# Assembly Language:
# Part 2

# Goals of this Lecture

Help you learn:

- Intermediate aspects of x86-64 assembly language…
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
- Structures

# Agenda

**Flattened C code**

Control flow with signed integers

Control flow with unsigned integers

Arrays

Structures

# Flattened C Code

## Problem

- Translating from C to assembly language is difficult when the C code contains **nested** statements

## Solution

- **Flatten** the C code to eliminate all nesting

# Flattened C Code

## C

```
if (expr)
{   statement1;
    …
    statementN;
}
```

```
if (expr)
{   statementT1;
    …
    statementTN;
}
else
{   statementF1;
    …
    statementFN;
}
```

## Flattened C

```
    if (! expr) goto endif1;
        statement1;
        …
        statementN;
endif1:
```

```
    if (! expr) goto else1;
        statementT1;
        …
        statementTN;
        goto endif1;
else1:
        statementF1;
        …
        statementFN;
endif1:
```

# Flattened C Code

## C

```
while (expr)
{   statement1;
    …
    statementN;
}
```

```
for (expr1; expr2; expr3}
{   statement1;
    …
    statementN;
}
```

## Flattened C

```
loop1:
   if (! expr) goto endloop1;
      statement1;
      …
      statementN;
      goto loop1;
endloop1:
```

```
   expr1;
loop1:
   if (! expr2) goto endloop1;
   statement1;
   …
   statementN;
   expr3;
   goto loop1;
endloop1:
```

See Bryant & O'Hallaron book for faster patterns

# Agenda

Flattened C code

**Control flow with signed integers**

Control flow with unsigned integers

Arrays

Structures

# `if` Example

C

```
int i;
…
if (i < 0)
   i = -i;
```

Flattened C

```
int i;
…
   if (i >= 0) goto endif1;
   i = -i;
endif1:
```

# `if` Example

## Flattened C

```
int i;
…
   if (i >= 0) goto endif1;
   i = -i;
endif1:
```

## Assem Lang

```
    .section ".bss"
i:  .skip 4
…
    .section ".text"
…
    cmpl $0, i
    jge  endif1
    negl i
endif1:
```

Note:

`cmp` instruction (counterintuitive operand order)

- Sets CC bits in EFLAGS register

`jge` instruction (conditional jump)

- Examines CC bits in EFLAGS register

# `if…else` Example

C

```
int i;
int j;
int smaller;
…
if (i < j)
   smaller = i;
else
   smaller = j;
```

Flattened C

```
int i;
int j;
int smaller;
…
   if (i >= j) goto else1;
   smaller = i;
   goto endif1;
else1:
   smaller = j;
endif1:
```

# `if`…`else` Example

## Flattened C

```
int i;
int j;
int smaller;
…
   if (i >= j) goto else1;
   smaller = i;
   goto endif1;
else1:
   smaller = j;
endif1:
```

Note:
`jmp` instruction
(unconditional jump)

## Assem Lang

```
    .section ".bss"
i:          .skip 4
j:          .skip 4
smaller:    .skip 4
…
    .section ".text"
…
    movl i, %eax
    cmpl j, %eax
    jge  else1
    movl i, %eax
    movl %eax, smaller
    jmp endif1
else1:
    movl j, %eax
    movl %eax, smaller
endif1:
```

# `while` Example

### C

```
int fact;
int n;
…
fact = 1;
while (n > 1)
{ fact *= n;
  n--;
}
```

### Flattened C

```
int fact;
int n;
…
   fact = 1;
loop1:
   if (n <= 1) goto endloop1;
   fact *= n;
   n--;
   goto loop1;
endloop1:
```

# `while` Example

## Flattened C

```
int fact;
int n;
…
   fact = 1;
loop1:
   if (n <= 1) goto endloop1;
   fact *= n;
   n--;
   goto loop1;
endloop1:
```

## Assem Lang

```
    .section ".bss"
fact: .skip 4
n:    .skip 4
…
    .section ".text"
…
   movl $1, fact
loop1:
   cmpl $1, n
   jle  endloop1
   movl fact, %eax
   imull n
   movl %eax, fact
   decl n
   jmp loop1
endloop1:
```

Note:

`jle` instruction (conditional jump)

`imul` instruction

# `for` Example

C

```
int power = 1;
int base;
int exp;
int i;
…
for (i = 0; i < exp; i++)
   power *= base;
```

Flattened C

```
int power = 1;
int base;
int exp;
int i;
…
   i = 0;
loop1:
   if (i >= exp) goto endloop1;
   power *= base;
   i++;
   goto loop1;
endloop1:
```

# `for` Example

## Flattened C

```
int power = 1;
int base;
int exp;
int i;
…
   i = 0;
loop1:
   if (i >= exp) goto endloop1;
   power *= base;
   i++;
   goto loop1;
endloop1:
```

## Assem Lang

```
    .section ".data"
power: .long 1
    .section ".bss"
base:  .skip 4
exp:   .skip 4
i:     .skip 4
…
    .section ".text"
…
   movl $0, i
loop1:
   movl i, %eax
   cmpl exp, %eax
   jge  endloop1
   movl power, %eax
   imull base
   movl %eax, power
   incl i
   jmp loop1
endloop1:
```

# Control Flow with Signed Integers

## Comparing signed integers

| | |
|---|---|
| `cmp{q,l,w,b} srcIRM, destRM` | Compare dest with src |

- Sets condition-code bits in the EFLAGS register
- Beware: operands are in counterintuitive order
- Beware: many other instructions set condition-code bits
  - Conditional jump should **immediately** follow `cmp`

# Control Flow with Signed Integers

## Unconditional jump

```
jmp label   Jump to label
```

## Conditional jumps after comparing signed integers

```
je  label   Jump to label if equal
jne label   Jump to label if not equal
jl  label   Jump to label if less
jle label   Jump to label if less or equal
jg  label   Jump to label if greater
jge label   Jump to label if greater or equal
```

- Examine CC bits in EFLAGS register

# Agenda

Flattened C

Control flow with signed integers

**Control flow with unsigned integers**

Arrays

Structures

# Signed vs. Unsigned Integers

## In C

- Integers are signed or unsigned
- Compiler generates assem lang instructions accordingly

## In assembly language

- Integers are neither signed nor unsigned
- Distinction is in the instructions used to manipulate them

## Distinction matters for

- Multiplication and division
- Control flow

# Handling Unsigned Integers

## Multiplication and division

- Signed integers: **`imul`**, **`idiv`**
- Unsigned integers: **`mul`**, **`div`**

## Control flow

- Signed integers: **`cmp`** + {**`je`**, **`jne`**, **`jl`**, **`jle`**, **`jg`**, **`jge`**}

Unsigned integers: "unsigned cmp" + {**`je`**, **`jne`**, **`jl`**, **`jle`**, **`jg`**, **`jge`**} ? No!!!

- Unsigned integers: **`cmp`** + {**`je`**, **`jne`**, **`jb`**, **`jbe`**, **`ja`**, **`jae`**}

# `while` Example

C

```
unsigned int fact;
unsigned int n;
…
fact = 1;
while (n > 1)
{ fact *= n;
  n--;
}
```

Flattened C

```
unsigned int fact;
unsigned int n;
…
   fact = 1;
loop1:
   if (n <= 1) goto endloop1;
   fact *= n;
   n--;
   goto loop1;
endloop1:
```

# `while` Example

## Flattened C

```
unsigned int fact;
unsigned int n;
…
   fact = 1;
loop1:
   if (n <= 1) goto endloop1;
   fact *= n;
   n--;
   goto loop1;
endloop1:
```

## Assem Lang

```
 .section ".bss"
fact: .skip 4
n:    .skip 4
…
   .section ".text"
…
   movl $1, fact
loop1:
   cmpl $1, n
   jbe  endloop1
   movl fact, %eax
   mull n
   movl %eax, fact
   decl n
   jmp loop1
endloop1:
```

Note:

`jbe` instruction (instead of `jle`)

`mull` instruction (instead of `imull`)

# `for` Example

C

```
unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
…
for (i = 0; i < exp; i++)
  power *= base;
```

Flattened C

```
unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
…
   i = 0;
loop1:
   if (i >= exp) goto endloop1;
   power *= base;
   i++;
   goto loop1;
endloop1:
```

# `for` Example

## Flattened C

```
unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
…
   i = 0;
loop1:
   if (i >= exp) goto endloop1;
   power *= base;
   i++;
   goto loop1;
endloop1:
```

## Assem Lang

```
 .section ".data"
power: .long 1
   .section ".bss"
base:  .skip 4
exp:   .skip 4
i:     .skip 4
…
   .section ".text"
…
   movl $0, i
loop1:
   movl i, %eax
   cmpl exp, %eax
   jae  endloop1
   movl power, %eax
   mull base
   movl %eax, power
   incl i
   jmp loop1
endloop1:
```

Note:

`jae` instruction (instead of `jge`)

`mull` instruction (instead of `imull`)

# Control Flow with Unsigned Integers

## Comparing unsigned integers

```
cmp{q,l,w,b} srcIRM, destRM          Compare dest with src
```

(Same as comparing signed integers)

## Conditional jumps after comparing unsigned integers

```
je  label   Jump to label if equal
jne label   Jump to label if not equal
jb  label   Jump to label if below
jbe label   Jump to label if below or equal
ja  label   Jump to label if above
jae label   Jump to label if above or equal
```

- Examine CC bits in EFLAGS register

# Agenda

Flattened C

Control flow with signed integers

Control flow with unsigned integers

**Arrays**

Structures

# Arrays: Indirect Addressing

C

```
int a[100];
int i;
int n;
…
i = 3;
…
n = a[i]
…
```

Assem Lang

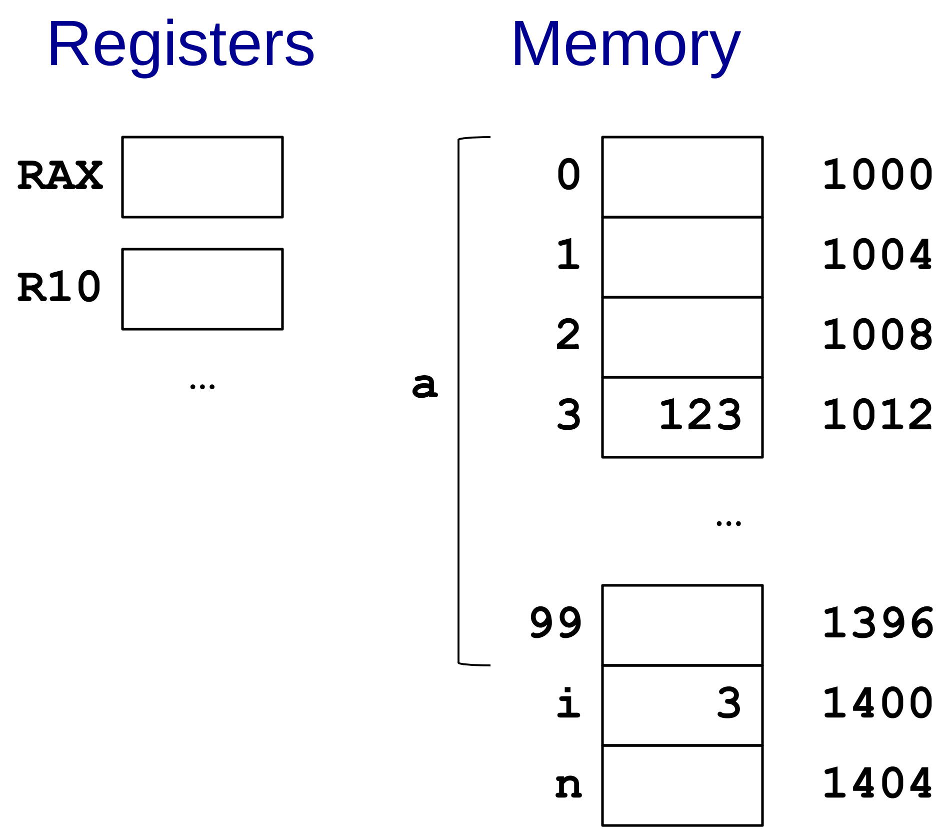
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movslq i, %rax
    salq $2, %rax
    addq $a, %rax
    movl (%rax), %r10d
    movl %r10d, n
    …
```

One step at a time…

# Arrays: Indirect Addressing

## Assem Lang

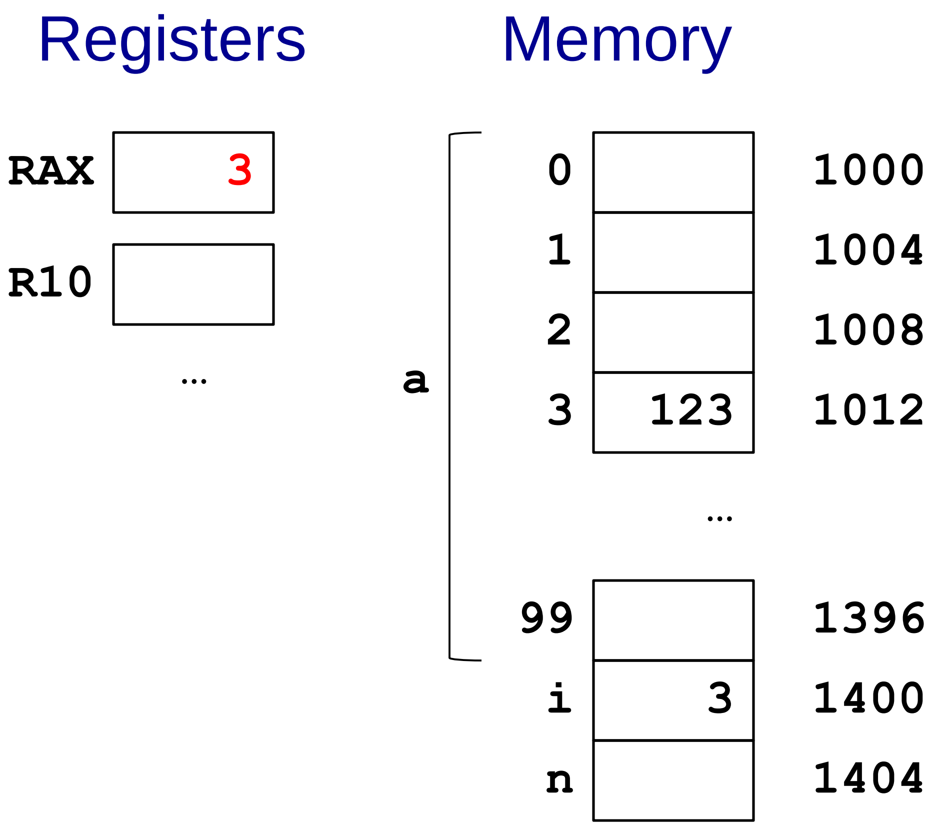
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movslq i, %rax
    salq $2, %rax
    addq $a, %rax
    movl (%rax), %r10d
    movl %r10d, n
    …
```



# Arrays: Indirect Addressing

## Assem Lang

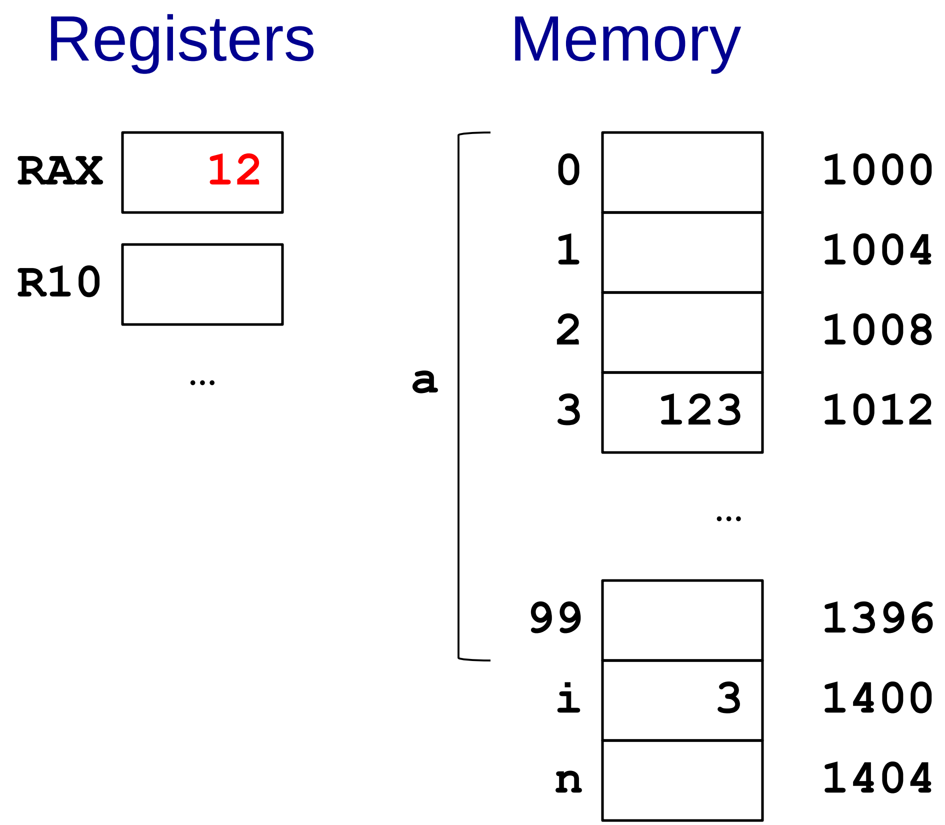
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movslq i, %rax
    salq $2, %rax
    addq $a, %rax
    movl (%rax), %r10d
    movl %r10d, n
    …
```



# Arrays: Indirect Addressing

## Assem Lang

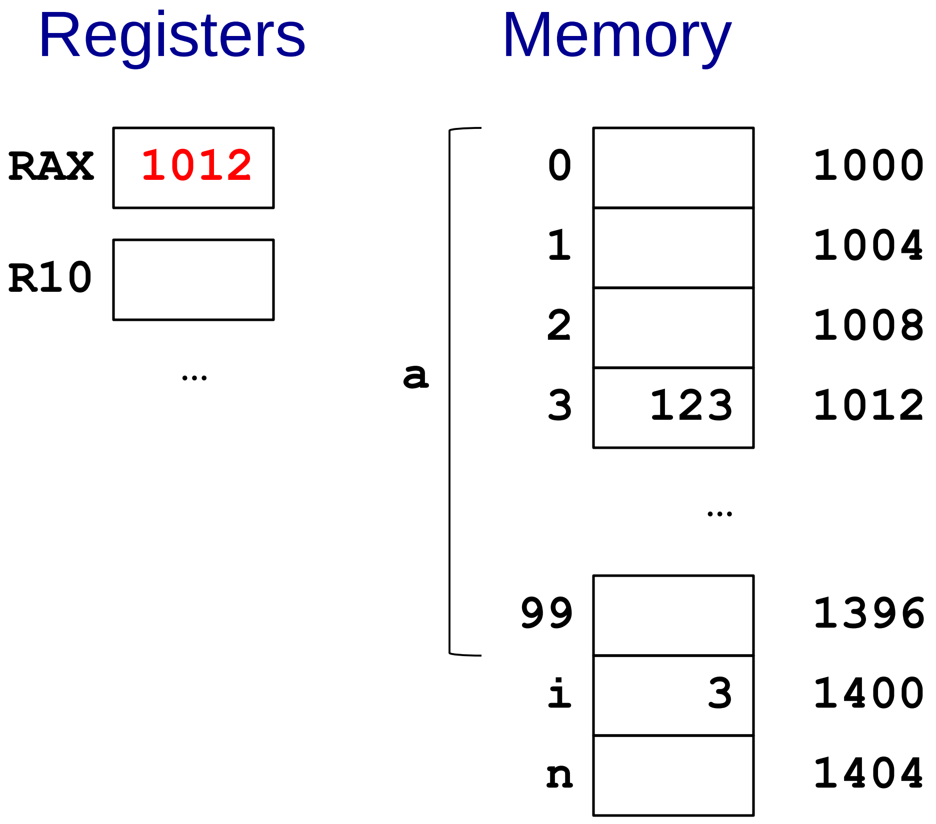
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movslq i, %rax
    salq $2, %rax
    addq $a, %rax
    movl (%rax), %r10d
    movl %r10d, n
    …
```



# Arrays: Indirect Addressing

## Assem Lang

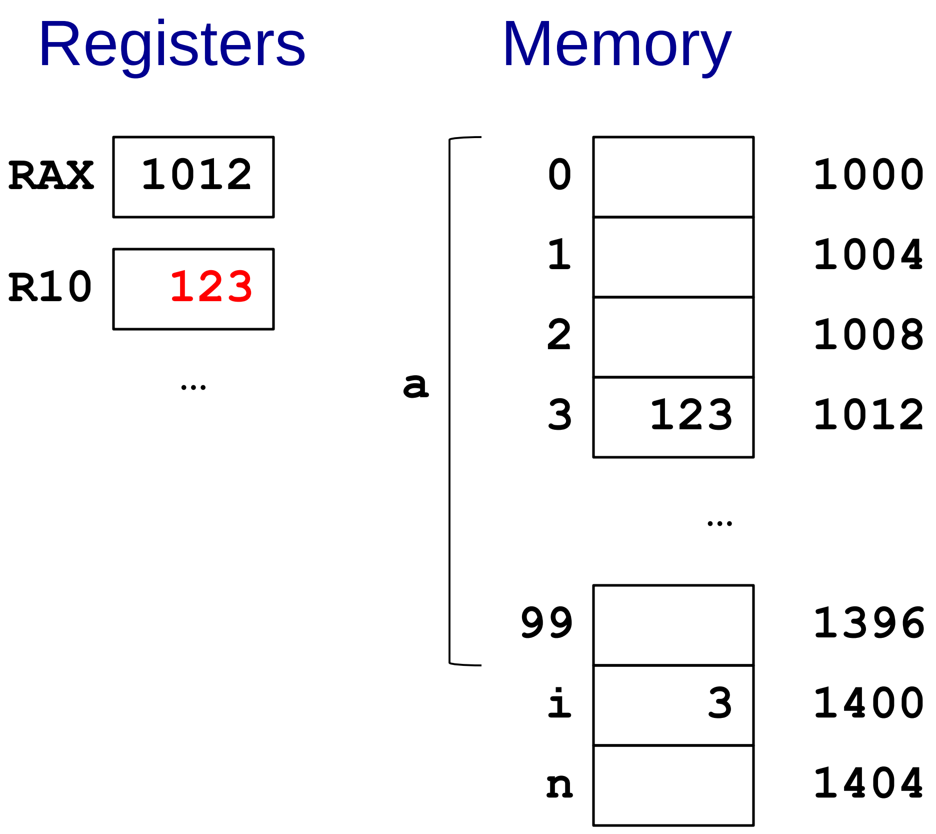
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movslq i, %rax
    salq $2, %rax
    addq $a, %rax
    movl (%rax), %r10d
    movl %r10d, n
    …
```



# Arrays: Indirect Addressing

## Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movslq i, %rax
    salq $2, %rax
    addq $a, %rax
    movl (%rax), %r10d
    movl %r10d, n
    …
```



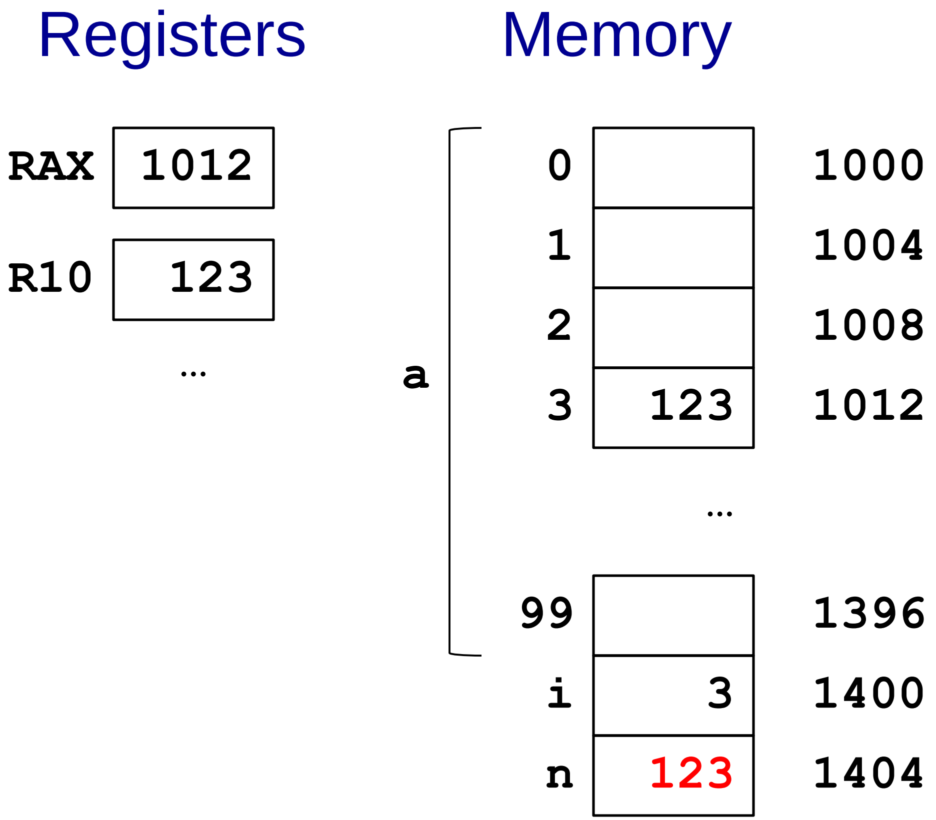

Note:

**Indirect** addressing

# Arrays: Indirect Addressing

## Assem Lang

```
        .section ".bss"
a:      .skip 400
i:      .skip 4
n:      .skip 4
        …
        .section ".text"
        …
        movl $3, i
        …
        movslq i, %rax
        salq $2, %rax
        addq $a, %rax
        movl (%rax), %r10d
        movl %r10d, n
        …
```



# Arrays: Base+Disp Addressing

C

```
int a[100];
int i;
int n;
…
i = 3;
…
n = a[i]
…
```

Assem Lang

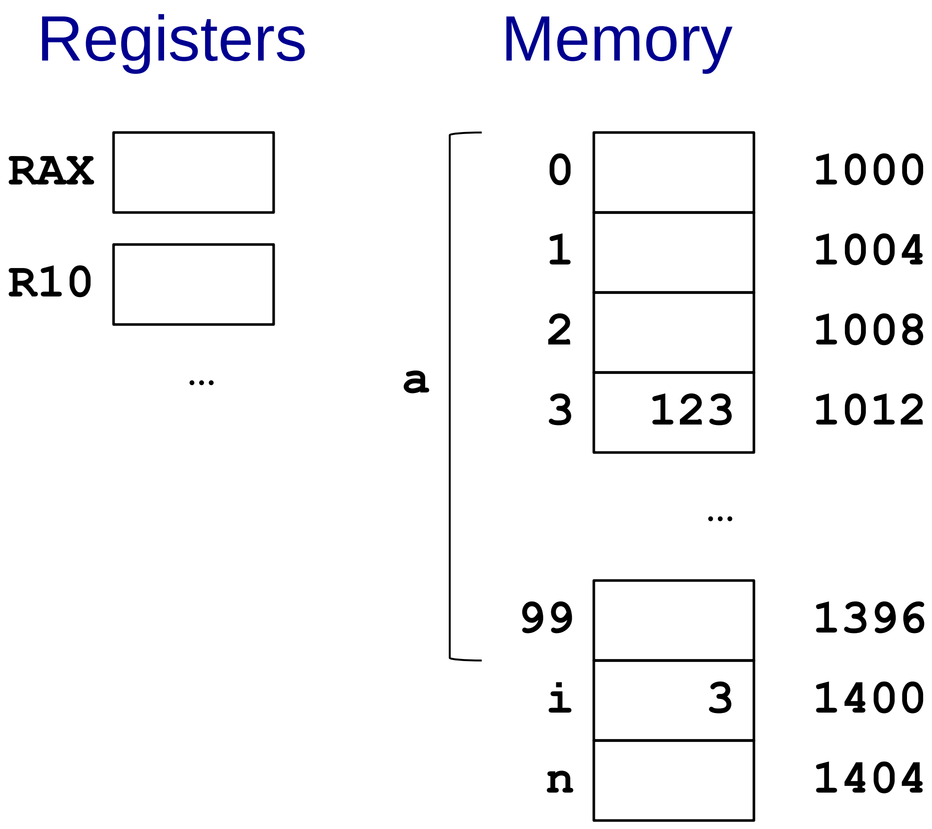
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    sall $2, %eax
    movl a(%eax), %r10d
    movl %r10d, n
    …
```

One step at a time…

# Arrays: Base+Disp Addressing

## Assem Lang

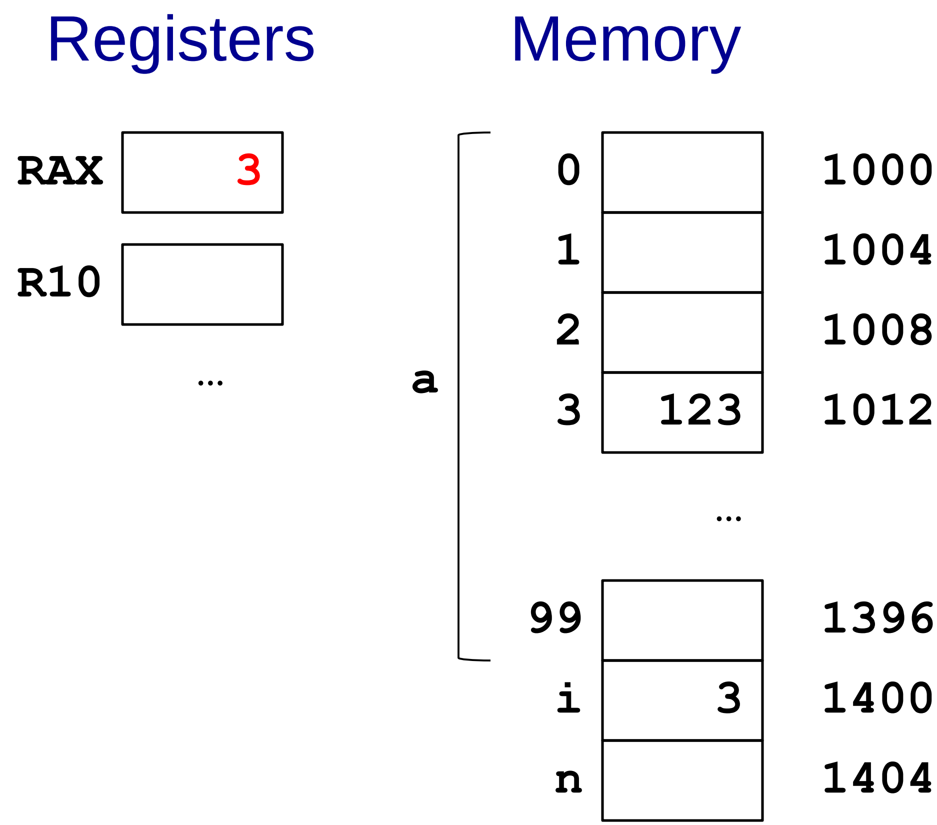
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    sall $2, %eax
    movl a(%eax), %r10d
    movl %r10d, n
    …
```



# Arrays: Base+Disp Addressing

## Assem Lang

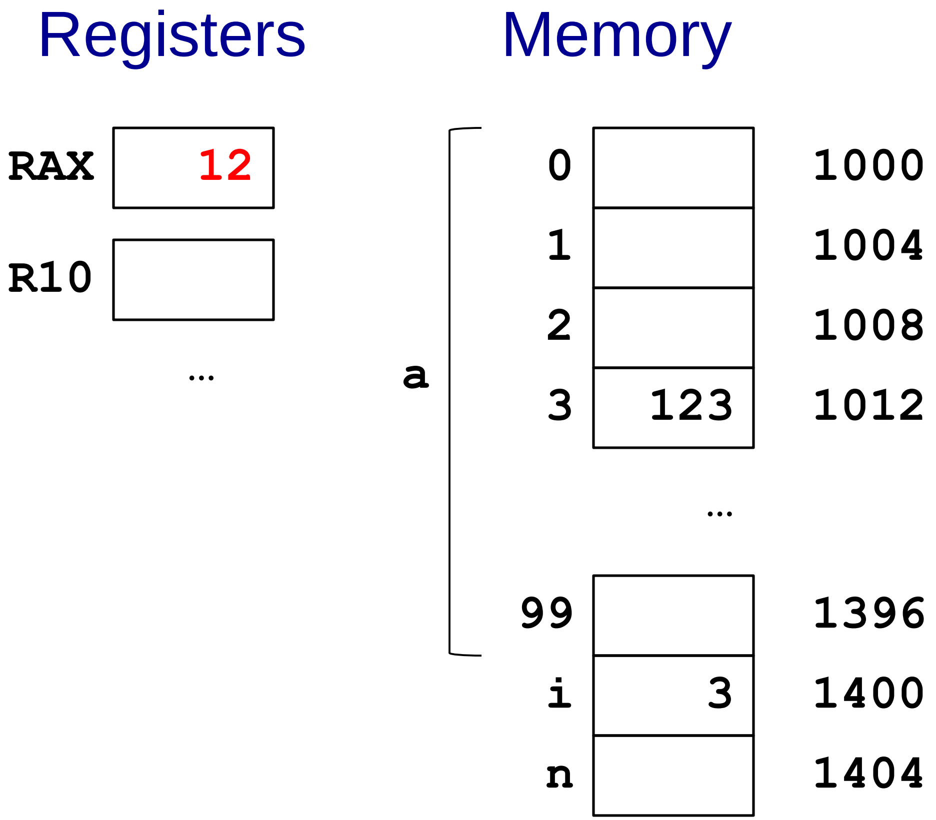
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    sall $2, %eax
    movl a(%eax), %r10d
    movl %r10d, n
    …
```



# Arrays: Base+Disp Addressing

## Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    sall $2, %eax
    movl a(%eax), %r10d
    movl %r10d, n
    …
```



# Arrays: Base+Disp Addressing

## Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    sall $2, %eax
    movl a(%eax), %r10d
    movl %r10d, n
    …
```

## Registers

| | |
|---|---|
| RAX | 12 |
| R10 | 123 |
| … | |

## Memory

| | | | |
|---|---|---|---|
| a | 0 | | 1000 |
| | 1 | | 1004 |
| | 2 | | 1008 |
| | 3 | 123 | 1012 |
| | | … | |
| | 99 | | 1396 |
| i | | 3 | 1400 |
| n | | | 1404 |

Note:

**Base+displacement** addressing

# Arrays: Base+Disp Addressing

## Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    sall $2, %eax
    movl a(%eax), %r10d
    movl %r10d, n
    …
```

## Registers

| | |
|---|---|
| RAX | 12 |
| R10 | 123 |
| … | |

## Memory

| | | | |
|---|---|---|---|
| a | 0 | | 1000 |
| | 1 | | 1004 |
| | 2 | | 1008 |
| | 3 | 123 | 1012 |
| | | … | |
| | 99 | | 1396 |
| | i | 3 | 1400 |
| | n | 123 | 1404 |

# Arrays: Scaled Indexed Addressing

C

```
int a[100];
int i;
int n;
…
i = 3;
…
n = a[i]
…
```

Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    movl a(,%eax,4), %r10d
    movl %r10d, n
    …
```

One step at a time…

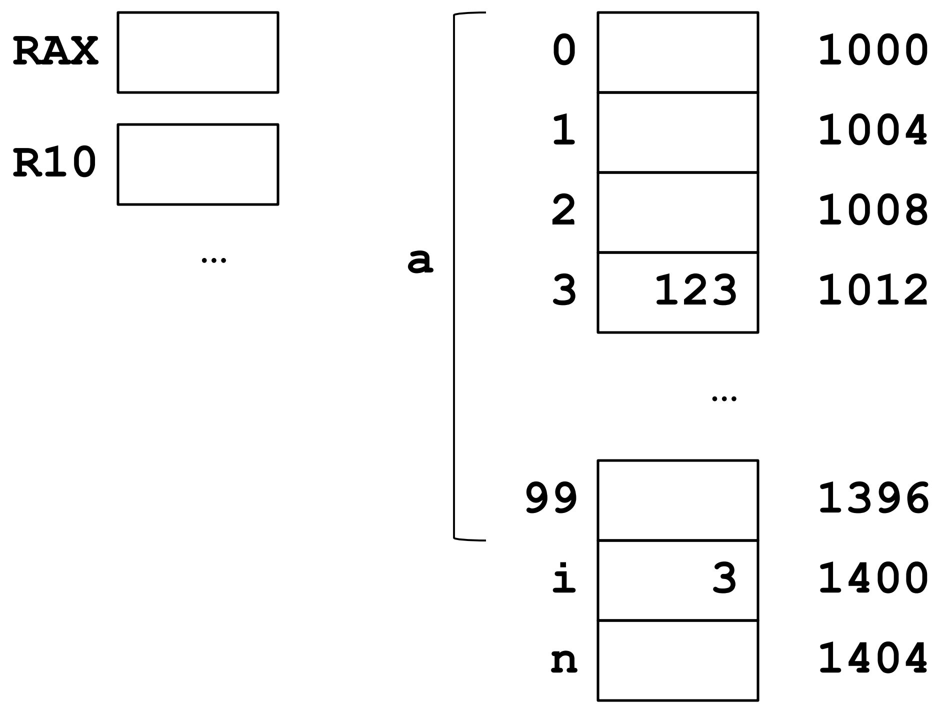
# Arrays: Scaled Indexed Addressing

## Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    movl a(,%eax,4), %r10d
    movl %r10d, n
    …
```

## Registers

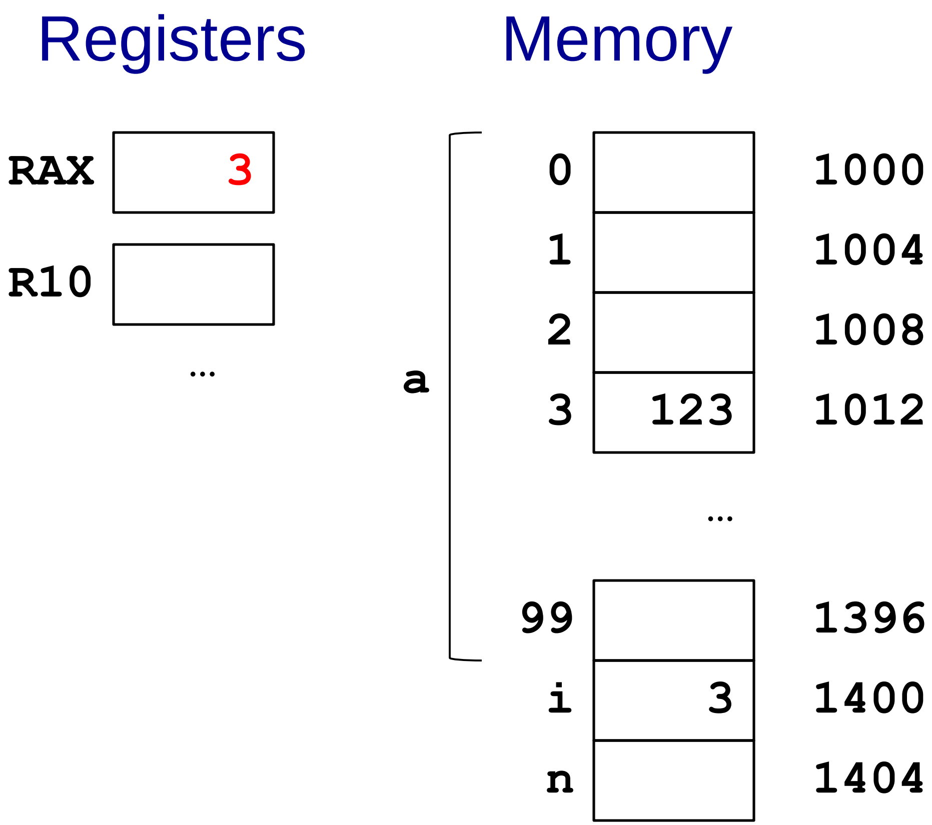
## Memory



# Arrays: Scaled Indexed Addressing

## Assem Lang

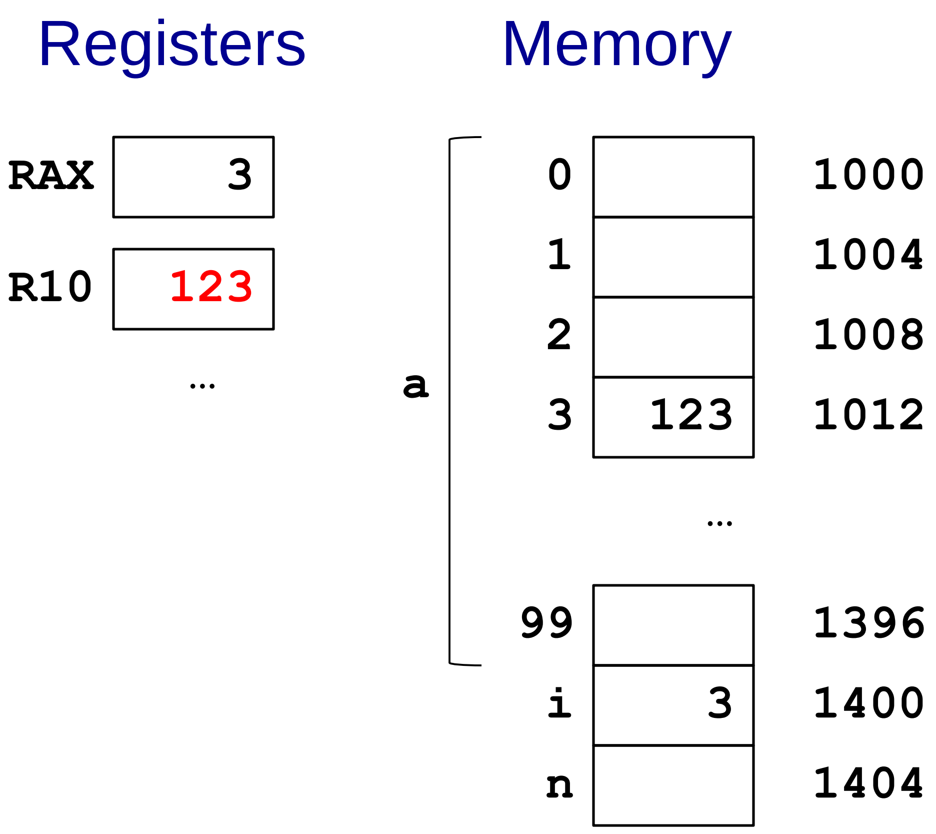
```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    movl a(,%eax,4), %r10d
    movl %r10d, n
    …
```



# Arrays: Scaled Indexed Addressing

## Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    movl a(,%eax,4), %r10d
    movl %r10d, n
    …
```




Note:

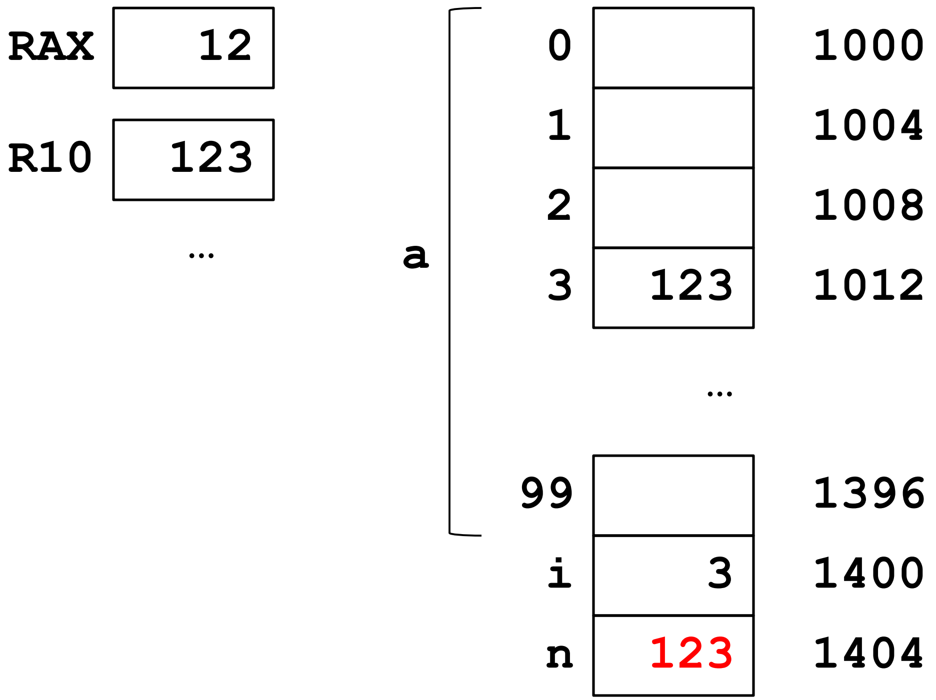
**Scaled indexed** addressing

# Arrays: Scaled Indexed Addressing

## Assem Lang

```
    .section ".bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
    …
    .section ".text"
    …
    movl $3, i
    …
    movl i, %eax
    movl a(,%eax,4), %r10d
    movl %r10d, n
    …
```

Registers Memory



# Generalization: Memory Operands

Full form of memory operands:

**`displacement(base,index,scale)`**

- **`displacement`** is an integer or a label (default = 0)
- **`base`** is a 4-byte or 8-byte register
- **`index`** is a 4-byte or 8-byte register
- **`scale`** is 1, 2, 4, or 8 (default = 1)

Meaning

- Compute the sum
  (displacement) + (contents of base) + ((contents of index) * (scale))
- Consider the sum to be an address
- Load from (or store to) that address

Note:

- All other forms are subsets of the full form…

# Generalization: Memory Operands

Valid subsets:

- **Direct addressing**
  - `displacement`
- **Indirect addressing**
  - `(base)`
- **Base+displacement addressing**
  - `displacement(base)`
- **Indexed addressing**
  - `(base, index)`
  - `displacement(base,index)`
- **Scaled indexed addressing**
  - `(,index, scale)`
  - `displacement(,index,scale)`
  - `(base,index,scale)`
  - `displacement(base,index,scale)`

# Operand Examples

## Immediate operands

- **`$5`** ⇒ use the number 5 (i.e. the number that is available immediately within the instruction)
- **`$i`** ⇒ use the address denoted by i (i.e. the address that is available immediately within the instruction)

## Register operands

- **`%rax`** ⇒ read from (or write to) register RAX

## Memory operands: **direct addressing**

- **`5`** ⇒ load from (or store to) memory at address 5 (silly; seg fault)
- **`i`** ⇒ load from (or store to) memory at the address denoted by i

## Memory operands: **indirect addressing**

- **`(%rax)`** ⇒ consider the contents of RAX to be an address; load from (or store to) that address

# Operand Examples

Memory operands: **base+displacement addressing**

- **`5(%rax)`** ⇒ compute the sum (5) + (contents of RAX); consider the sum to be an address; load from (or store to) that address
- **`i(%rax)`** ⇒ compute the sum (address denoted by i) + (contents of RAX); consider the sum to be an address; load from (or store to) that address

Memory operands: **indexed addressing**

- **`5(%rax,%r10)`** ⇒ compute the sum (5) + (contents of RAX) + (contents of R10); consider the sum to be an address; load from (or store to) that address
- **`i(%rax,%r10)`** ⇒ compute the sum (address denoted by i) + (contents of RAX) + (contents of R10); consider the sum to be an address; load from (or store to) that address

# Operand Examples

Memory operands: **scaled indexed addressing**

- **`5(%rax,%r10,4)`** ⇒ compute the sum (5) + (contents of RAX) + ((contents of R10) * 4); consider the sum to be an address; load from (or store to) that address
- **`i(%rax,%r10,4)`** ⇒ compute the sum (address denoted by i) + (contents of RAX) + ((contents of R10) * 4); consider the sum to be an address; load from (or store to) that address

# Aside: The `lea` Instruction

**`lea`**: **l**oad **e**ffective **a**ddress

- Unique instruction: suppresses memory load/store

Example

- **`movq 5(%rax), %r10`**
  - Compute the sum (5) + (contents of RAX); consider the sum to be an address; load 8 bytes from that address into R10
- **`leaq 5(%rax), %r10`**
  - Compute the sum (5) + (contents of RAX); move that sum to R10

Useful for

- Computing an address, e.g. as a function argument
  - See precept code that calls **`scanf()`**
- Some quick-and-dirty arithmetic

# iClicker Question

Q: What is the effect of the following code?

```
leaq (%rax,%rax,4),%rax
```

A. Move the quad at 4*(contents of RAX) to RAX
B. Move the quad at 5*(contents of RAX) to RAX
C. Multiply RAX by 4
D. Multiply RAX by 5
E. Double RAX and add 4

# Agenda

Flattened C

Control flow with signed integers

Control flow with unsigned integers

Arrays

**Structures**

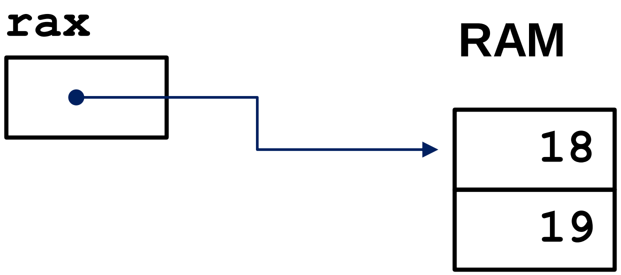
# Structures: Indirect Addressing

## C

```
struct S
{ int i;
  int j;
};
…
struct S myStruct;
…
myStruct.i = 18;
…
myStruct.j = 19;
```

## Assem Lang

```
   .section ".bss"
myStruct: .skip 8
   …
   .section ".text"
   …
   movq $myStruct, %rax
   movl $18, (%rax)
   …
   movq $myStruct, %rax
   addq $4, %rax
   movl $19, (%rax)
```




Note:

Indirect addressing

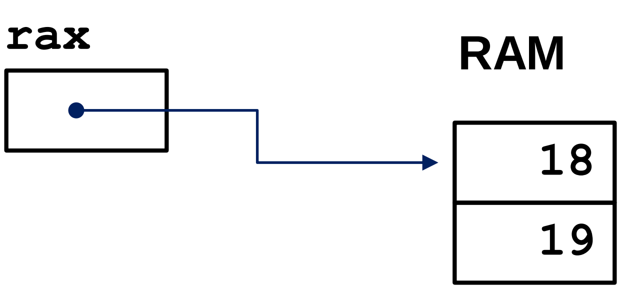
# Structures: Base+Disp Addressing

## C

```
struct S
{ int i;
  int j;
};
…
struct S myStruct;
…
myStruct.i = 18;
…
myStruct.j = 19;
```

## Assem Lang

```
    .section ".bss"
myStruct: .skip 8
    …
    .section ".text"
    …
    movq $myStruct, %rax
    movl $18, 0(%rax)
    …
    movl $19, 4(%rax)
```



# Structures: Padding

C

```
struct S
{ char c;
  int i;
};
…
struct S myStruct;
…
myStruct.c = 'A';
…
myStruct.i = 18;
```

Three-byte pad here

Assem Lang

```
    .section ".bss"
myStruct: .skip 8
    …
    .section ".text"
    …
    movq $myStruct, %rax
    movb $'A', 0(%rax)
    …
    movl $18, 4(%rax)
```

Beware:

Compiler sometimes inserts padding after fields

# Structures: Padding

## x86-64/Linux rules

| Data type | Within a struct, must begin at address that is evenly divisible by: |
|---|---|
| (unsigned) char | 1 |
| (unsigned) short | 2 |
| (unsigned) int | 4 |
| (unsigned) long | 8 |
| float | 4 |
| double | 8 |
| long double | 16 |
| any pointer | 8 |

- Compiler may add padding after last field if struct is within an array

# Summary

Intermediate aspects of x86-64 assembly language…

Flattened C code

Control transfer with signed integers

Control transfer with unsigned integers

Arrays

- Full form of instruction operands

Structures

- Padding

# Appendix

Setting and using CC bits in EFLAGS register

# Setting Condition Code Bits

## Question

- How does `cmp{q,l,w,b}` set condition code bits in EFLAGS register?

## Answer

- (See following slides)

# Condition Code Bits

Condition code bits

- **ZF**: **zero** flag: set to 1 iff result is **zero**
- **SF**: **sign** flag: set to 1 iff result is **negative**
- **CF**: **carry** flag: set to 1 iff **unsigned overflow** occurred
- **OF**: **overflow** flag: set to 1 iff **signed overflow** occurred

# Condition Code Bits

Example: **`addq src, dest`**

- Compute sum (**`dest`**+**`src`**)
- Assign sum to **`dest`**
- ZF: set to 1 iff sum == 0
- SF: set to 1 iff sum < 0
- CF: set to 1 iff unsigned overflow
  - Set to 1 iff sum<**`src`**
- OF: set if signed overflow
  - Set to 1 iff
    (**`src`**>0 && **`dest`**>0 && sum<0) ||
    (**`src`**<0 && **`dest`**<0 && sum>=0)

# Condition Code Bits

Example: **`subq src, dest`**

- Compute sum (**`dest`**+(-**`src`**))
- Assign sum to **`dest`**
- ZF: set to 1 iff sum == 0
- SF: set to 1 iff sum < 0
- CF: set to 1 iff unsigned overflow
  - Set to 1 iff **`dest`**<**`src`**
- OF: set to 1 iff signed overflow
  - Set to 1 iff
    (**`dest`**>0 && **`src`**<0 && sum<0) ||
    (**`dest`**<0 && **`src`**>0 && sum>=0)

Example: **`cmpq src, dest`**

- Same as **`subq`**
- But does not affect **`dest`**

# Using Condition Code Bits

## Question

- How do conditional jump instructions use condition code bits in EFLAGS register?

## Answer

- (See following slides)

# Conditional Jumps: Unsigned

After comparing **unsigned** data

| Jump Instruction | Use of CC Bits |
|---|---|
| je label | ZF |
| jne label | ~ZF |
| jb label | CF |
| jae label | ~CF |
| jbe label | CF \| ZF |
| ja label | ~(CF \| ZF) |

Note:

- If you can understand why `jb` jumps iff CF
- … then the others follow

# Conditional Jumps: Unsigned

Why does jb jump iff CF? Informal explanation:

(1) largenum – smallnum (not below)

- Correct result
- ⇒ CF=0 ⇒ don't jump

(2) smallnum – largenum (below)

- Incorrect result
- ⇒ CF=1 ⇒ jump

# Conditional Jumps: Signed

After comparing **signed** data

| Jump Instruction | Use of CC Bits |
|---|---|
| je label | ZF |
| jne label | ~ZF |
| jl label | OF ^ SF |
| jge label | ~(OF ^ SF) |
| jle label | (OF ^ SF) \| ZF |
| jg label | ~((OF ^ SF) \| ZF) |

Note:

- If you can understand why `jl` jumps iff OF^SF
- … then the others follow

# Conditional Jumps: Signed

Why does jl jump iff OF^SF? Informal explanation:

(1) largeposnum – smallposnum (not less than)

- Certainly correct result
- ⇒ OF=0, SF=0, OF^SF==0 ⇒ don't jump

(2) smallposnum – largeposnum (less than)

- Certainly correct result
- ⇒ OF=0, SF=1, OF^SF==1 ⇒ jump

(3) largenegnum – smallnegnum (less than)

- Certainly correct result
- ⇒ OF=0, SF=1 ⇒ (OF^SF)==1 ⇒ jump

(4) smallnegnum – largenegnum (not less than)

- Certainly correct result
- ⇒ OF=0, SF=0 ⇒ (OF^SF)==0 ⇒ don't jump

# Conditional Jumps: Signed

## (5) posnum – negnum (not less than)

- Suppose correct result
- ⇒ OF=0, SF=0 ⇒ (OF^SF)==0 ⇒ don't jump

## (6) posnum – negnum (not less than)

- Suppose incorrect result
- ⇒ OF=1, SF=1 ⇒ (OF^SF)==0 ⇒ don't jump

## (7) negnum – posnum (less than)

- Suppose correct result
- ⇒ OF=0, SF=1 ⇒ (OF^SF)==1 ⇒ jump

## (8) negnum – posnum (less than)

- Suppose incorrect result
- ⇒ OF=1, SF=0 ⇒ (OF^SF)==1 ⇒ jump