COMP 520 - Compilers

Lecture 12 – Runtime Organization + Hardware

Grades on Canvas

- I have added your grades to canvas. If something is missing or incorrect, let me know.

- If you turned in an assignment late, it is likely incorrect on canvas and I will fix it as soon as I can.

Today's Goals

- Be convinced that assembly, hardware, and the CPU in general is not magic

Today's Goals

- Be convinced that assembly, hardware, and the CPU in general is not magic

- First: Simple hardware organization (COMP-541)
- Second: Modern CPU organization (COMP-311)
- Lastly: x86 assembly

Goals

- Be convinced that assembly, hardware, and the CPU in general is not magic

Not tested in this class, covered for your benefit

- First: Simple hardware organization (COMP-541/311)
- Second: Modern CPU organization (COMP-311)
- Lastly: x86 assembly

Goals

- Be convinced that assembly, hardware, and the CPU in general is not magic

- First: Simple hardware organization (COMP-541/311)
- Second: Modern CPU organization (COMP-311)
- Lastly: x86 assembly

This will be VERY useful when working on PA4, covered for your benefit

Goals

- Be convinced that assembly, hardware, and the CPU in general is not magic

- First: Simple hardware organization (COMP-541/311)
- Second: Modern CPU organization (COMP-311)
- Lastly: x86 assembly

This is PA4

covered for your benefit

Demystifying Hardware

An extended commercial for COMP-541

Some coverage of the ALU

- In the interest of time, will only cover some parts of the ALU (Arithmetic Logic Unit)
- Specifically, let's cover the adder (where a subtracter is just a small, clever modification)

Not Tested Material

What is 2's complement?

Not Tested Material

What is 2's complement?

Bits	Unsigned value	Signed value (Two's complement)
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	−128
1000 0001	129	−127
1000 0010	130	−126
1111 1110	254	−2
1111 1111	255	−1

Not Tested Material

Motivation: Example Goal

- Add the following:

```
    0000  0101
+   0000  1101
-------------
```

Not Tested Material

Let's first focus on single bit addition

0 + 0 0 + 1 1 + 0 1 + 1

Not Tested Material

Let's first focus on single bit addition

0 + 0	0 + 1	1 + 0	1 + 1
0	**1**		

Not Tested Material

Let's first focus on single bit addition

0 + 0	0 + 1	1 + 0	1 + 1
0	**1**	**1**	**10?**

Not Tested Material

1+1 Gets an entire slide!

- Let's rewrite this:

1+1 = 0 (**carry 1**)

Not Tested Material

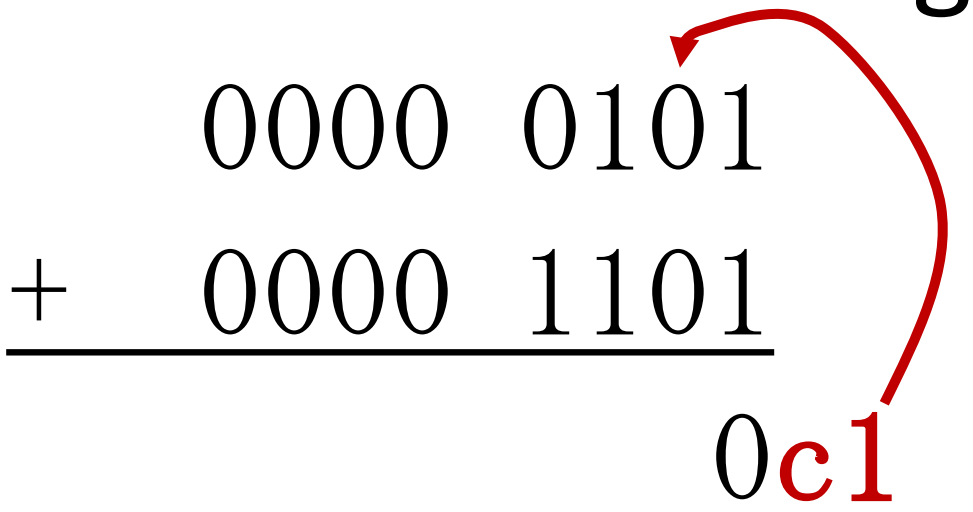
Let's first focus on single bit addition

0 + 0	0 + 1	1 + 0	1 + 1
0 c0	**1 c0**	**1 c0**	**0 c1**

Not Tested Material

Motivation: Example Goal

- Add the following:



Not Tested Material

Let's first focus on single bit addition

0 + 0 + c1

1 c0

0 + 1 + c1

1 + 1 + c1

Not Tested Material

Let's first focus on single bit addition

0 + 0 + c1

1 c0

0 + 1 + c1

0 c1

1 + 1 + c1

1 c1

Not Tested Material

Motivation: Example Goal

- Add the following:

		c1 c1 c1		
	0000	0101		5
+	0000	1101		+13
	0001	0010		18

Not Tested Material

Can we come up with something more formal? What does this look like in hardware?

Truth Tables (COMP-283)

A	0	0	1	1	0	0	1	1
B	0	1	0	1	0	1	0	1
C_{in}	0	0	0	0	1	1	1	1
S	0	1	1	0	1	0	0	1
C_{out}	0	0	0	1	0	1	1	1

Not Tested Material

Truth Tables (COMP-283)

A	0	0	1	1	0	0	1	1
B	0	1	0	1	0	1	0	1
C_{in}	0	0	0	0	1	1	1	1
S	0	1	1	0	1	0	0	1
C_{out}	0	0	0	1	0	1	1	1

If odd number of 1s: S=1, otherwise S=0

What is the boolean logic look like?

Not Tested Material

Truth Tables (COMP-283)

A	0	0	1	1	0	0	1	1
B	0	1	0	1	0	1	0	1
C_{in}	0	0	0	0	1	1	1	1
S	0	1	1	0	1	0	0	1
C_{out}	0	0	0	1	0	1	1	1

$$S = C_{in} \oplus A \oplus B$$

Where $\oplus \equiv$ exclusive or

Not Tested Material

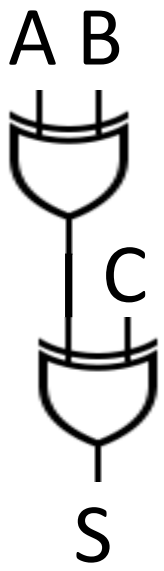
Adder Boolean Algebra

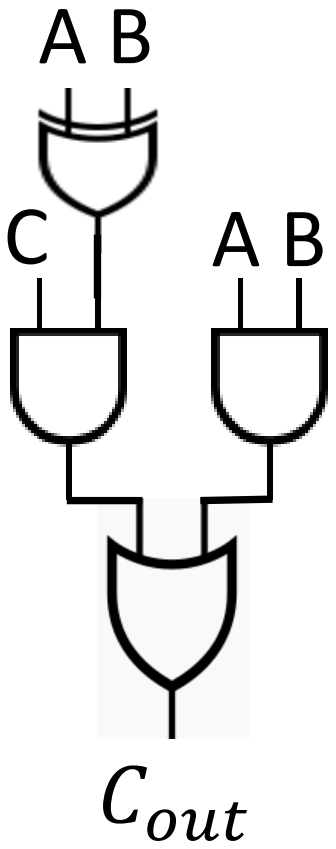
- $S = C_{in} \oplus A \oplus B$
- $C_{out} = \left(C_{in} \wedge (\mathrm{A} \oplus B)\right) \vee (\mathrm{A} \wedge \mathrm{B})$

- **And what can we do with boolean algebra?**

Not Tested Material

Single Bit Adder

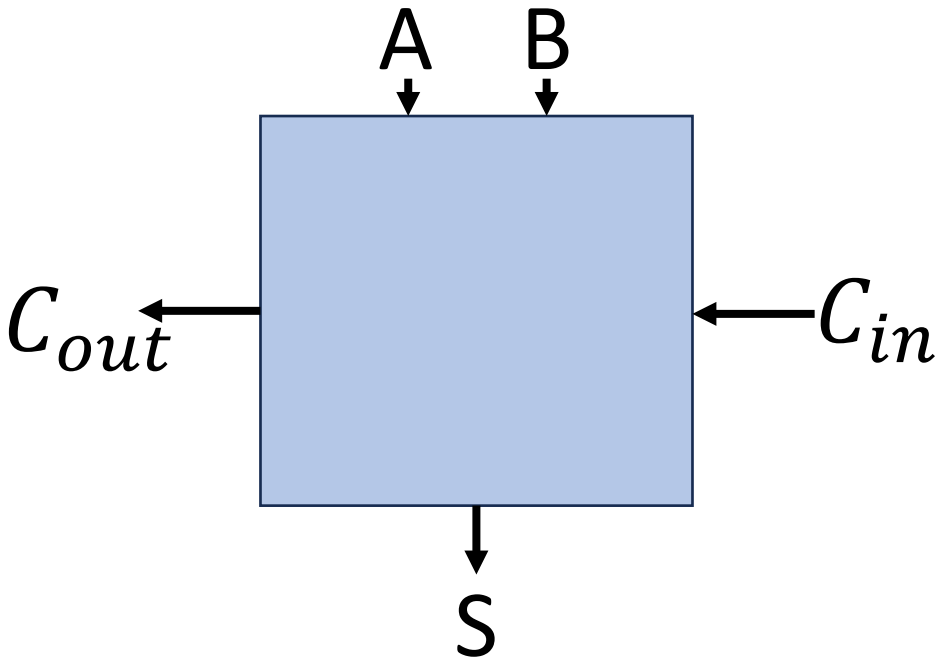




C_{out}

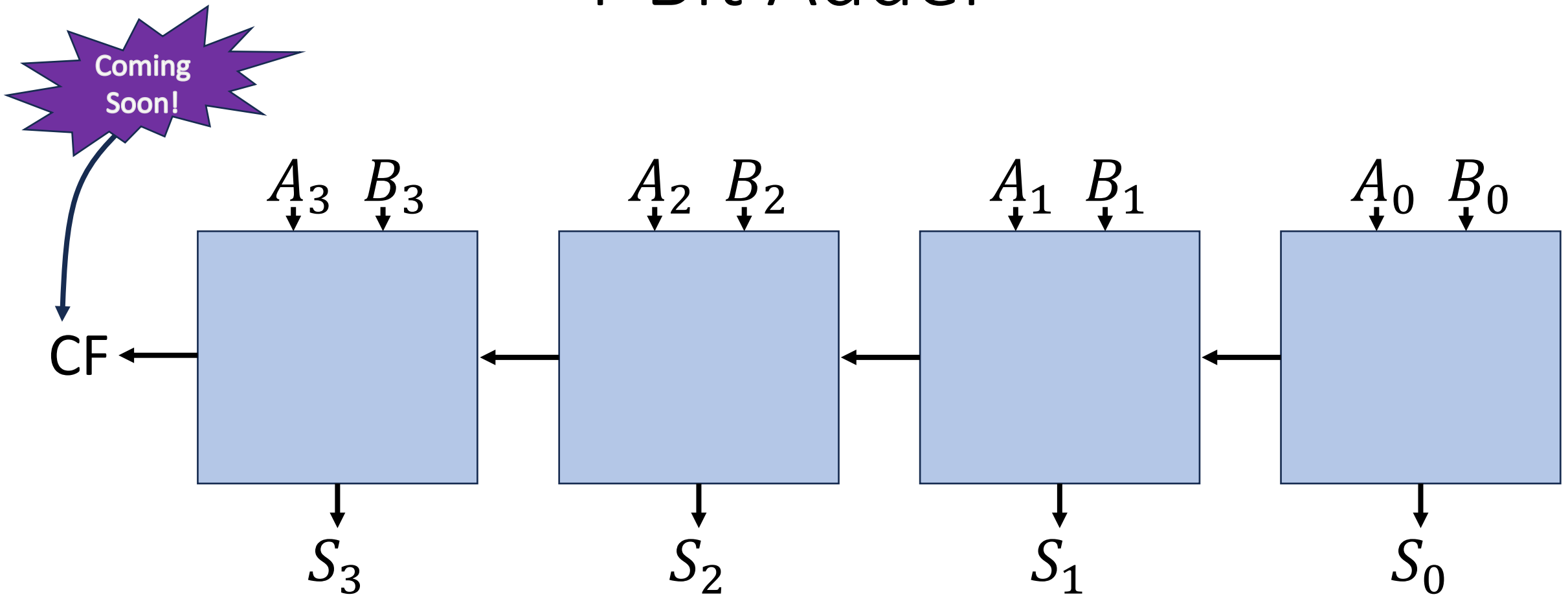
Not Tested Material

1-Bit Adder



Not Tested Material

4-Bit Adder



Not Tested Material

Adder Input/Output

A=0000	0101	5
+B=0000	1101	+13
S=0001	0010	18

Where 18= $S_0 + 2S_1 + 4S_2 + \cdots$

Aka, $S = \sum_i S_i \cdot 2^i$

Not Tested Material

Thus, Hardware ***IS NOT*** Magic

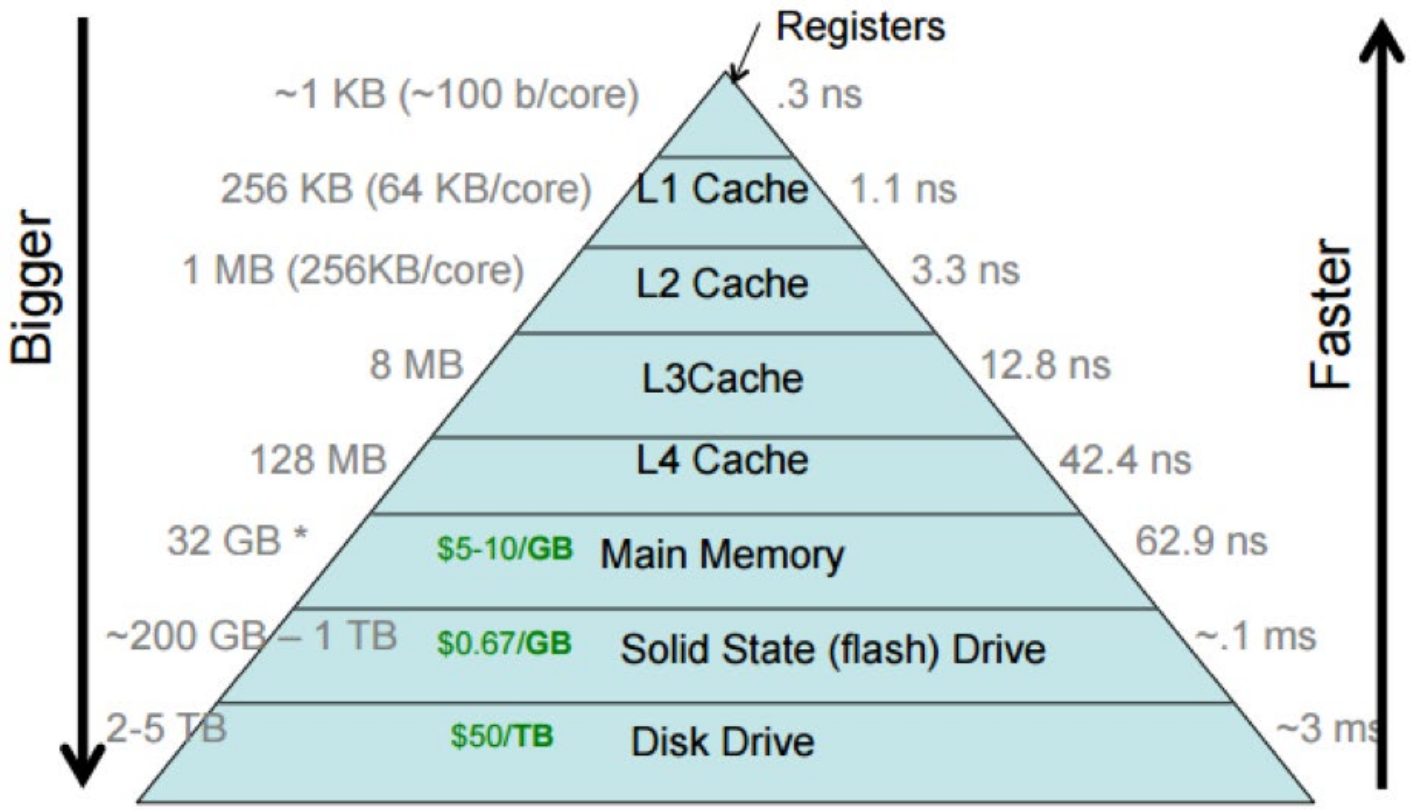
- Addition is not magic,
  it is simple simple circuitry
- But it would be a stretch to say that
  "the rest of the hardware follows similarly"

- If you are curious, see COMP-541

Not Tested Material

CPU Organization

Cache Hierarchy



L1 Cache: Paper on Desk

L2 Cache: Go to bookshelf

L3: Go to local library

RAM: It's in another library in North Carolina

Storage access time

1) Head to the North Atlantic Ocean
2) Sail to London
3) Go to Oxford
4) Wait in line for the Oxford English dictionary, make copies of what you need
5) Sail back

Thankfully, Memory is Memory

- From the perspective of the machine code, accessing memory looks the same (not counting page faults, for that, see COMP-530 and 630 to implement it)

- This means we don't consider "*where*" memory is in cache to be relevant when writing machine code
- All handled by the hardware to be nearly invisible

So what types of memory DO we worry about?

- Stack
- Heap

- Segments:
  - .text (Executable section)
  - .bss (Uninitialized static data)
  - .data (Initialized static data)

We will discuss .idata and other segments later

Data Segment

- Consider static elements such as:

```
class A {
    private static int a = 0;
    private static String b =  “Hello World” ;
}
```

Variable data known before runtime

```
private static int a = 0;
private static String b = “Hello World” ;
```

So instead of having to do an initialization step:

“Store 0 at data 0x5000”,
“Store ‘Hello World\0’ at data 0x5004”

Variable data known before runtime

```
private static int a = 0;
private static String b = “Hello World”;
```

So instead of having to do an initialization phase:

“Store 0 at data 0x5000”,
“Store ‘Hello World\0’ at data 0x5004”

Just store the raw data directly!

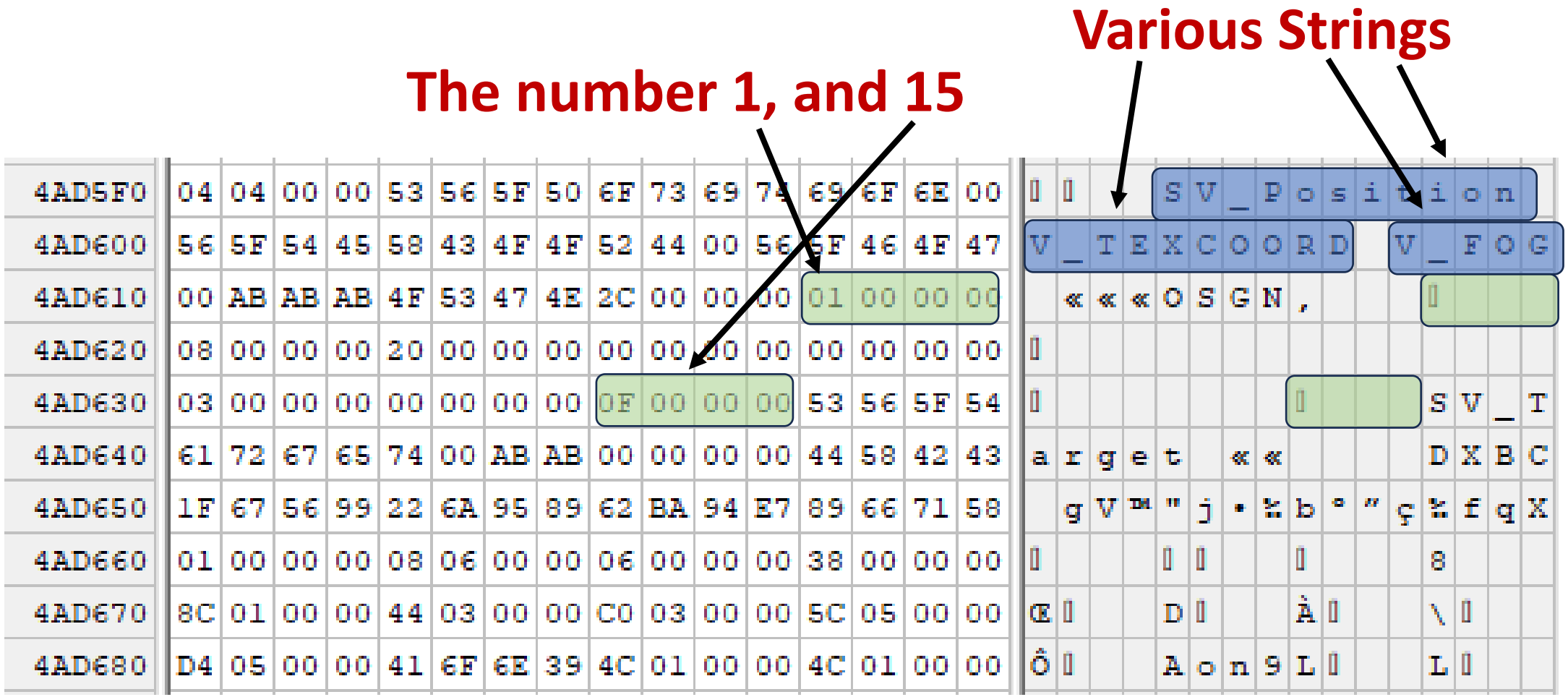
XVI32 Example

If you open an executable file in notepad…

4AD5F0	04	04	00	00	53	56	5F	50	6F	73	69	74	69	6F	6E	00
4AD600	56	5F	54	45	58	43	4F	4F	52	44	00	56	5F	46	4F	47
4AD610	00	AB	AB	AB	4F	53	47	4E	2C	00	00	00	01	00	00	00
4AD620	08	00	00	00	20	00	00	00	00	00	00	00	00	00	00	00
4AD630	03	00	00	00	00	00	00	00	0F	00	00	00	53	56	5F	54
4AD640	61	72	67	65	74	00	AB	AB	00	00	00	00	44	58	42	43
4AD650	1F	67	56	99	22	6A	95	89	62	BA	94	E7	89	66	71	58
4AD660	01	00	00	00	08	06	00	00	06	00	00	00	38	00	00	00
4AD670	8C	01	00	00	44	03	00	00	C0	03	00	00	5C	05	00	00
4AD680	D4	05	00	00	41	6F	6E	39	4C	01	00	00	4C	01	00	00

▯	▯			S	V	_	P	o	s	i	t	i	o	n	
V	_	T	E	X	C	O	O	R	D		V	_	F	O	G
	«	«	«	O	S	G	N	,				▯			
▯															
▯								▯				S	V	_	T
a	r	g	e	t		«	«					D	X	B	C
	g	V	™	"	j	•	‰	b	°	″	ç	‰	f	q	X
▯				▯	▯			▯				8			
Œ	▯			D	▯			À	▯			\	▯		
Ô	▯			A	o	n	9	L	▯			L	▯		

XVI32 Example



Idea: Take chunk of data and load it in memory

- Chunk of data (".data") in executable:

`int a = 4; int b = 5; String c = "Hello";`

And when our program is running, it sees this data in the same ordered form, s.t.,

MemPos+0x00 = `int a` MemPos+0x08= `String c`

MemPos+0x04 = `int b`

Uninitialized Memory (.bss)

- Static memory that is initialized to zero
- Described by a single number usually

Uninitialized Memory (.bss)

- Static memory that is initialized to zero
- Described by a single number usually
- "I need 4096 bytes of data initialized to zero"

```
static int a; static int b; static String c;
```

None initialized. Respectively, positions:
.bss+0, .bss+4, .bss+8 will contain our variables a, b, c.

Planning!

- Before we talk about .text, stack, and heap, we will first talk about some x86 basics

x86-64 Basics

Instructions

- Each “command” that you issue to a processor is an instruction.
- For example, “store this variable there” or “load this data”.

- Each instruction is a very simple operation.

Register File

- Lots of registers, but here are the 6 general purpose registers that we are concerned with:

rax, rcx, rdx, rbx, rsi, rdi

Registers are like variables

- That's right, we're pretty much going to do everything simple in 6 registers, and a few extra registers for additional functionality

Instructions Location?

- The serialized instructions are contained in a .text segment.
- Not always called .text, but we will refer to “the .text segment” with the assumption that it is the relevant executable segment.

Where am I?

- Some extremely special purpose registers that aren't interacted with directly.
- Example: `rip`

- Instruction pointer (points to the memory address of the current instruction)

Load/Store memory

```
mov rax,[rsi+0048]
mov [rsi+0048],rax
```

This notation is ambiguous, but acceptable shorthand

Load/Store memory

```
mov rax,[rsi+0048]
mov [rsi+0048],rax
```

v.s.

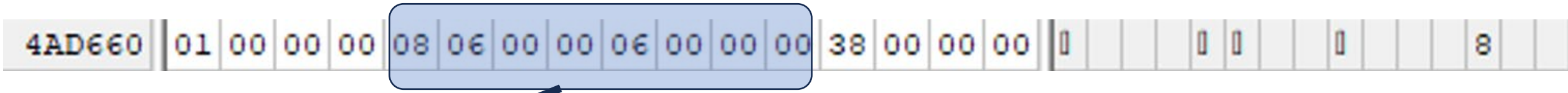
```
mov rax, qword[rsi+0048]
mov qword[rsi+0048],rax
```

Better! Size of storage is known

Load/Store memory

```
mov rax, qword[4AD664]
```

Idea: Find memory location,
put 8 bytes of data in rax



rax assigned value 0x0600000608 (leading 00s omitted)

Add/Subtract

```
add rax, rdx      // rax += rdx
add rcx, 5        // rcx += 5
sub rax, 1        // rax -= 1
```

Jump/Branch

```
jmp rax
jmp 4000 0096
```

Unconditional Jump (not very interesting)

Jump to some other location in executable code

Jump/Branch

```
cmp eax,0      // Compare eax to 0
               // Will set conditional flags
```

CF, PF, AF, ZF, SF, OF

ZF ≡ “Is previous comparison zero?”

CFLAGs

CF, PF, AF, ZF, SF, OF

Idea: When comparing two parameters, generate everything! Are they equal? Is a<b? Etc.

CFLAGs

CF, PF, AF, ZF, SF, OF

Idea: When comparing two parameters, generate everything! Are they equal? Is a<b? Etc.

The actual operation: a−b

If it is zero (ZF), they are equal. If positive (¬SF): a>b;
If negative (SF): a<b; If positive (¬SF) OR ZF: a≥b; etc.

Jump/Branch

`jge, jg, jle, jl, je/jz, jne/jnz`

In order: “Jump if… $\geq, >, \leq, <, =, \neq$ “

Comparison of code

Code that you see

```
if( rax == 3 )
      rcx = 4;
else
      rcx = 5;
print( rcx )
```

Code that CPU sees

```
          cmp rax,3
          je IsEqual
          mov rcx,5
          jmp End
IsEqual:  mov rcx,4
End:      push rcx
          call print
```

More on this later

- We will cover branching in more depth on Thursday
- For now, let's go back to .text, stack, and heap

Stack pointer(s)

`rsp, rbp`

`rsp= stack pointer`

`rbp= stack base pointer`

Stack pointer(s)

`rsp, rbp`

`rsp= stack pointer`

`rbp= stack base pointer`

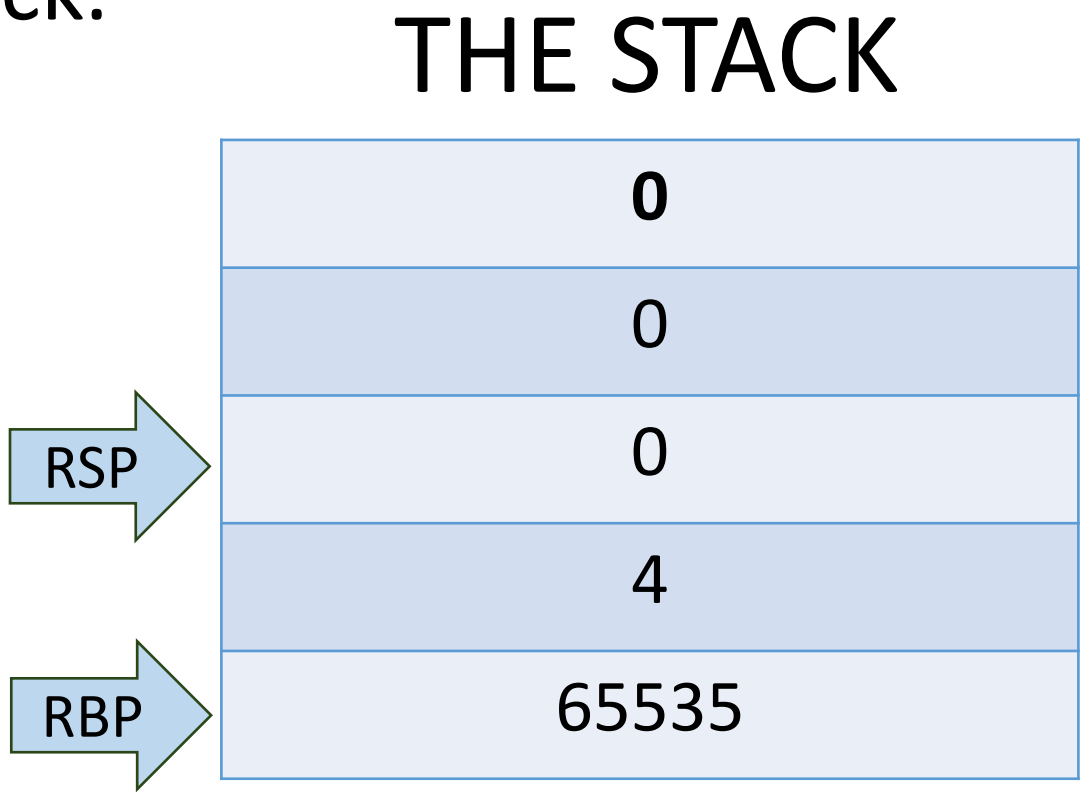
Used for: (1) parameters in a function call, (2) temporary variables, (3) stack framing for more temp variables, and more!

Stack Growth

Push/Pop data on/off the stack.

Each “entry” is 8 bytes
in this example.

For 32-bit,
it would be 4 bytes.

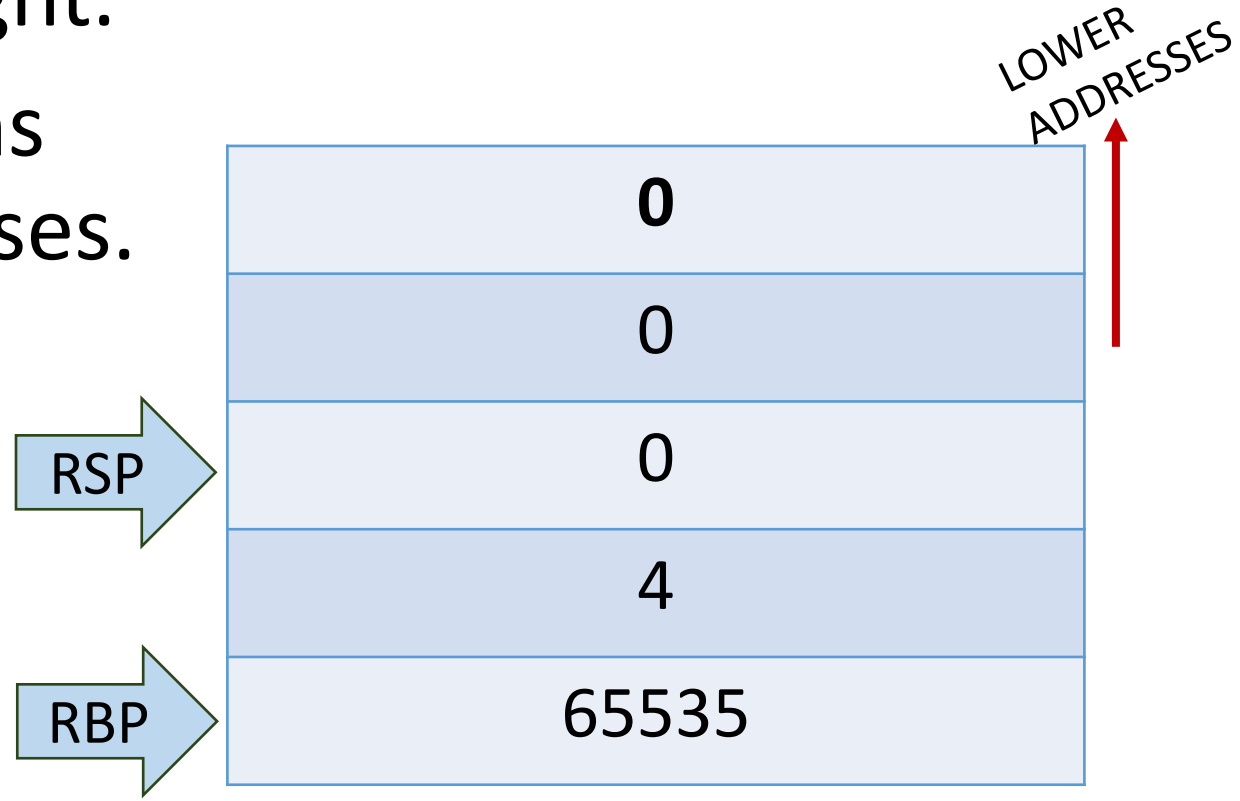


Stack Growth

Shown in the stack on the right.

Unintuitively, higher positions are at lower memory addresses.

E.g. the number “4” is at rbp-8, not rbp+8, or rsp+8, not rsp-8



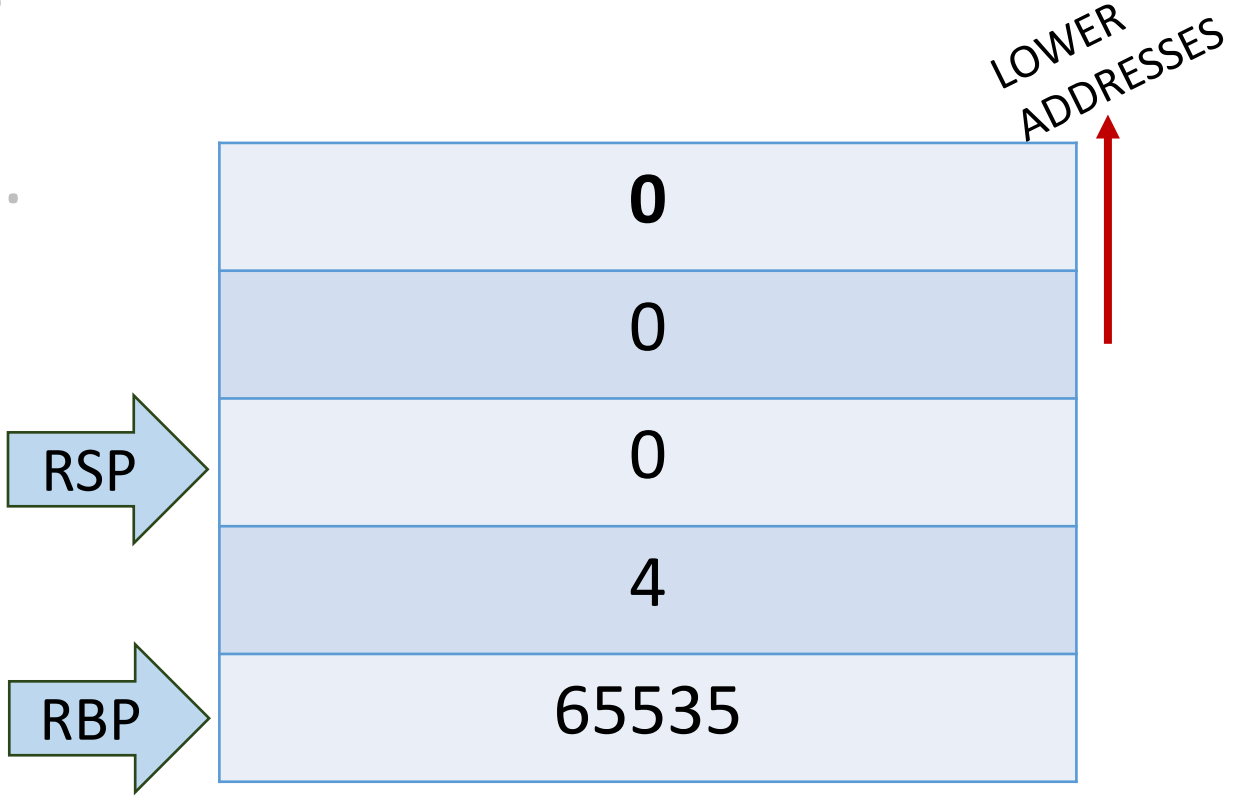
Stack Growth

Shown in the stack on the right.

Unintuitively, higher positions are at lower memory addresses.

E.g. the number “4” is at rbp-8, not rbp+8, or rsp+8, not rsp-8

Question: Why can’t the stack grow + instead of -?



Local Variables

- First covered usage of the stack: local variables!

```
public class MainMethod {
    public static void main(String[] args) {
        int x = 4;
        x = 5;
        System.out.println(x);
    }
}
```

Local Variables

- Local variables in a method are not a static variable in .bss, nor .data
- Two methods: use the heap (shown later), use the stack (recommended)
- Question: Why not give every local a location in bss?

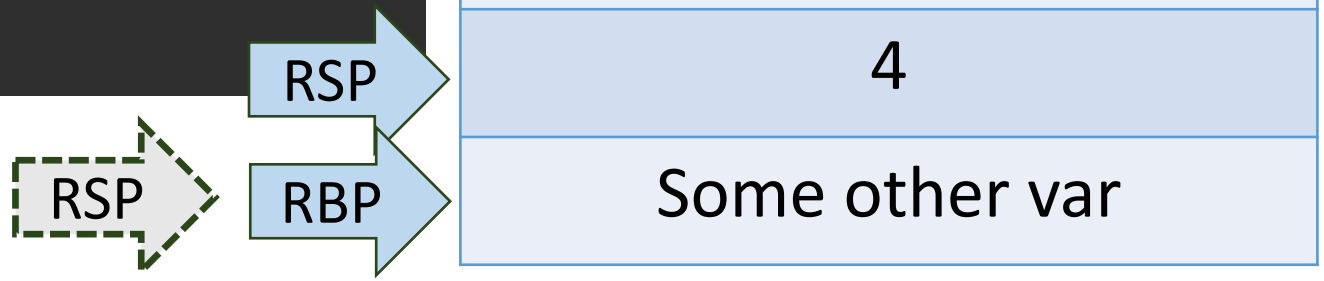
Local Variables

Consider:

```
public class MainMethod {
    public static void main(String[] args) {
        int x = 4;
        x = 5;
        System.out.println(x);
    }
}
```

RIP

push 4



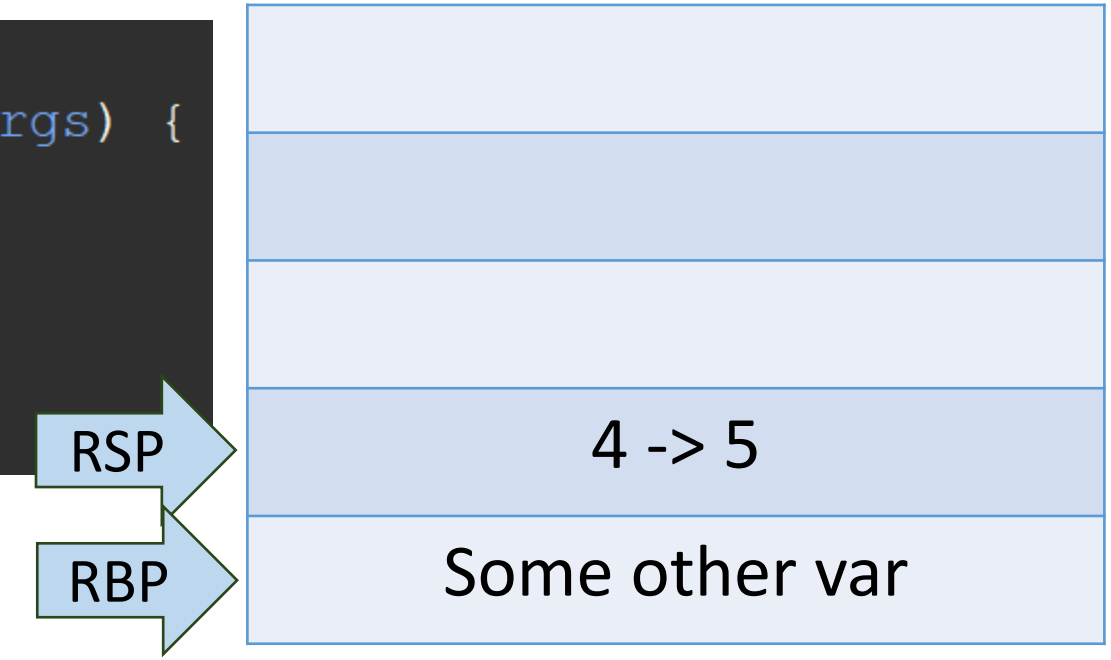
Local Variables

Consider:

```
public class MainMethod {
    public static void main(String[] args) {
        int x = 4;
        x = 5;
        System.out.println(x);
    }
}
```

RIP

```
mov [rbp-8], 5
```



Local Variables

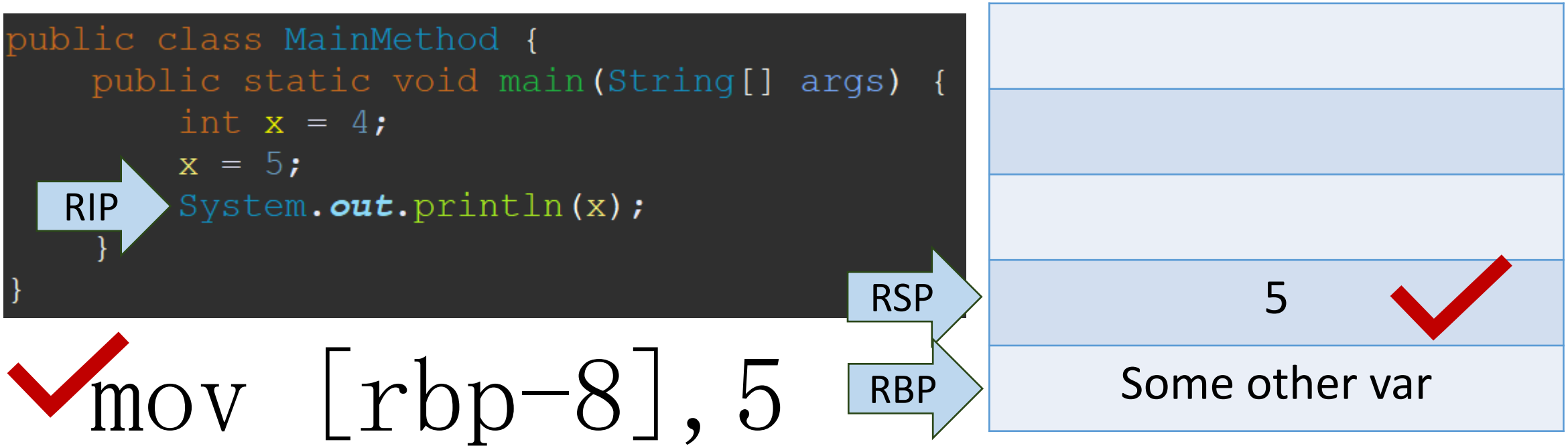
Consider:

```
public class MainMethod {
    public static void main(String[] args) {
        int x = 4;
        x = 5;
        System.out.println(x);
    }
}
```

RIP

RSP

RBP



✓ mov [rbp−8], 5

Next use of the stack

- How can we pass parameters to a method?

```
a = 3; b = 5;
someMethod( a, b );
```

Stack pointer(s)

```
rsp, rbp
```

someMethod(int a, int b);

```
push dword[b]
push dword[a]
call someMethod
```

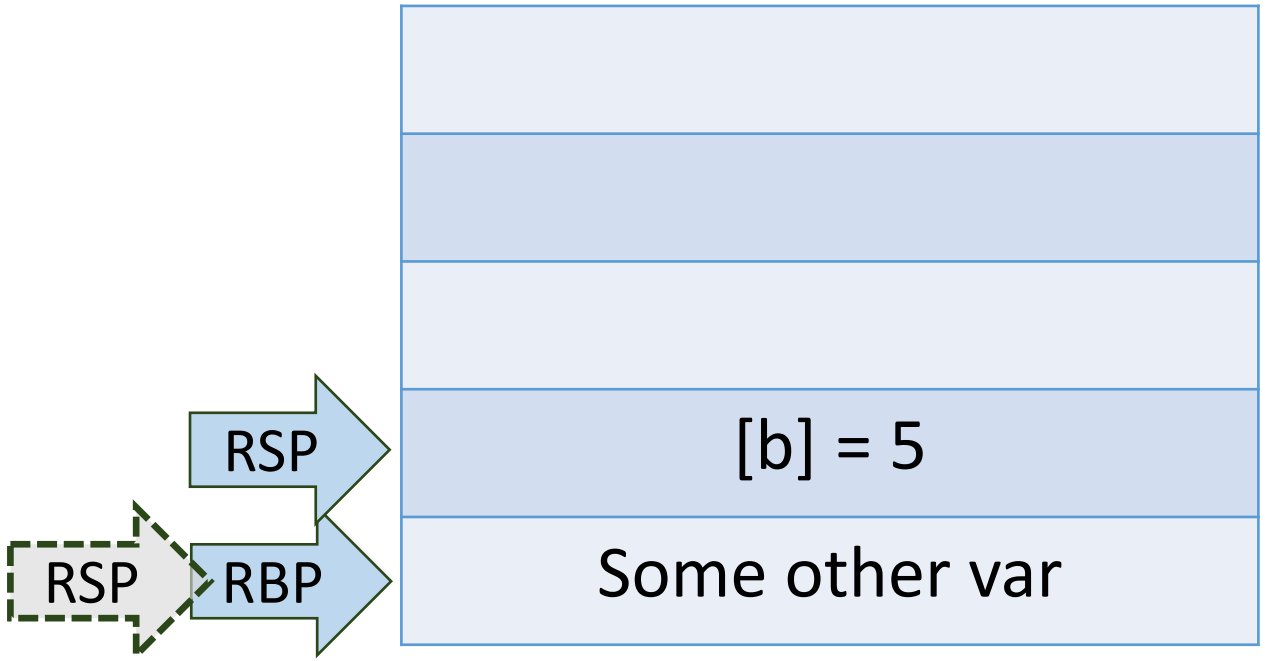
Stack pointer(s) – How to call a method

```
rsp, rbp
```

someMethod(a, b);

RIP

```
push dword[b]
push dword[a]
call someMethod
```



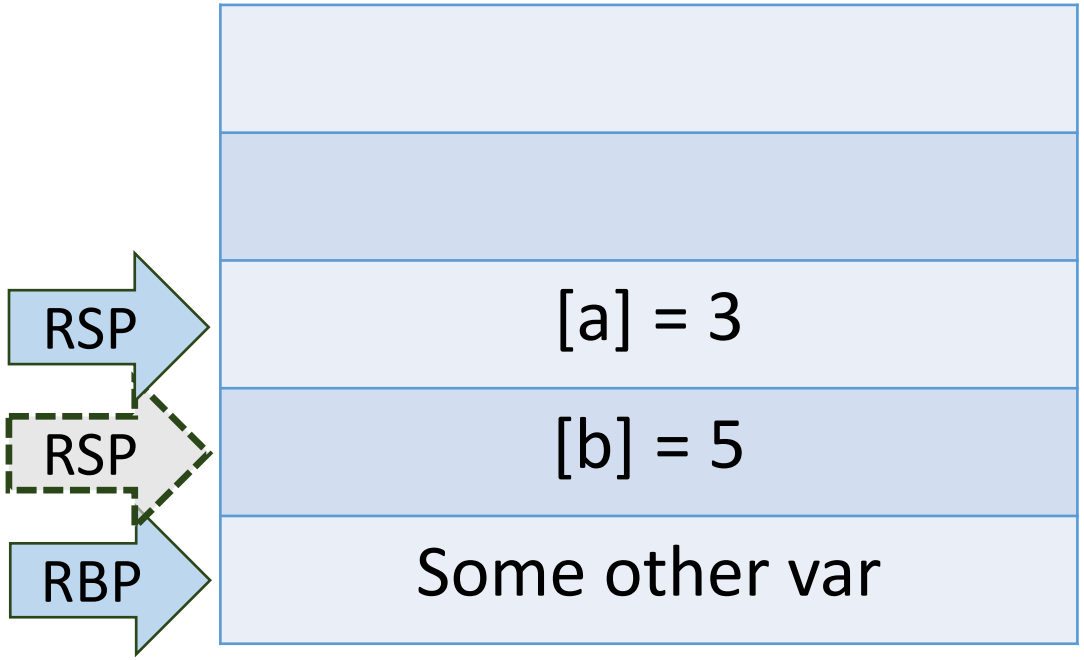
Stack pointer(s) – How to call a method

```
rsp, rbp
```

someMethod(a, b);

```
push dword[b]
push dword[a]
call someMethod
```

RIP



Stack pointer(s) – How to call a method

```
rsp, rbp
```

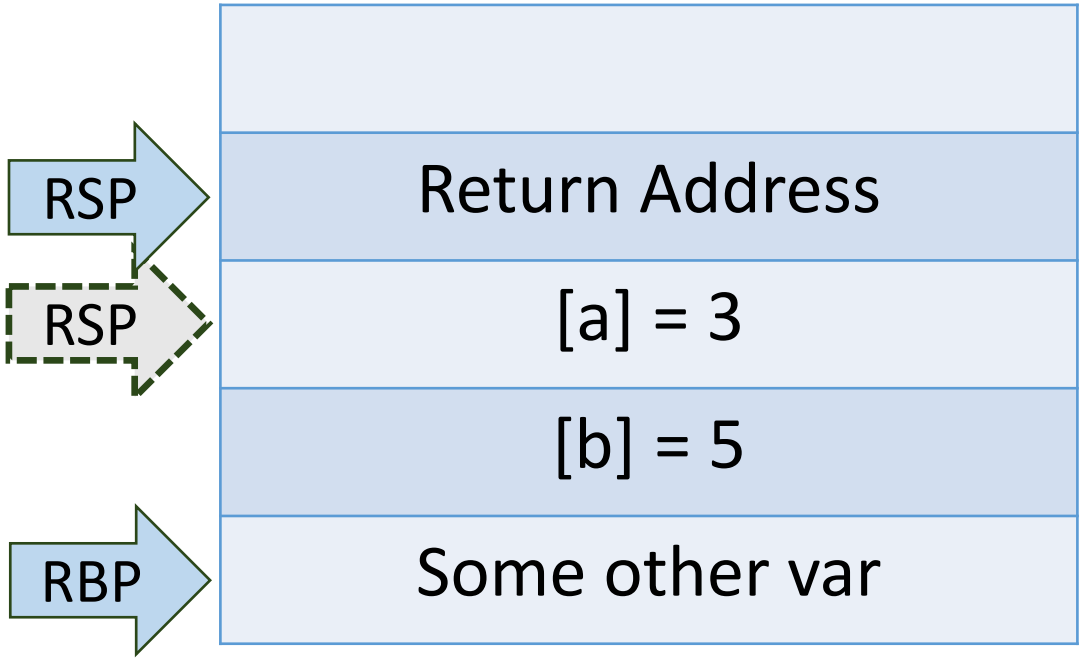
someMethod(a, b);

```
push dword[b]
push dword[a]
call someMethod
```

RIP

Return Address

…



Stack pointer(s) – How to call a method

`rsp, rbp`

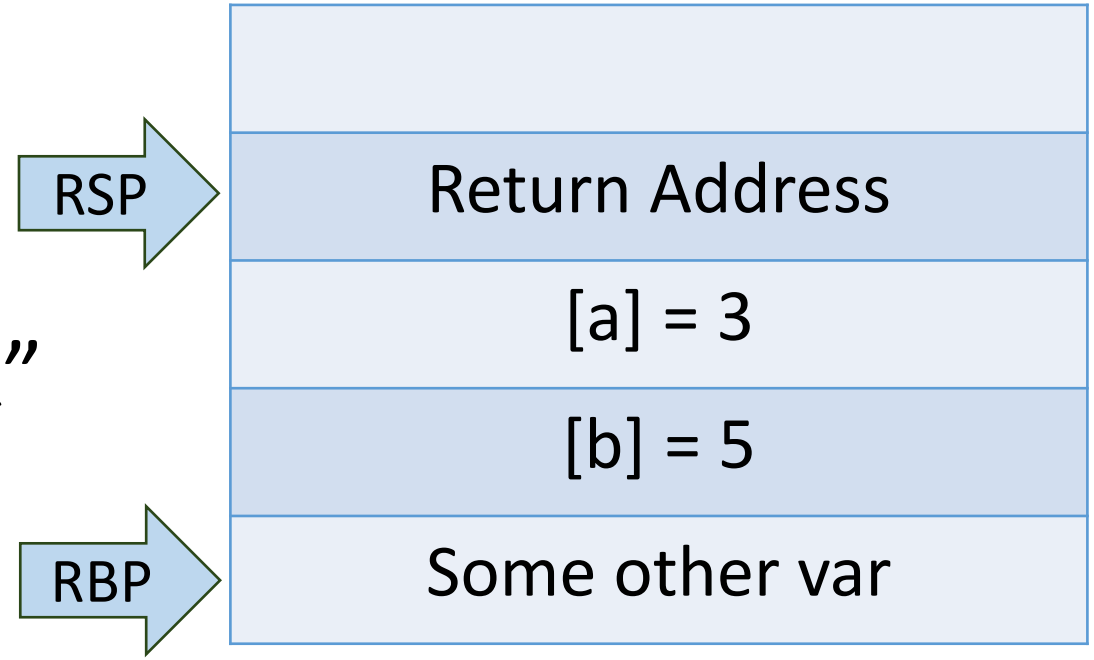
At the end of someMethod:

RIP `ret`

“Take address at top of stack”

“Set RIP to be that address”

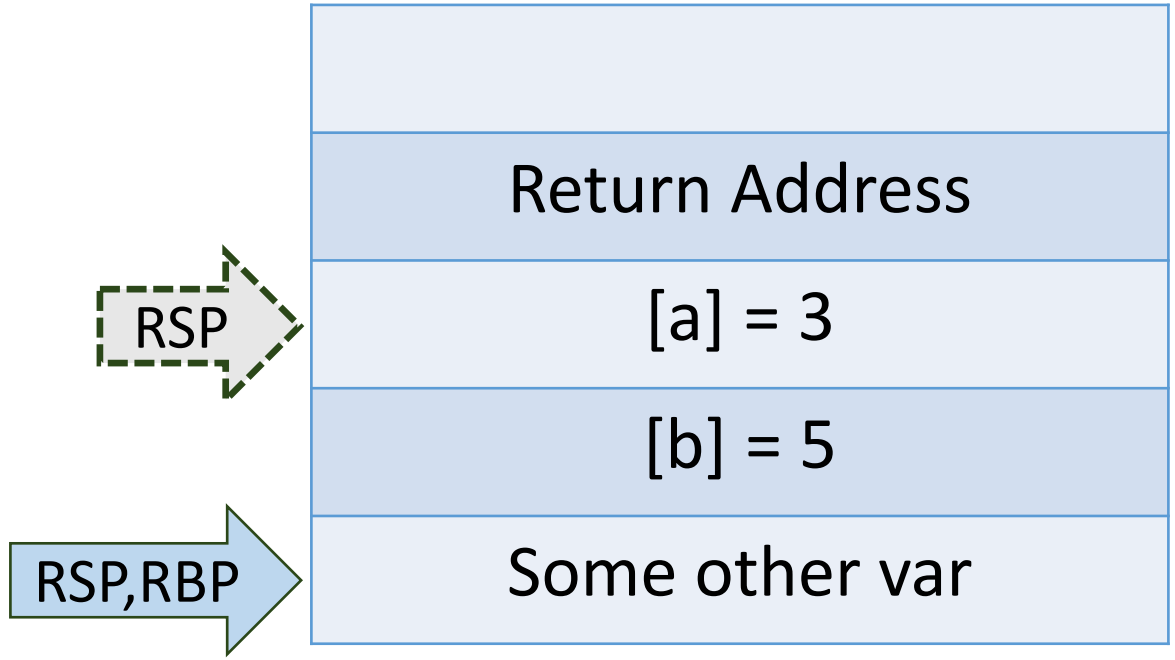
“Pop the top of the stack”



Stack pointer(s) – How to call a method

```
rsp, rbp
```

```
someMethod( a, b );
        push dword[b]
        push dword[a]
        call someMethod
RIP     add rsp,16
```



What does a **CALLED** method look like?

- Stack framing will be covered in an upcoming lecture

- Idea: "keep our local variables in the stack, then when we call a method, create a new '*frame*' where that method can store ITS OWN local variables"

Stack Framing will be a separate lecture

- I know it feels strange to keep saying "this will be done later"
- But some of these are heavy topics that need some separation (aka, breathing room)

Stack Framing will be a separate lecture

- I know it feels strange to keep saying “this will be done later”
- But some of these are heavy topics that need some separation (aka, breathing room)
- For now, focus on the basics, and the nitty gritty details (which are actually the most interesting) will be thoroughly investigated, rest assured.

Lastly, pop

- Very straight forward,

1) Reads the value at the top of the stack (`rsp`), stores it in some destination register

```
pop rcx
```

2) Adds 8 to `rsp`

Back to memory organization

.text Segment

- As specified earlier, this is where code is stored

```
00007FF7796B18CB B9 08 00 00 00       mov         ecx,8
00007FF7796B18D0 E8 67 F7 FF FF       call        operator new (07FF7796B103Ch)
00007FF7796B18D5 48 89 85 E8 00 00 00 mov         qword ptr [rbp+0E8h],rax
00007FF7796B18DC 48 83 BD E8 00 00 00 00 cmp          qword ptr [rbp+0E8h],0
00007FF7796B18E4 74 20                jz          main+56h (07FF7796B1906h)
```

- Question: can code exist in other segments? What about non-executable segments?

Static vs Dynamic memory

- But where is THIS data stored?

```
int[] p = new int[ someVariableSize ];
```

Doesn't fit our notions of .bss nor .data! (Why?)
If we use the stack, then we consume
a ton of stack space.

Dynamic memory is in the heap

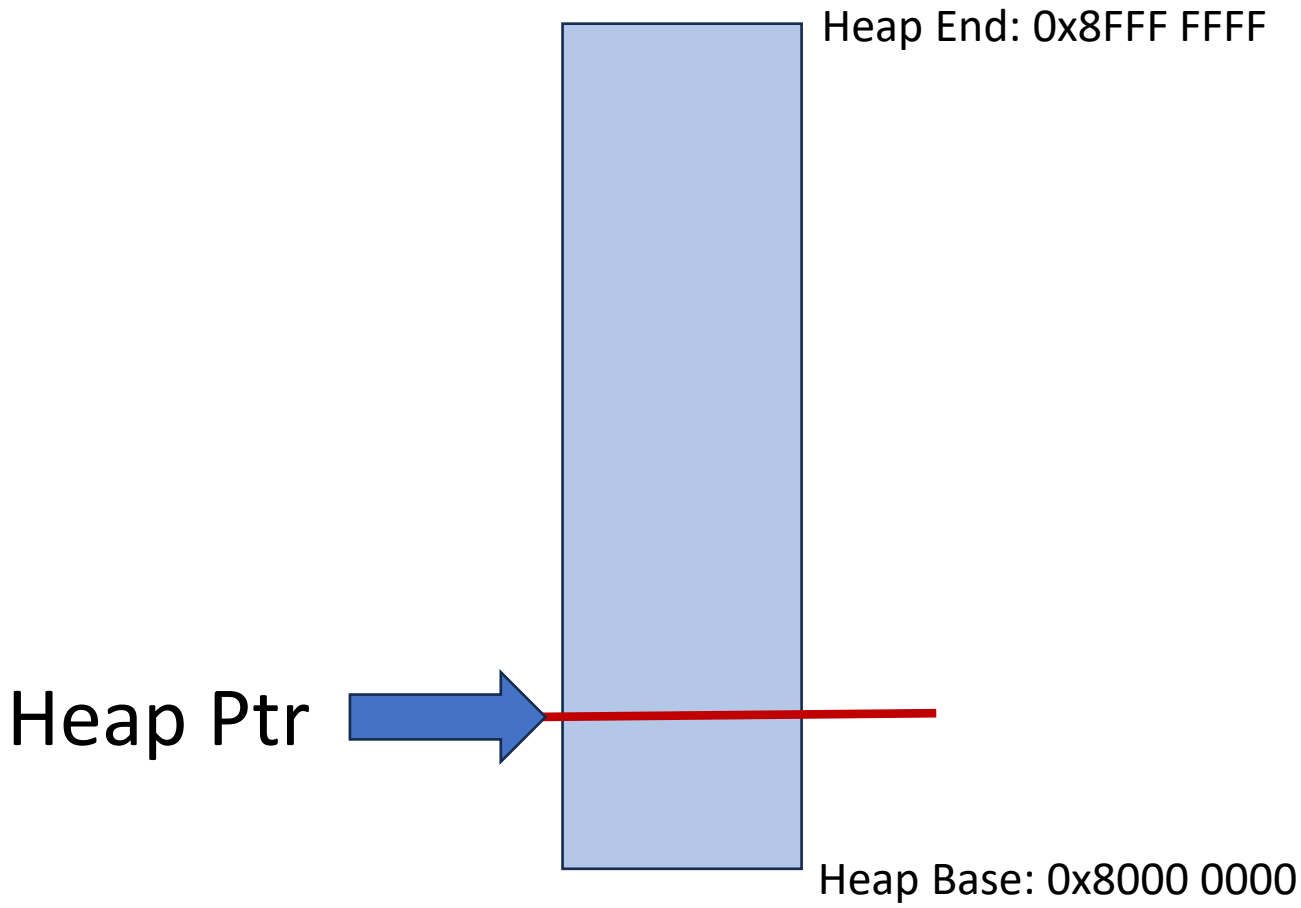
```
int[] p = new int[ someVariableSize ];
```

The heap is just a memory location.

Simplest heap possible:

```
void* malloc(size_t size) {
    void* addr = heap_ptr;
    heap_ptr += size;
    return addr;
}
```

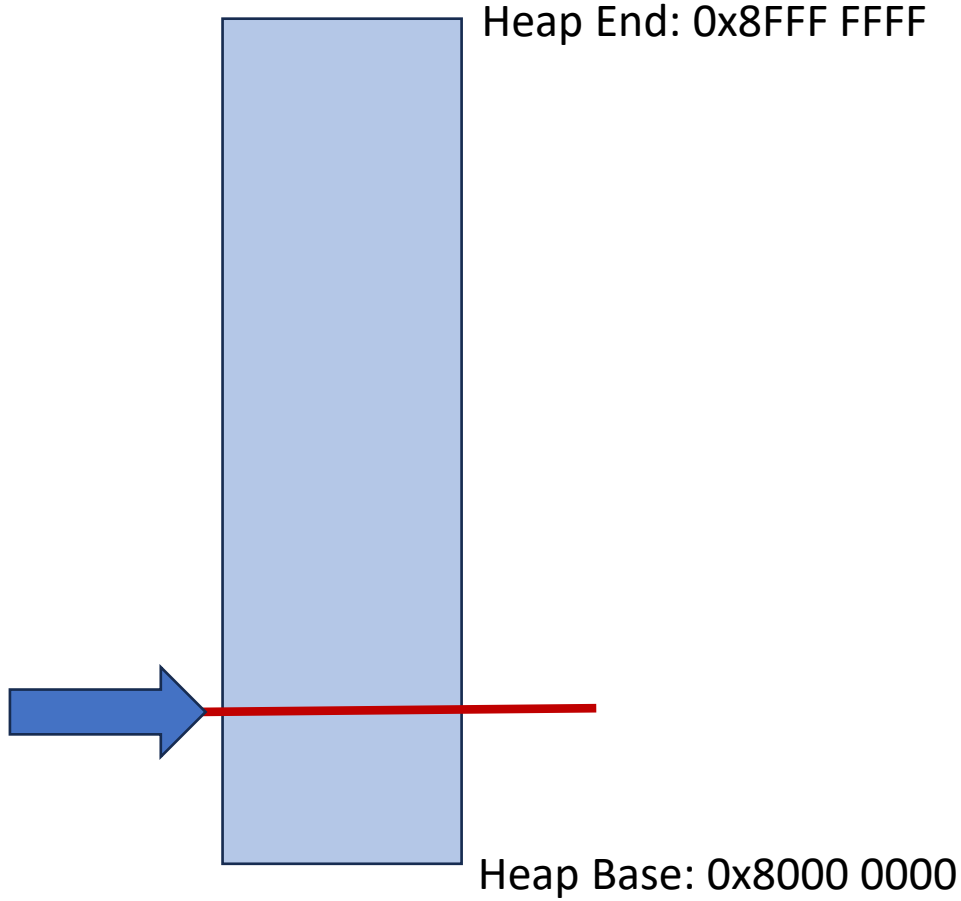
Super-simple heap



Super-simple heap

I want 3072 bytes of data!

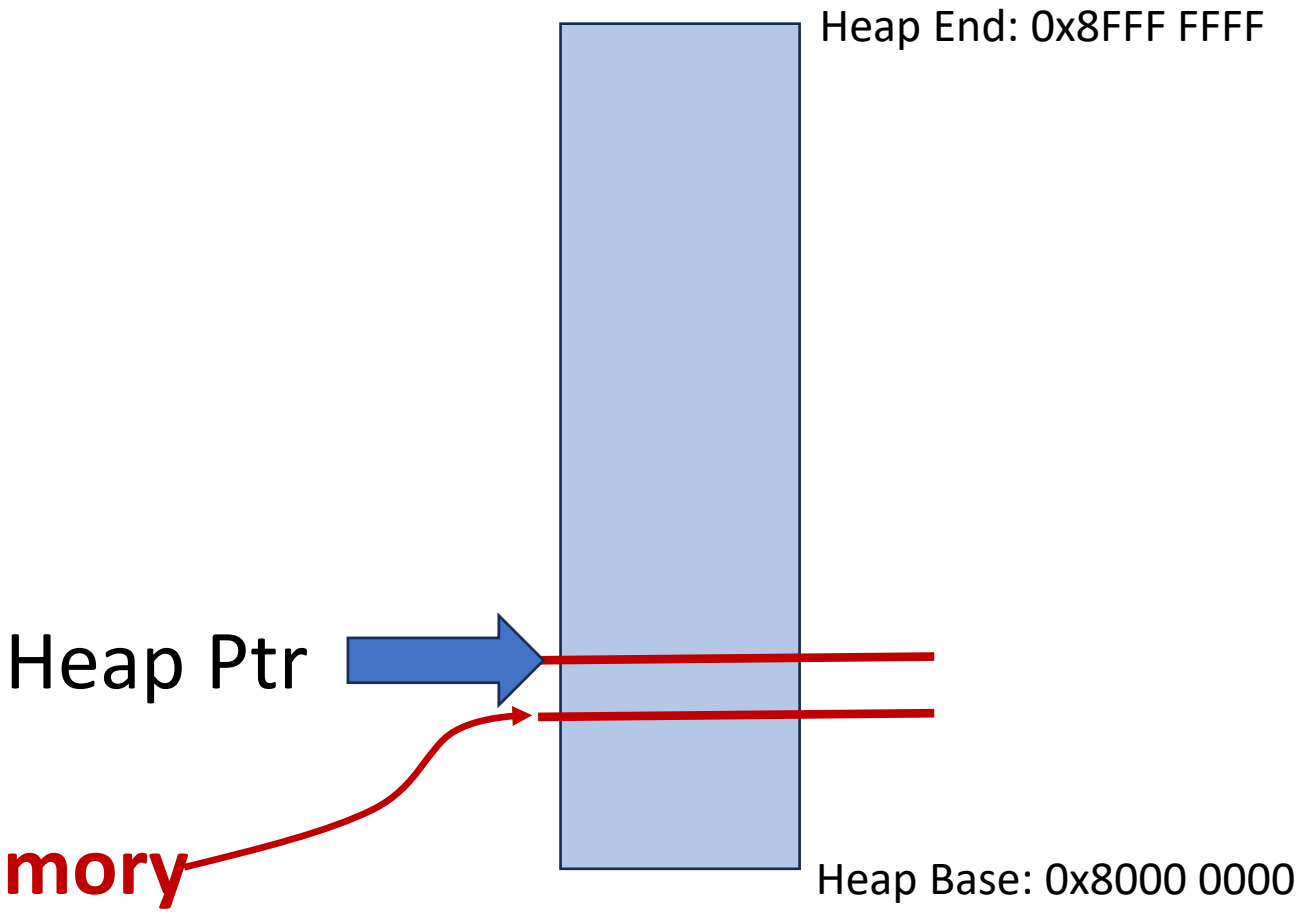
```
malloc(3072)
new char[3072]
```



Super-simple heap

I want 3072 bytes of data!

```
malloc(3072)
new char[3072]
```



Returned start of allocated memory

What does this look like?

Consider:

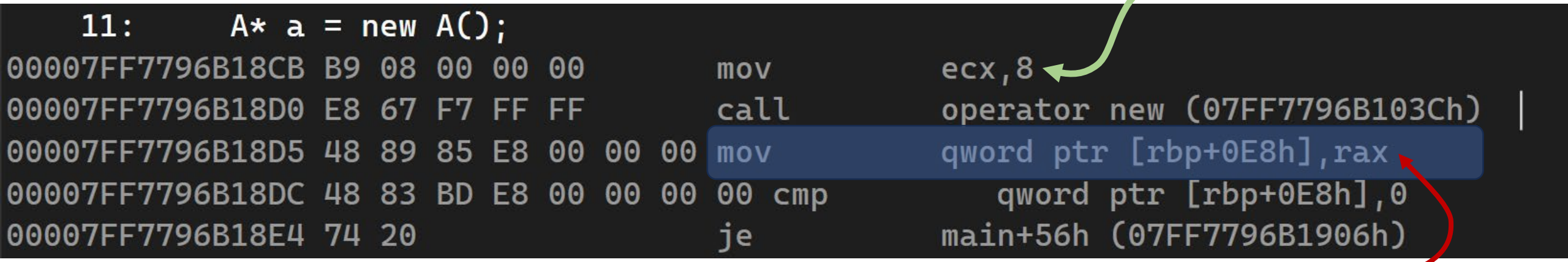
```
1 class A {
2     public int x;
3     public int y;
4 }
5
6 public class MainMethod {
7     public static void main(String[] args) {
8         A a = new A();
9         a.x = 3;
10        a.y = 5;
11    }
12 }
```

How is something returned?

```
    11:     A* a = new A();
00007FF7796B18CB B9 08 00 00 00       mov         ecx,8
00007FF7796B18D0 E8 67 F7 FF FF       call        operator new (07FF7796B103Ch)
00007FF7796B18D5 48 89 85 E8 00 00 00 mov         qword ptr [rbp+0E8h],rax
00007FF7796B18DC 48 83 BD E8 00 00 00 00 cmp          qword ptr [rbp+0E8h],0
00007FF7796B18E4 74 20                je          main+56h (07FF7796B1906h)
```

- Assume [rbp+0E8h] is the variable a
- Where is the returned value from the “new” operation?

How is something returned?



- Assume [rbp+0E8h] is the variable a
- Where is the return value from the “new” operation?

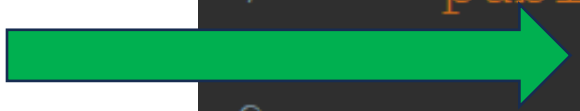
Assumptions

- We will assume that all functions/methods return their value in rax
- So rax will be “reserved” when doing a call, but general purpose otherwise

What does this look like?

Consider:

```
push 8
call malloc
mov [a], rax
```

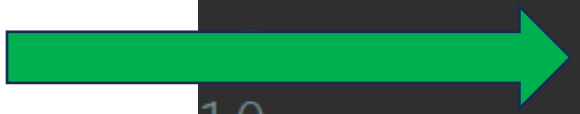


```
1  class A {
2      public int x;
3      public int y;
4  }
5
6  public class MainMethod {
7      public static void main(String[] args) {
8          A a = new A();
9          a.x = 3;
10         a.y = 5;
11     }
12 }
```

What does this look like?

Consider:

```
push 8
call malloc
mov [a], rax
mov [rax+0],3
```

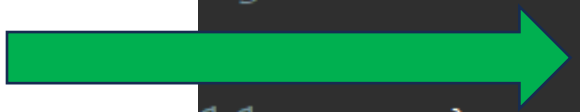


```
1 class A {
2     public int x;
3     public int y;
4 }
5
6 public class MainMethod {
7     public static void main(String[] args) {
8         A a = new A();
9         a.x = 3;
10        a.y = 5;
11    }
12 }
```

What does this look like?

Consider:

```
push 8
call malloc
mov [a], rax
mov [rax+0],3
mov [rax+4],5
```



```
1  class A {
2      public int x;
3      public int y;
4  }
5
6  public class MainMethod {
7      public static void main(String[] args) {
8          A a = new A();
9          a.x = 3;
           a.y = 5;
11     }
12 }
```

Field variables are offsets

```
1 class A {
2     public int x;
3     public int y;
4 }
```

x: From some base address, **add +0**

y: From some base address, **add +4**

So how did we figure out the alloc size of "A"?

- It has two variables, both of them are `int`, so assume `int` means 4 bytes (we will assume 8 bytes in miniJava), then the alloc size of A will be 8 bytes total.

```
1  class A {
2      public int x;
3      public int y;
4  }
5
6  public class MainMethod {
7      public static void ma
8          A a = new A();
9          a.x = 3;
10         a.y = 5;
11     }
12 }
```

So how did we figure out the alloc size of "A"?

- It has two variables, both of them are `int`, so assume `int` means 4 bytes (we will assume 8 bytes in miniJava), then the alloc size of A will be 8 bytes total.
- Thus:
```
push 8
call malloc
```

```
1  class A {
2      public int x;
3      public int y;
4  }
5
6  public class MainMethod {
7      public static void ma
8          A a = new A();
9          a.x = 3;
10         a.y = 5;
11     }
12 }
```

Coming up next!

- How can we handle a.b.c.d.x?

- What about classes like this:
- What is their size?
- (How many bytes is allocated when creating a new A()?)

```
1 class A {
2       int x;
3       B b;
4 }
5
6 class B {
7       int x;
8       A a;
9 }
```

Review

Review Lec 12

- Basic assembly operations: `mov` from memory, `add`, subtract, store to memory (`mov`), `call`, `push`, `pop`, `jmp`, `cmp`, conditional jumps (`jle`,`jl`,`jge`,`jg`,`je`,`jne`)

- Know about stack, heap, .bss, .data, .text

Thursday

- Live demo of branching

- Will step through the assembly and show it in action
- Will prove that assembly is just like any other thing you have programmed

End