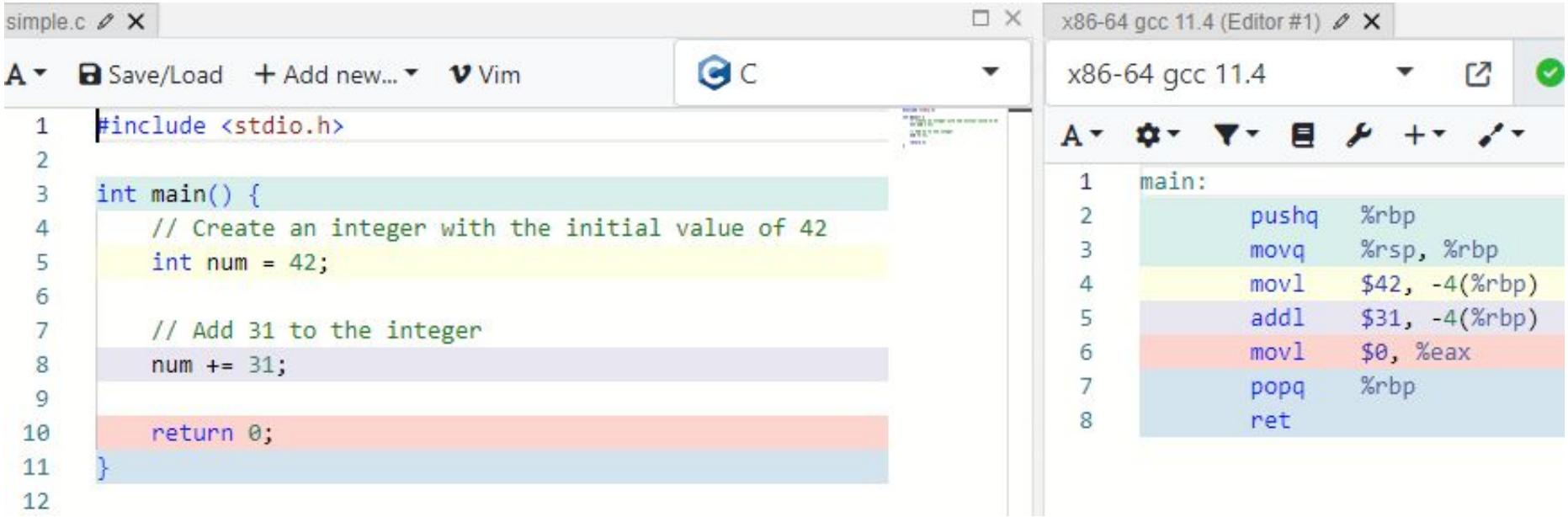
Let's put this all together!



Source: *MalwareUnicorn's Blog*

Tools to Become Familiar With

godbolt.org - You can use this site to browser the machine code for any program



Tools to Become Familiar With

`objdump -zd <binary>` - Linux tool for producing the same results locally

```
0000000000001000 <main>:
    1000:       f3 0f 1e fa             endbr64
    1004:       55                      push   %rbp
    1005:       48 89 e5                mov    %rsp,%rbp
    1008:       c7 45 fc 2a 00 00 00    movl   $0x2a,-0x4(%rbp)
    100f:       83 45 fc 1f             addl   $0x1f,-0x4(%rbp)
    1013:       b8 00 00 00 00          mov    $0x0,%eax
    1018:       5d                      pop    %rbp
    1019:       c3                      ret
```

Wait… where is the hello world?

- We are missing an important part for that, ***syscalls***!

```
# What does this do?
mov $60, %rax
mov $42, %rdi
syscall
```

Setup the registers, depending on what you want to do

Wait… where is the hello world?

- We are missing an important part for that, ***syscalls***!

```
# What does this do?
mov $60, %rax
mov $42, %rdi
syscall
```

Say the magic word!

Wait… where is the hello world?

- We are missing an important part for that, ***syscalls***!
- No need to memorize, just look at a [lookup table]

```
# What does this do?
mov $60, %rax
mov $42, %rdi
syscall
```

Syscall ID

x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
59	execve	man/ cs/	0x3b	const char *filename	const char *const *argv	const char *const *envp	-	-	-
60	exit	man/ cs/	0x3c	int error_code	-	-	-	-	-
61	wait4	man/ cs/	0x3d	pid_t pid	int *stat_addr	int options	struct rusage *ru	-	-
62	kill	man/ cs/	0x3e	pid_t pid	int sig	-	-	-	-
63	uname	man/ cs/	0x3f	struct old_utsname *	-	-	-	-	-
64	semget	man/ cs/	0x40	key_t key	int nsems	int semflg	-	-	-

Next week: Hello world!

```
.section .text
    .global _start

_start:
    movq $0×0a6f6c6c6548, %rax # Hello\n → 48 65 6C 6C 6F A0
    pushq %rax # Push to the stack

    movq $1, %rax
    movq $1, %rdi
    movq %rsp, %rsi
    movq $6, %rdx
    syscall

    movq $60, %rax
    xorq %rdi, %rdi
    syscall
```

x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-

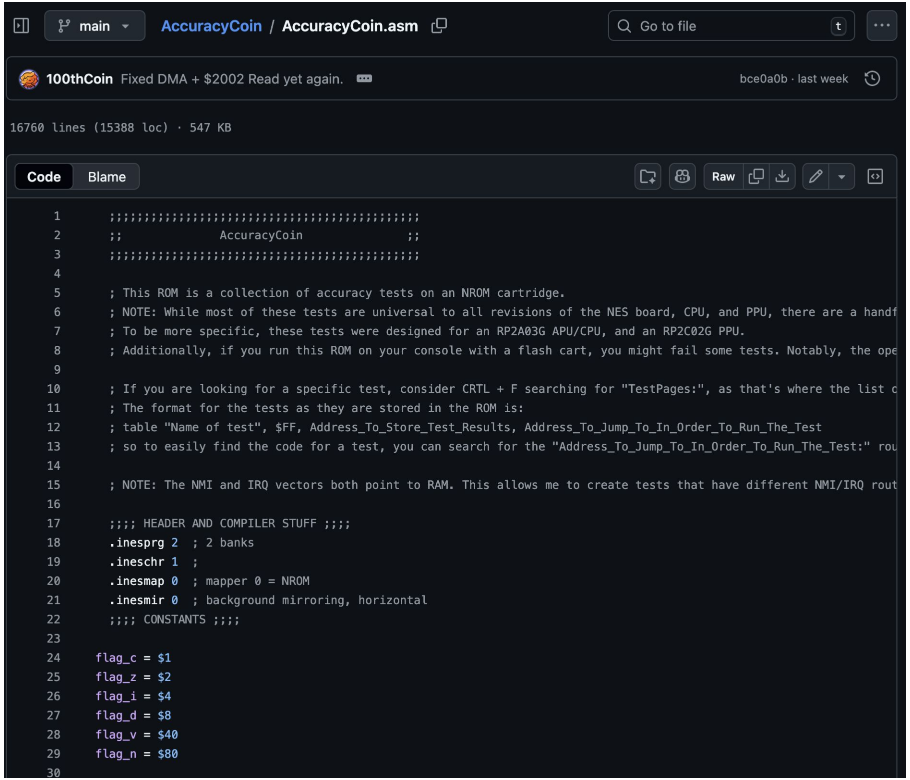
In-class practice

more_complicated_hw.s | Makefile

more_complicated_hw.s

```
1   .global main
2   .section      .text
3
4   main:
5       push $1
6       pop %rax   # RAX = 1 means, we want to use the write syscall
7       push $1 # We will set RDI to this, means we are writing to FD=1 (stdout)
8       push $10 # RDX needs to be set to the number of
9       pop %rdx # Pop the last thing off the stack into RDX (10)
10      pop %rdi # Pop from the stack into RDI (1)
11      mov $hello, %rsi # Move the memory address of hello into RSI
12      syscall # Execute the syscall
13
14      # We'll talk about this in a bit
15      mov $60, %rax
16      mov $42, %rdi
17      syscall
18
19  .section .data
20  hello:
21      .string "Hello world!\n"
```

How Inaccurate are Nintendo's Official Emulators?




```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;              AccuracyCoin               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; This ROM is a collection of accuracy tests on an NROM cartridge.
; NOTE: While most of these tests are universal to all revisions of the NES board, CPU, and PPU, there are a handf
; To be more specific, these tests were designed for an RP2A03G APU/CPU, and an RP2C02G PPU.
; Additionally, if you run this ROM on your console with a flash cart, you might fail some tests. Notably, the ope

; If you are looking for a specific test, consider CRTL + F searching for "TestPages:", as that's where the list c
; The format for the tests as they are stored in the ROM is:
; table "Name of test", $FF, Address_To_Store_Test_Results, Address_To_Jump_To_In_Order_To_Run_The_Test
; so to easily find the code for a test, you can search for the "Address_To_Jump_To_In_Order_To_Run_The_Test:" rou

; NOTE: The NMI and IRQ vectors both point to RAM. This allows me to create tests that have different NMI/IRQ rout

;;;; HEADER AND COMPILER STUFF ;;;;
.inesprg 2  ; 2 banks
.ineschr 1  ;
.inesmap 0  ; mapper 0 = NROM
.inesmir 0  ; background mirroring, horizontal
;;;; CONSTANTS ;;;;

flag_c = $1
flag_z = $2
flag_i = $4
flag_d = $8
flag_v = $40
flag_n = $80
```