

Lecture 4: Introduction to Assembly

CS 105

Fall 2024

Programs

```
#include<stdio.h>

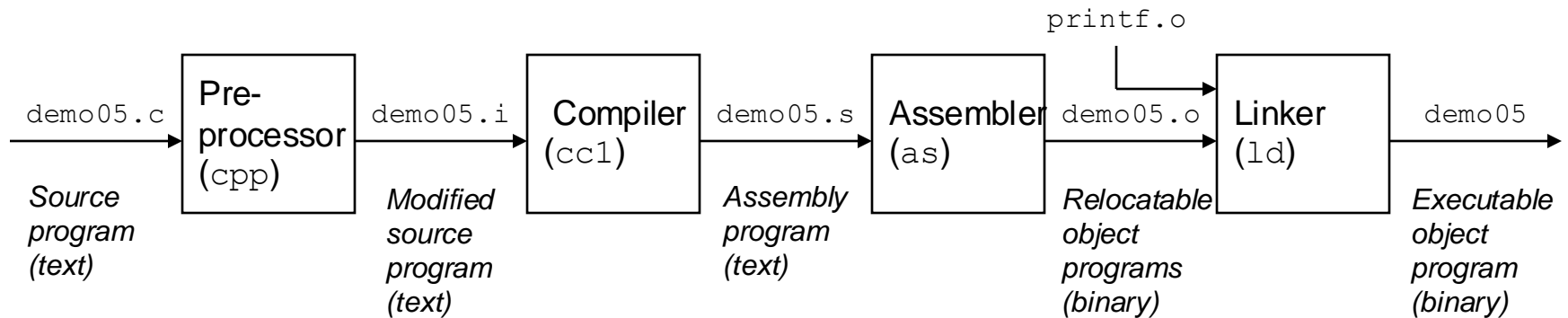
int main(int argc,
         char** argv) {

    printf("Hello world!\n");

    return 0;
}
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

Compilation



```
#include<stdio.h>

int main(int argc,
         char ** argv){

    printf("Hello
           world!\n");
    return 0;
}
```

```
...
int printf(const char *
           restrict,
           ...)
    __attribute__((__format__
                  (__printf__, 1, 2)));
...
int main(int argc,
         char ** argv){

    printf("Hello
           world!\n");
    return 0;
}
```

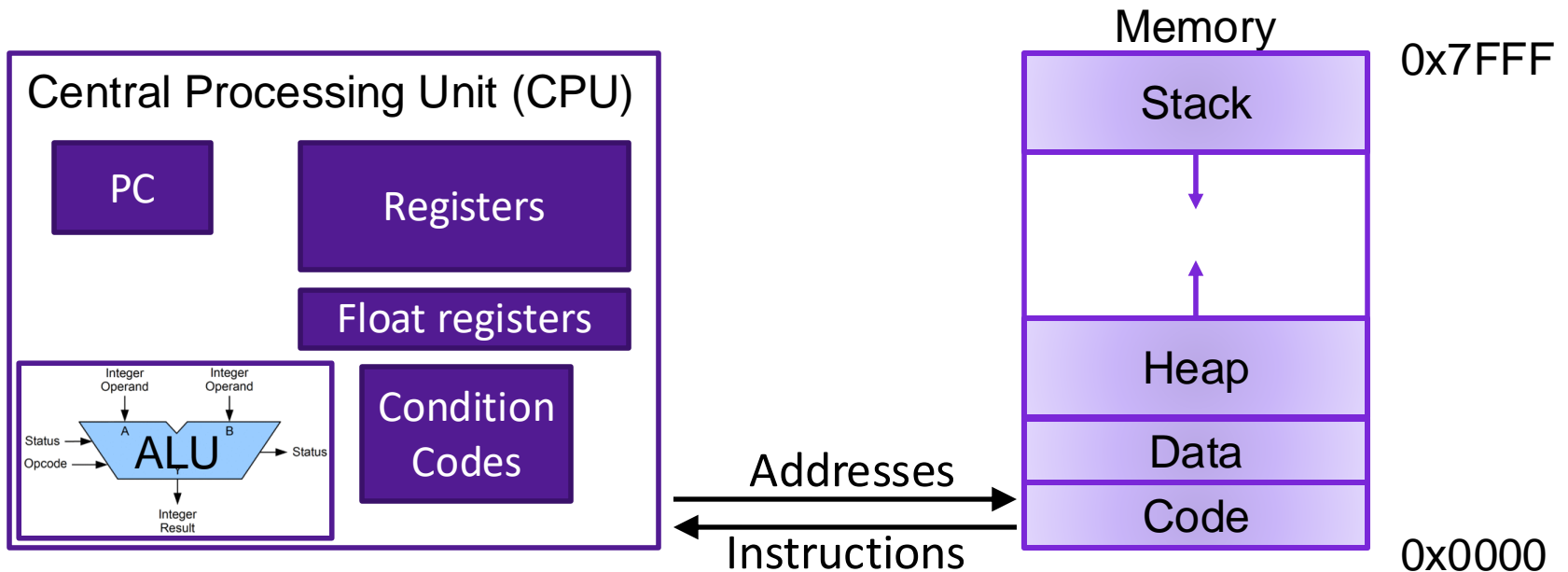
```
pushq   %rbp
movq    %rsp, %rbp
subq    $32, %rsp
leaq    L_.str(%rip), %rax
movl    $0, -4(%rbp)
movl    %edi, -8(%rbp)
movq    %rsi, -16(%rbp)
movq    %rax, %rdi
movb    $0, %al
callq   _printf
xorl    %ecx, %ecx
movl    %eax, -20(%rbp)
movl    %ecx, %eax
addq    $32, %rsp
popq    %rbp
retq
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

x86-64 Assembly Language

- Evolutionary design, going back to 8086 in 1978
 - Basis for original IBM Personal Computer, 16-bits
- Intel Pentium 4E (2004): 64 bit instruction set
- High-level languages are translated into x86 instructions and then executed on the CPU
 - Actual instructions are sequences of bytes
 - We give them mnemonic names

Assembly/Machine Code View



Programmer-Visible State

- ▶ PC: Program counter (%rip)
- ▶ Register file: 16 Registers
- ▶ Float registers
- ▶ Condition codes

Memory

- ▶ Byte addressable array
- ▶ Code and user data
- ▶ Stack to support procedures

Assembly Characteristics: Instructions

- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Perform arithmetic operations on register or memory data
- Transfer control
 - Unconditional jumps to/from functions
 - Conditional branches

X86-64 Integer Registers

%rax

%rbx

%rcx

%rdx

%rsi

%rdi

%rsp

%rbp

%r8

%r9

%r10

%r11

%r12

%r13

%r14

%r15

Data Movement Instructions

- MOV source, dest Moves data source->dest
dest = source

Operand Forms

- Immediate:
 - Syntax: \$c Ex: \$47 Val: c C Equiv: 47
- Register:
 - Syntax: r Ex: %rdi Val: Reg[r] C Equiv: x
- Memory (Absolute):
 - Syntax: addr Ex: 0x4050 Val: Mem[addr] ~~C Equiv: *0x60201a~~
- Memory (Indirect):
 - Syntax: (r) Ex: (%rsp) Val: Mem[Reg[r]] C Equiv: *x

Exercise: Operands

Register	Value
%rax	0x100
%rcx	0x01
%rdx	0x03

Memory Address	Value
0x100	0xFF
0x101	0x47
0x102	0x13
0x103	0xE1
0x104	0xAB
0x105	0x2F

- What are the values of the following operands (assuming register and memory state shown above)?
 1. %rax
 2. 0x104
 3. \$0x102
 4. (%rax)

mov Operand Combinations

	Source	Dest	Src, Dest	C Analog
mov	Imm	Reg	mov \$0x4,%rax	x = 4;
		Mem	mov \$-147, (%rdx)	*p = -147;
	Reg	Reg	mov %rax,%rcx	y = x;
		Mem	mov %rax, (%rdx)	*p = x;
	Mem	Reg	mov (%rdx),%rax	x = *p;

Cannot do memory-memory transfer with a single instruction

Exercise: Moving Data

- For each of the following move instructions, write an equivalent C assignment
 1. `mov $0x40604a, %rbx`
 2. `mov %rbx, %rax`
 3. `mov $47, (%rax)`

Sizes of C Data Types in x86-64

C declaration	Size (bytes)	Intel data type	Assembly suffix
char	1	Byte	b
short	2	Word	w
int	4	Double word	l
long	8	Quad word	q
char *	8	Quad word	q
float	4	Single precision	s
double	8	Double precision	l

Data Movement Instructions

- MOV source, dest
 - movb Move 1 byte
 - movw Move 2 bytes
 - movl Move 4 bytes
 - movq Move 8 bytes

X86-64 Integer Registers

%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%bsl
%rbp	%ebp	%bp	%bpl

%r8	%r8d		
%r9	%r9d		
%r10	%r10d		
%r11	%r11d		
%r12	%r12d		
%r13	%r13d		
%r14	%r14d		
%r15	%r15d		

X86-64 Integer Registers

%rax (function result)

%rbx

%rcx (fourth argument)

%rdx (third argument)

%rsi (second argument)

%rdi (first argument)

%rsp (stack pointer)

%rbp

%r8 (fifth argument)

%r9 (sixth argument)

%r10

%r11

%r12

%r13

%r14

%r15

Exercise: Translating Assembly

- Write a C function `void decode1(long* xp, long* yp)` that will do the same thing as the following assembly code:

```
decode:
```

```
    movq (%rdi), %rax
    movq (%rsi), %rcx
    movq %rax, (%rsi)
    movq %rcx, (%rdi)
    ret
```

```
void decode(long* xp, long* yp){
}

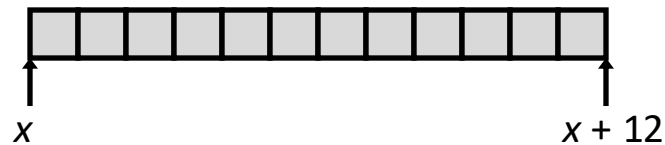
```

Register	Use(s)
<code>%rdi</code>	Argument <code>xp</code>
<code>%rsi</code>	Argument <code>yp</code>

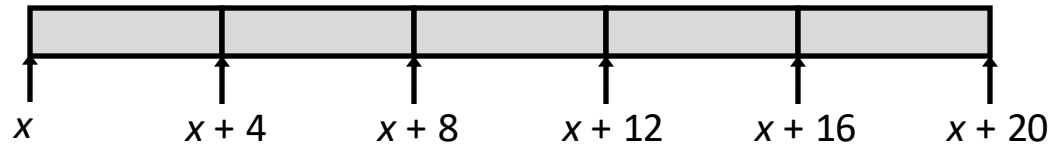
Review: Array Allocation

- Basic Principle $T \mathbf{A}[L]$;
 - Array of data type T and length L
 - Contiguously allocated region of $L * \mathbf{sizeof}(T)$ bytes in memory
 - Identifier \mathbf{A} can be used as a pointer to array element 0: Type T^*

`char string[12];`



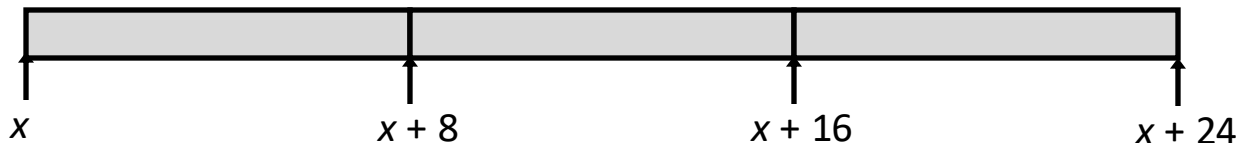
`int val[5];`



`double a[3];`

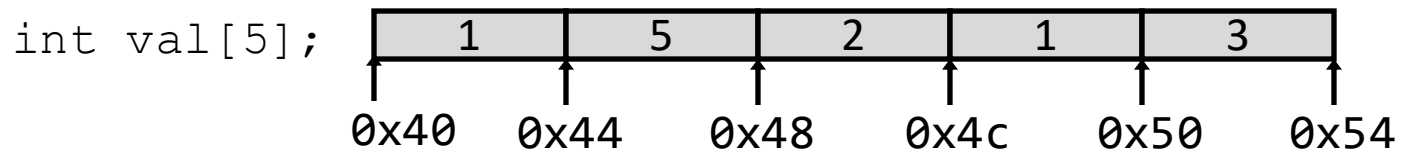


`char* p[3];`



Exercise: Array Access

- Basic Principle $T \mathbf{A}[L]$;
 - Array of data type T and length L
 - Contiguously allocated region of $L * \mathbf{sizeof}(T)$ bytes in memory
 - Identifier \mathbf{A} can be used as a pointer to array element 0: Type T^*



- Reference Type Value
 - `val[4]`
 - `val`
 - `val+1`
 - `&(val[2])`
 - `val[5]`
 - `*(val+1)`

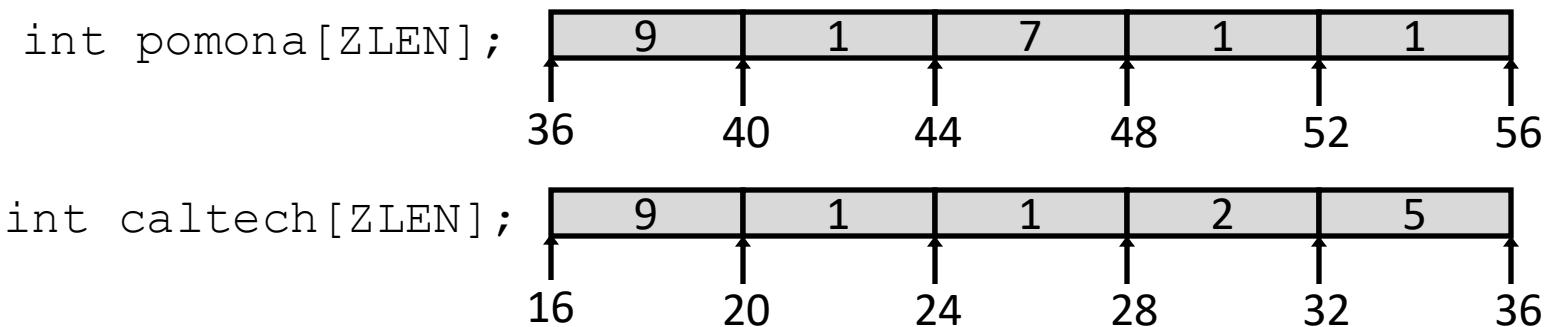
Array Example

```
#define ZLEN 5

int pomona[ZLEN] = { 9, 1, 7, 1, 1 };
int caltech[ZLEN] = { 9, 1, 1, 2, 5 };

void cycle_digits(int* zipcode){
    int temp = zipcode[0];
    zipcode[0] = zipcode[1];
    zipcode[1] = zipcode[2];
    zipcode[2] = zipcode[3];
    zipcode[3] = zipcode[4];
    zipcode[4] = temp;
}
```

???



Operand Forms

- Immediate:

- Syntax: \$c

Ex: \$47

Val: c

C Equiv: 47

- Register:

- Syntax: r

Ex: %rbp

Val: Reg[r]

C Equiv: x

- Memory (Absolute):

- Syntax: addr

Ex: 0x4050

Val: Mem[addr]

C Equiv: *0x60201a

- Memory (Indirect):

- Syntax: (r)

Ex: (%rsp)

Val: Mem[Reg[r]]

C Equiv: *x

- Memory (Base+displacement):

- Syntax: c(r)

Ex: 12(%rsp)

Val: Mem[Reg[r]+c]

C Equiv: *(x+12)

Exercise: Operands

Register	Value
%rax	0x108
%rcx	0x01
%rdx	0x03

Memory Address	Value
0x108	0xFF
0x109	0x47
0x10a	0x13
0x10b	0xE1
0x10c	0xAB
0x10d	0x2F

- What are the values of the following operands (assuming register and memory state shown above)?
 1. `(%rax)`
 2. `1(%rax)`
 3. `4(%rax)`

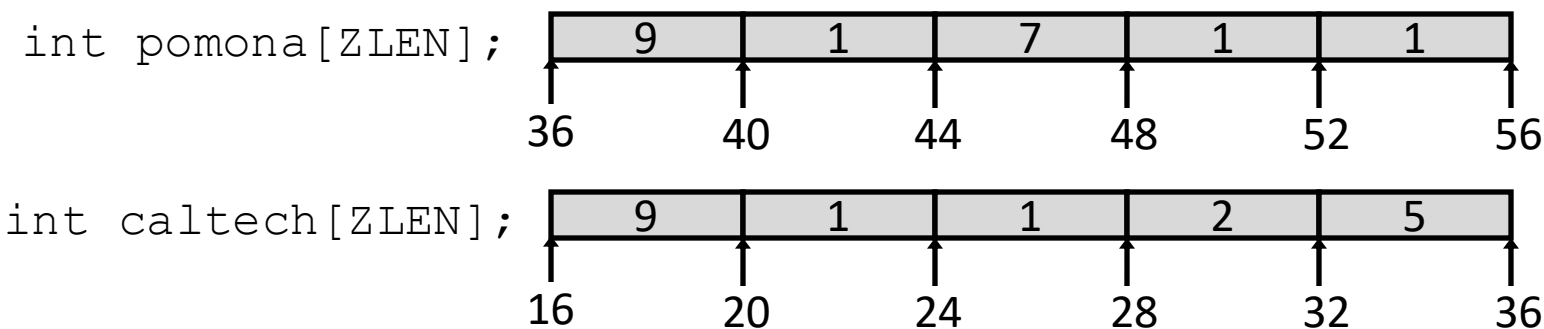
Array Example

```
#define ZLEN 5

int pomona[ZLEN] = { 9, 1, 7, 1, 1 };
int caltech[ZLEN] = { 9, 1, 1, 2, 5 };

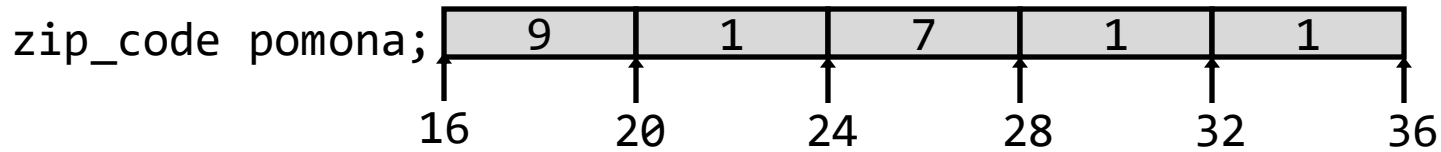
void cycle_digits(int* zipcode){
    int temp = zipcode[0];
    zipcode[0] = zipcode[1];
    zipcode[1] = zipcode[2];
    zipcode[2] = zipcode[3];
    zipcode[3] = zipcode[4];
    zipcode[4] = temp;
}
```

```
movl (%rdi), %rdx
movl 4(%rdi), %rcx
movl %rcx, (%rdi)
movl 8(%rdi), %rcx
movl %rcx, 4(%rdi)
movl 12(%rdi), %rcx
movl %rcx, 8(%rdi)
movl 16(%rdi), %rcx
movl %rcx, 12(%rdi)
movl %rdx, 16(%rdi)
```



Register	Use(s)
%rdi	z
%rsi	digit
%rax	return val

Array Accessing Example



```
int get_digit(int* zipcode, int digit){  
    return z[digit];  
}
```

???

Operand Forms

- Immediate:
 - Syntax: \$c Ex: \$47 Val: c C Equiv: 47
- Register:
 - Syntax: r Ex: %rbp Val: Reg[r] C Equiv: x
- Memory (Absolute):
 - Syntax: addr Ex: 0x4050 Val: Mem[addr] C Equiv: *0x60201a
- Memory (Indirect):
 - Syntax: (r) Ex: (%rsp) Val: Mem[Reg[r]] C Equiv: *x
- Memory (Base+displacement):
 - Syntax: c(r) Ex: 12(%rsp) Val: Mem[Reg[r]+c] C Equiv: *(x+12)
- Memory (Scaled indexed):
 - Syntax: (r1,r2,s) Ex: (%rdx,%rsi,4) Val: Mem[Reg[r1]+Reg[r2]*s] C: r1[r2]
- Memory (Scaled indexed w/ displacement):
 - Syntax: c(r1,r2,s) Ex: 8(%rdx,%rsi,4) Val: Mem[Reg[r1]+Reg[r2]*s+c] C: (r1+8)[r2]

Exercise: Operands

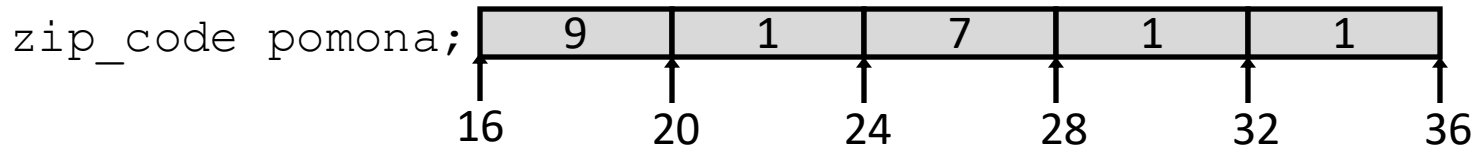
Register	Value
%rax	0x100
%rcx	0x01
%rdx	0x03

Memory Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x47

- What are the values of the following operands (assuming register and memory state shown above)?
 1. $(\%rax, \%rcx, 4)$
 2. $(\%rax, \%rdx, 4)$
 3. $8(\%rax, \%rcx, 4)$

Register	Use(s)
%rdi	z
%rsi	digit
%rax	return val

Array Accessing Example



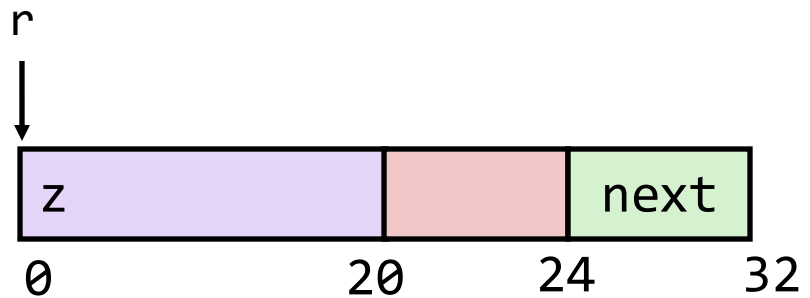
```
int get_digit(int* zipcode, int digit){
    return z[digit];
}
```

```
movl (%rdi,%rsi,4), %eax # ret = z[digit]
```

- Register %rdi contains starting address of array zipcode
- Register %rsi contains array index digit
- Desired digit at $\%rdi + 4 * \%rsi$
- Use memory reference $(\%rdi, \%rsi, 4)$

Structure Representation

```
struct node {  
    int z[5];  
    struct node* next;  
};
```

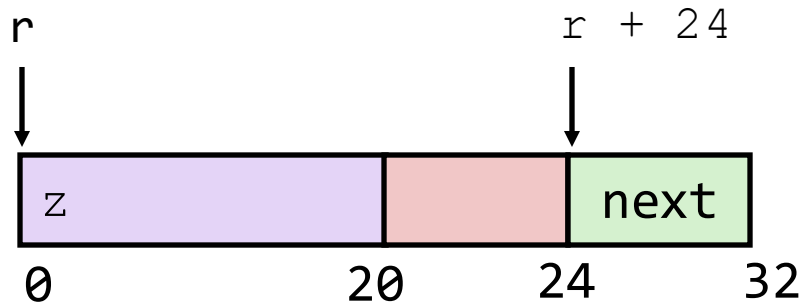


- Structure represented as block of memory
 - **Big enough to hold all of the fields**
- Fields ordered according to declaration
 - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
 - **Machine-level program has no understanding of the structures in the source code**

Register	Use(s)
%rdi	n
%rax	return val

Accessing Fields

```
struct node {
    int z[5];
    struct node* next;
};
```



- Accessing a field in a struct
 - Offset of each structure member determined at compile time

```
struct node* get_next(struct node* n){
    return n->next;
}
```

```
# n in %rdi
movq 24(%rdi), %rax
ret
```

C is close to Machine Language

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e: 48 89 03
```

- C Code
 - Store value `t` where designated by `dest`
- Assembly
 - Move 8-byte value to memory
 - Quad words in x86-64 parlance
 - Operands:
 - `t`: Register `%rax`
 - `dest`: Register `%rbx`
 - `*dest`: Memory `M[%rbx]`
- Object Code
 - 3-byte instruction
 - at address `0x40059e`