x86-64

Agenda

- x86-64
  - Concepts
  - Exercises
  - Demo

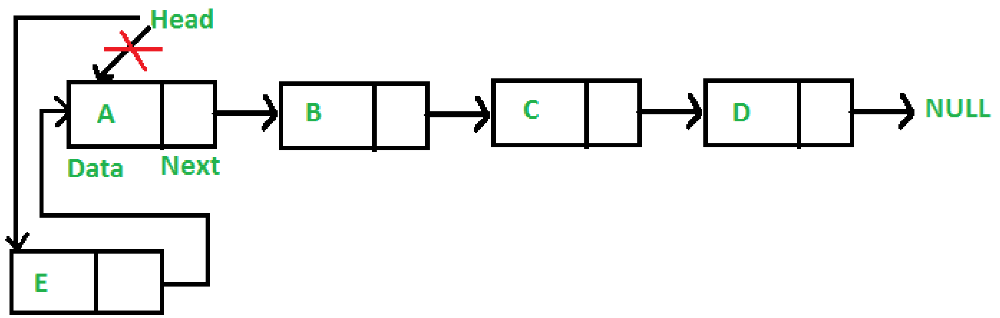
Example of Memory Stack and Heap

- Stack_Heap_Link

Insertion – At the front

Check the top answer(answer with most votes) in this LINK to see why head is passed as a double pointer.

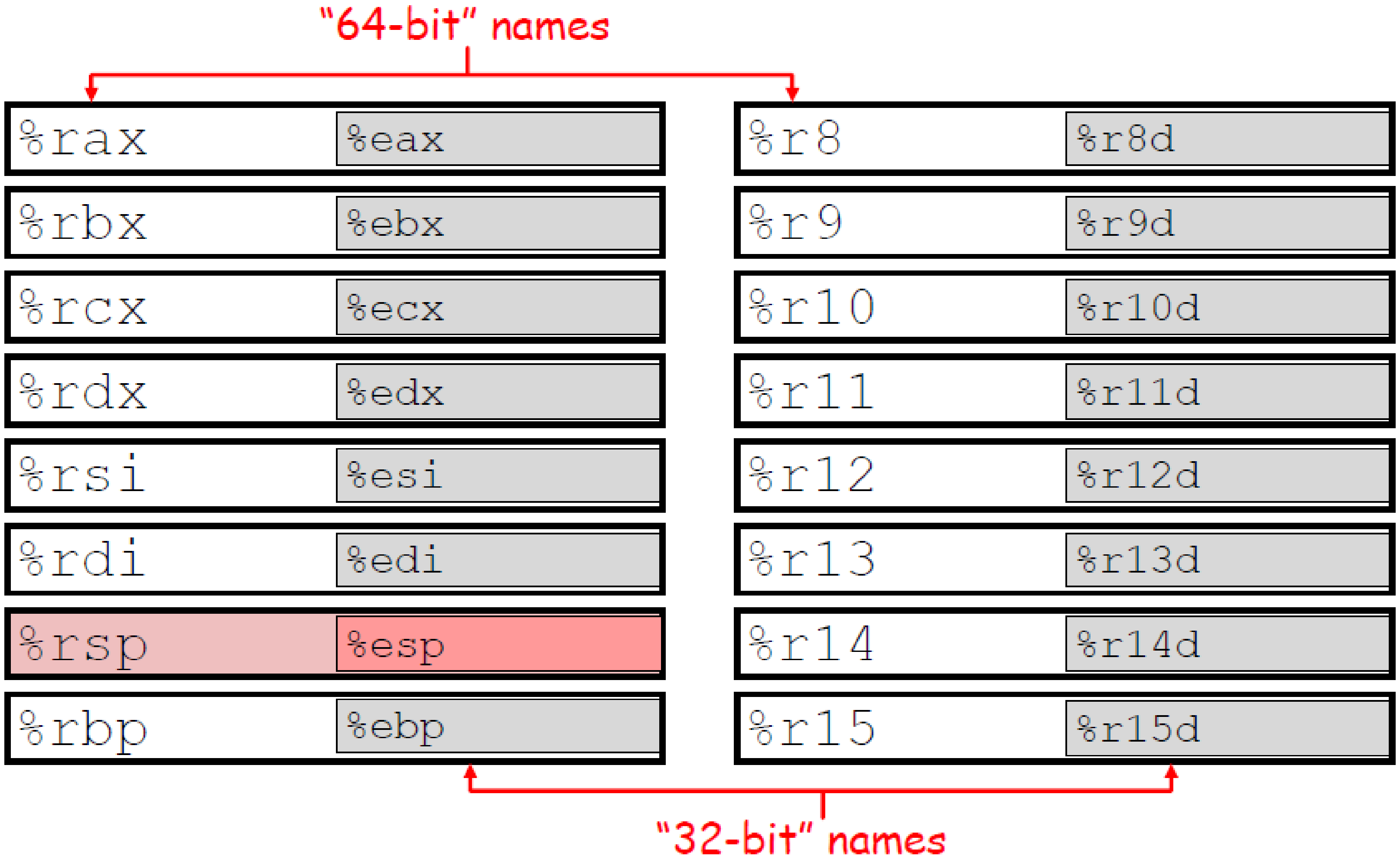
```
void insert_front(struct Node** head, int new_data)
{
/* 1. allocate node */
struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

/* 2. put in the data  */
new_node->data  = new_data;

/* 3. Make next of new node as head */
new_node->next = (*head);

/* 4. move the head to point to the new node */
(*head) = new_node;
}
```

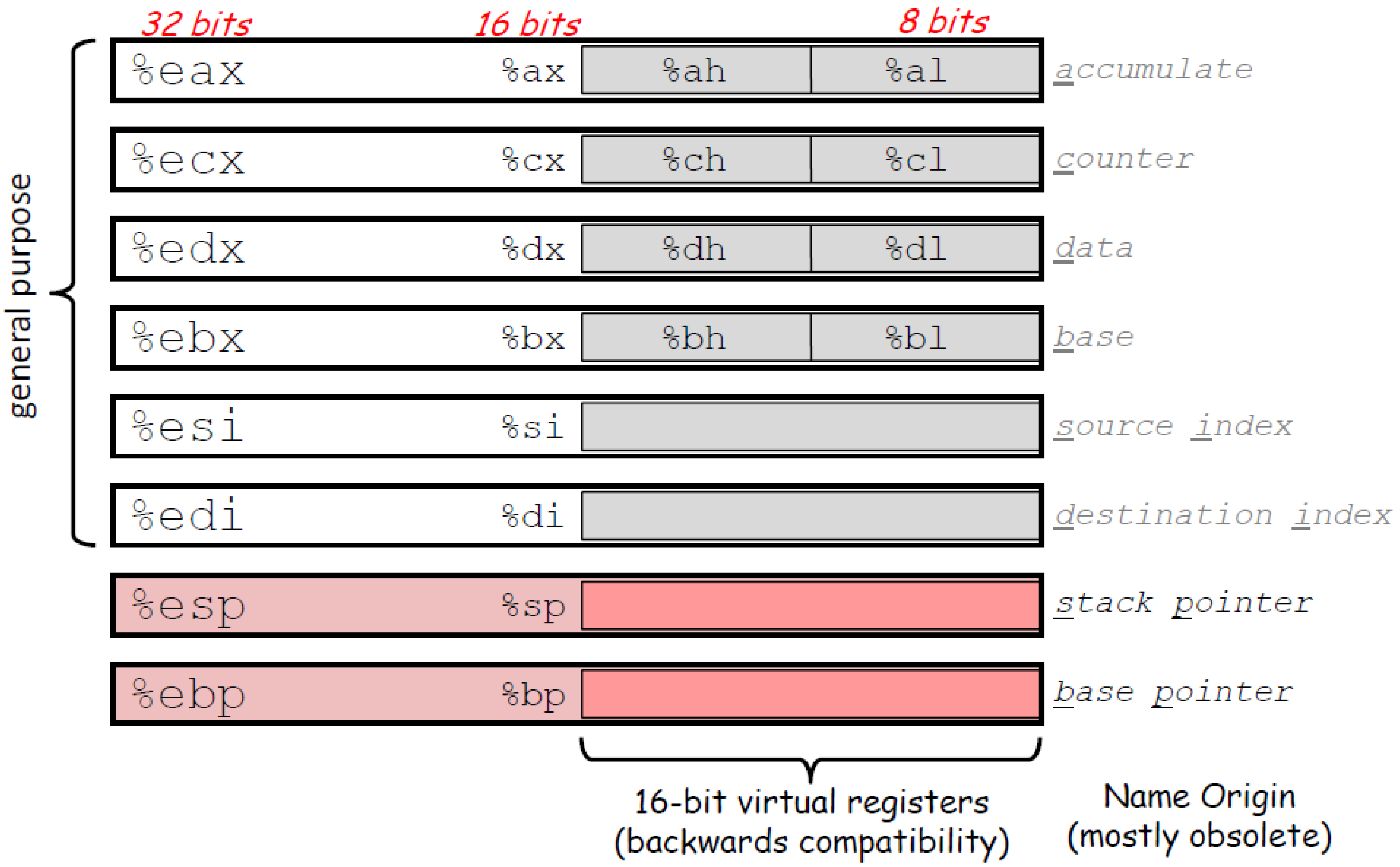


Registers

- A register is a location within the processor that is able to store data
  - Names, not addresses
  - Much faster than DRAM
  - Can hold any value: addresses, values from operations, characters etc.
  - Usually, register
    - `%rip` stores the address of the next instruction
    - `%rsp` is used as a stack pointer
    - `%rax` holds the return value from a function
  - A register in x86-64 is 64 bits wide
    - ‘The lower 32-, 16- and 8-bit portions are selectable by a pseudo-register name’.

https://ctf101.org/binary-exploitation/what-are-registers/
https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html


"64-bit" names
%rax
%eax
%rbx
%ebx
%rcx
%ecx
%rdx
%edx
%rsi
%esi
%rdi
%edi
%rsp
%esp
%rbp
%ebp
%r8
%r8d
%r9
%r9d
%r10
%r10d
%r11
%r11d
%r12
%r12d
%r13
%r13d
%r14
%r14d
%r15
%r15d
"32-bit" names



32 bits
16 bits
8 bits
general purpose
%eax
%ax
%ah
%al
accumulate
%ecx
%cx
%ch
%cl
counter
%edx
%dx
%dh
%dl
data
%ebx
%bx
%bh
%bl
base
%esi
%si
source index
%edi
%di
destination index
%esp
%sp
stack pointer
%ebp
%bp
base pointer
16-bit virtual registers
(backwards compatibility)
Name Origin
(mostly obsolete)

mov

General form: `mov_ source, destination`

- `movb src, dst`
  Move 1-byte "byte"
- `movw src, dst`
  Move 2-byte "word"
- `movl src, dst`
  Move 4-byte "long word"
- `movq src, dst`
  Move 8-byte "quad word"

- movq src, dst # general form of instruction dst = src
- movl $0, %eax # %eax = 0
- movq %rax, $100 #Invalid!! destination cannot be an immediate value
- movsbl %al, %edx # copy 1-byte %al, sign-extend into 4-byte %edx
- movzbl %al, %edx # copy 1-byte %al, zero-extend into 4-byte %edx

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html
Dr Petrucci's slides - "Intro to x86-64"

Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	`movq $0x4, %rax`	`var_a = 0x4;`
		Mem	`movq $-147, (%rax)`	`*p_a = -147;`
	Reg	Reg	`movq %rax, %rdx`	`var_d = var_a;`
		Mem	`movq %rax, (%rdx)`	`*p_d = var_a;`
	Mem	Reg	`movq (%rax), %rdx`	`var_d = *p_a;`

Addressing Modes

- ‘An addressing mode is an expression that calculates an address in memory to be read/written to.’

- **General** expression
  - D (Rb, Ri, S) = Mem[Reg[Rb]+Reg[Ri]*S+D]

Addressing Modes - Example

- `movq %rdi, 0x568892` # direct (address is constant value)
- `movq %rdi, (%rax)` # indirect (address is in register %rax)
- mov (%rsi), %rdi #%rdi = Mem[%rsi]
- `movq %rdi,-24(%rbp)` # indirect with displacement (address = %rbp -24)
- `movq %rsi, 8(%rsp, %rdi, 4)`

indirect with displacement and scaled-index (address = 8 + %rsp + %rdi*4)

- `movq %rsi, 0x4(%rax, %rcx)` #Mem[0x4 + %rax +%rcx*1] = %rsi
- `movq %rsi, 0x8(, %rdx, 4)` #Mem(0x8 + %rdx*4) = %rsi

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html

lea

- `leaq src, dst`
  - "`lea`" stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
  - `dst` is a register
  - Sets `dst` to the *address* computed by the `src` expression (does not go to memory! - it just does math)
  - Example: `leaq (%rdx,%rcx,4), %rax`

lea

- lea or Load effective address
  - Does not dereference the source address, it simply calculates its location.
  - leaq 0x20(%rsp), %rdi   # %rdi = %rsp + 0x20 (no dereference!)
  - leaq (%rdi,%rdx,1), %rax   # %rax = %rdi + %rdx * 1

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html

Exercise 1

```
int compound(int y)
{
    int *t3 = &y;
    *t3 = *t3 + *t3;

    return *t3;
}
```

```
compound:
        ____________________
        ret
```

Exercise 1

```
int compound(int y)
{
    int *t3 = &y;
    *t3 = *t3 + *t3;

    return *t3;
}
```

```
compound:
        leal (%rdi,%rdi), %eax
        ret
```

Exercise 2

```
int compound(int x, int y)
{
  int a[2] = {x, y};
  int z = a[0];
  int y1 = z + a[1];
  int *y2 = &y1;

  return *y2;
}
```

```
compound:

    ________________
    ret
```

Exercise 2

```c
int compound(int x, int y)
{
  int a[2] = {x, y};
  int z = a[0];
  int y1 = z + a[1];
  int *y2 = &y1;

  return *y2;
}
```

```
compound:
        leal (%rdi,%rsi), %eax
        ret
```

Example

```
long compound(long *xp, long *yp, long *zp)
{


    long t0 = *xp;
    long t1 = *yp;
    long t10 = *zp;
    long t15 = 15;
    long t2 = t0 + t1 + t10 - t15;
    long t3 = t2 + t2;

    return t2;
}
```

```
compound:
      movq (%rsi), %rax
      addq (%rdi), %rax
      addq (%rdx), %rax
      subq     $15, %rax
      ret
```

Exercise 3

```
int pointer1(int *x)
{
    return *(x+1);
}
```

```
int* pointer2(int **x)
{
    return *(x+1);
}
```

```
pointer1(int*):
        ____________________
        ret
```

```
pointer2(int*):
        ____________________
        ret
```

Exercise 3

```
int pointer1(int *x)
{
    return *(x+1);
}
```

```
pointer1(int*):
        movl    4(%rdi), %eax
        ret
```

```
int* pointer2(int **x)
{
    return *(x+1);
}
```

```
pointer2(int*):
        movq    8(%rdi), %rax
        ret
```

What is the C equivalent?

x86-64 code

```
arith(int, int, int):
        movl    %edi, %ecx
        imull   %esi, %ecx
        imull   %edx, %edi
        imull   %edx, %esi
        leal    (%rcx,%rdi), %eax
        addl    %esi, %eax
        ret
```

Intermediate Pseudo Code 1

```
arith(int x, int y, int z)
{
      ecx = edi
      ecx = ecx * esi
      edi = edi * edx
      esi = esi * edx
      eax = edi + ecx
      eax = eax + esi
      return eax
}
```

What is the C equivalent?

Intermediate Pseudo Code 1

```
arith(int x, int y, int z)
{
      ecx = edi
      ecx = ecx * esi
      edi = edi * edx
      esi = esi * edx
      eax = edi + ecx
      eax = eax + esi
      return eax
}
```

Intermediate Pseudo Code 2

```
arith(int x, int y, int z)
{
      ecx = x
      ecx = x * y
      edi = x * z
      esi = y * z
      eax = x * z + x * y
      eax = eax + y * z
      return eax
}
```

What is the C equivalent?

Intermediate Pseudo Code 2

```
arith(int x, int y, int z)
{
	ecx = x
	ecx = x * y
	edi = x * z
	esi = y * z
	eax = x * z + x * y
	eax = eax + y * z
	return eax
}
```

C code

```
int arith (int x, int y, int z)
{
	return x*y + x*z + y*z;
}
```

Example

x86-64 code

```
test(int*, int*, int**):
      movl 4(%rsi), %eax
      addl 4(%rdi), %eax
      movq -8(%rdx), %rdx
      addl (%rdx), %eax
      ret
```

Intermediate Pseudocode

```
test(int *x, int *y, int **z)
{
      eax = *(y+1)
      eax = *(x+1) + *(y+1)
      rdx = *(z-1)
      eax =  *(x+1) + *(y+1) + **(z-1)
      return eax
}
```

Example

Intermediate Pseudocode

```
test(int *x, int *y, int **z)
{
        eax = *(y+1)
        eax = *(x+1) + *(y+1)
        rdx = *(z-1)
        eax =  *(x+1) + *(y+1) + **(z-1)
        return eax
}
```

C code

```
int test (int *x, int *y, int**z)
{
        return *(x+1)+*(y+1)+**(z-1);
}
```

Exercise 4

```
long type_cast(char a, int b, long c)
{
    ____________;
}
```

```
type_cast(char, int, long):
        movsbl  %dil, %edi
        addl    %edi, %esi
        movslq  %esi, %rsi
        leaq    (%rsi,%rdx), %rax
        ret
```

Exercise 4

```
long type_cast(char a, int b, long c)
{
  return a+b+c;
}
```

```
type_cast(char, int, long):
        movsbl  %dil, %edi
        addl    %edi, %esi
        movslq  %esi, %rsi
        leaq    (%rsi,%rdx), %rax
        ret
```

Exercise 4.1

```
long type_cast(unsigned char a,
int b, long c)
{
        return a+b+c;
}
```

```
type_cast(unsigned char, int, long):
        movzbl  %dil, %edi
        addl    %edi, %esi
        movslq  %esi, %rsi
        leaq    (%rsi,%rdx), %rax
        ret
```

Demo

- Demo of using gdb and objdump
    - Refer to the video in Panopto.