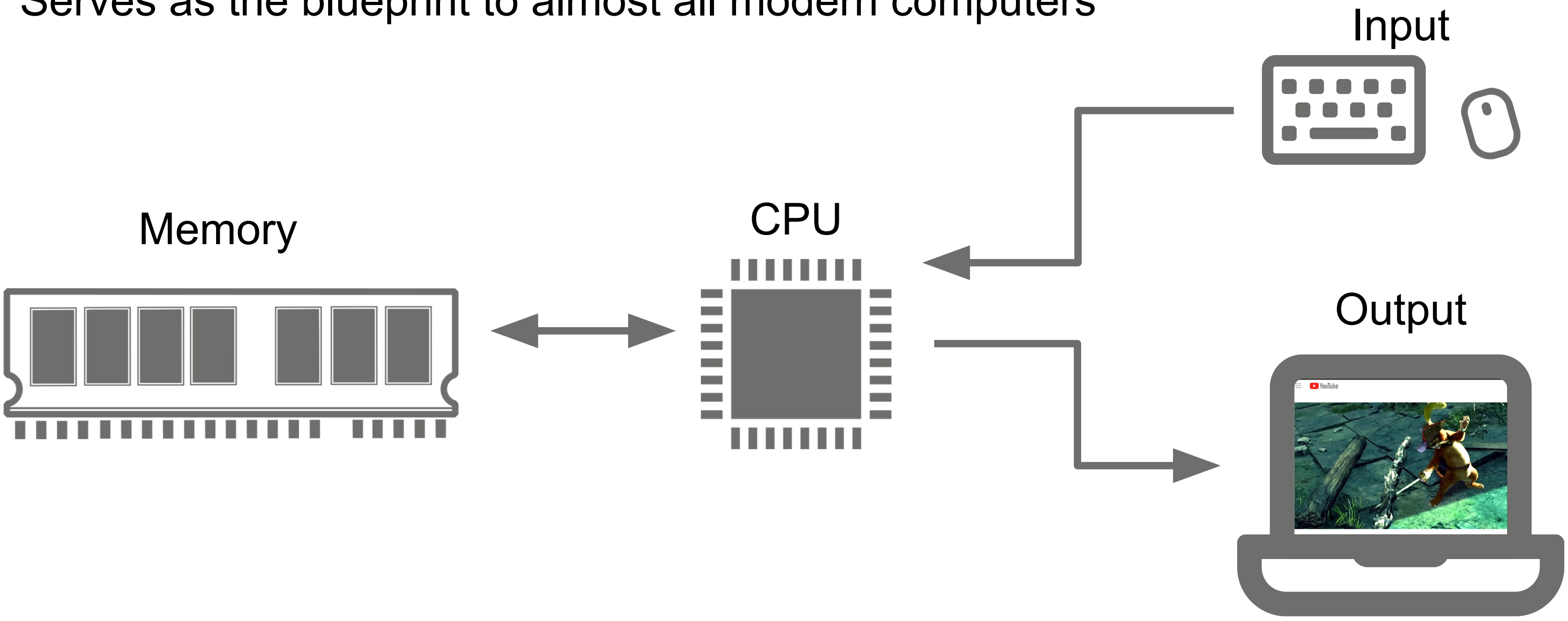
NC STATE UNIVERSITY

CSC 405
Assembly

Alexandros Kapravelos
akaprav@ncsu.edu

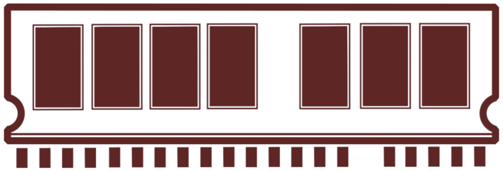
The von Neumann Architecture

Serves as the blueprint to almost all modern computers



The von Neumann Architecture

Memory



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] 5 2B 55 48 83   ó..ú.=%/...u+UH.
000010F0   [illegible] 8 8B 3D 06 2F   =./...H.åt.H.=./
00001100   [illegible] F C6 05 FD 2E   ..è)ÿÿÿèdÿÿÿÆ.ý.
00001110   [illegible] 0 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...............
```

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!

```
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                                      2B 55 48 83  ó..ú.=%/...u+UH.
000010F0                                      8B 3D 06 2F  =./...H.åt.H.=./
00001100                                      C6 05 FD 2E  ..è)ÿÿÿèdÿÿÿÆ.ý.
00001110                                      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...............
```

Type	Value
byte	31

Compiling Programs

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

```
#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"

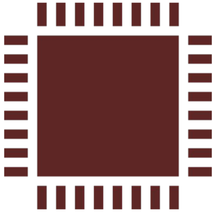
00001000	F30F	1EFA	5548	89E5	C745	FC2A	0000	0083	ó..úUH.åÇEü*....
00001010	45FC	1FB8	0000	0000	5DC3	0000	0000	0000	Eü.,....]Ã......

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

The von Neumann Architecture

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

CPU



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** … on to be fetched

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

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**

"binary"

```
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:

`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**

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

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

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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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
- ARM
- MIPS
- plus **many** more…

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

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

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`

Move the value 42 into register ebx

* Slight variations between the two

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)

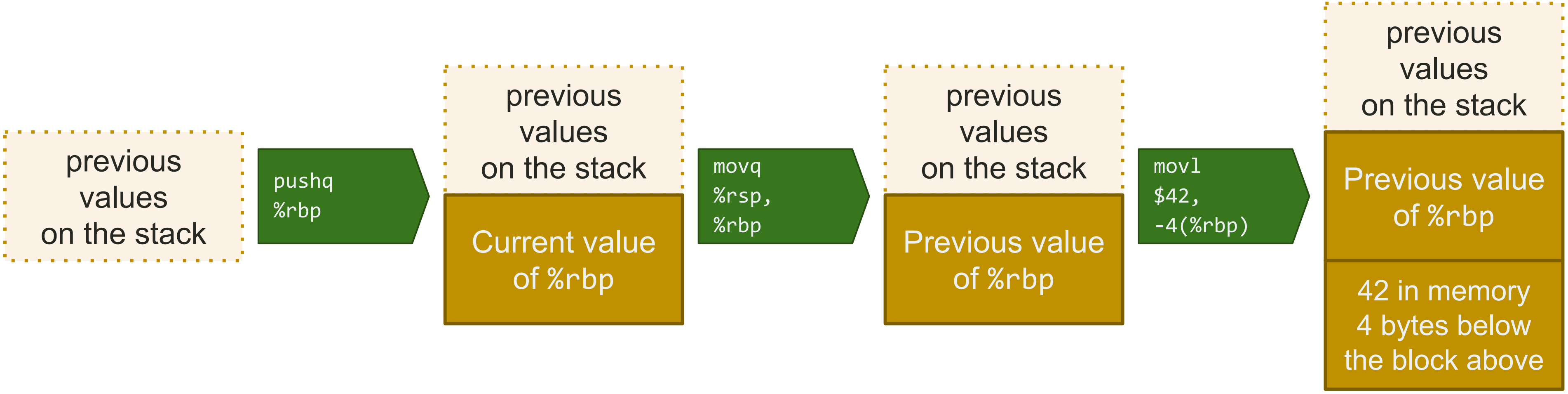
`.bss` - reserved for global variables, contains uninitialized data

`.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)
```

```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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        ret
```

```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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        ret
```

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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        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

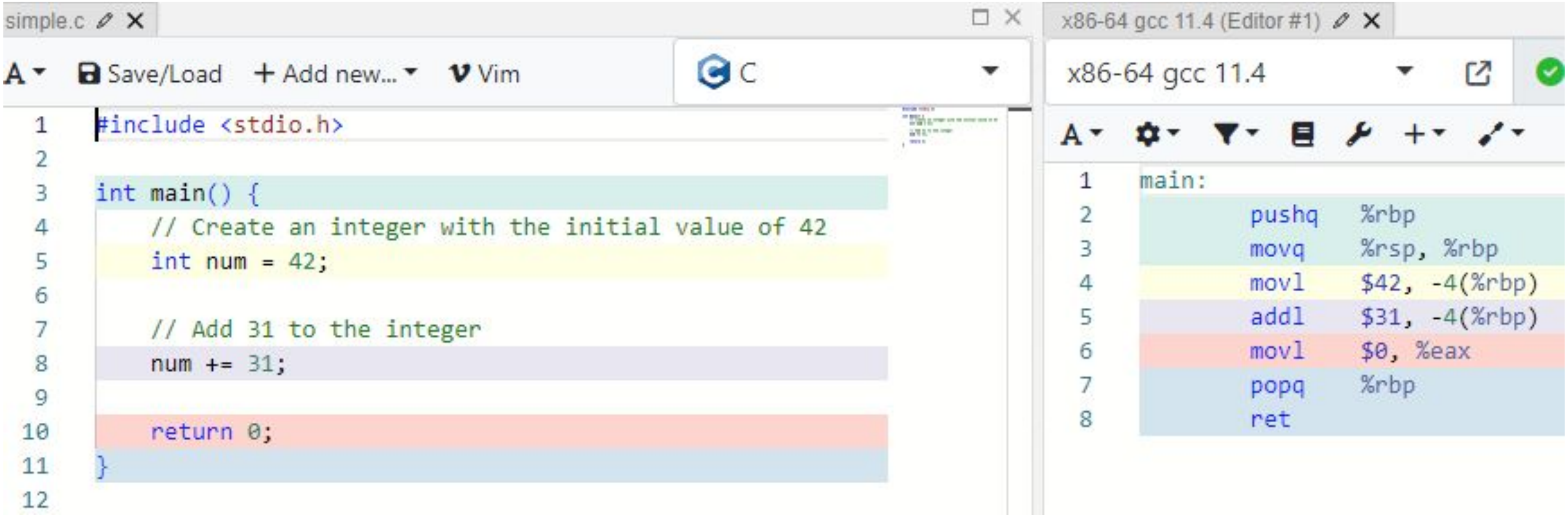
Let's break down the machine code of simple.c

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $42, -4(%rbp)
        addl    $31, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        ret
```

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

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
```

Security Zen - World's First MIDI Shellcode

https://psi3.ru/blog/swl01u/