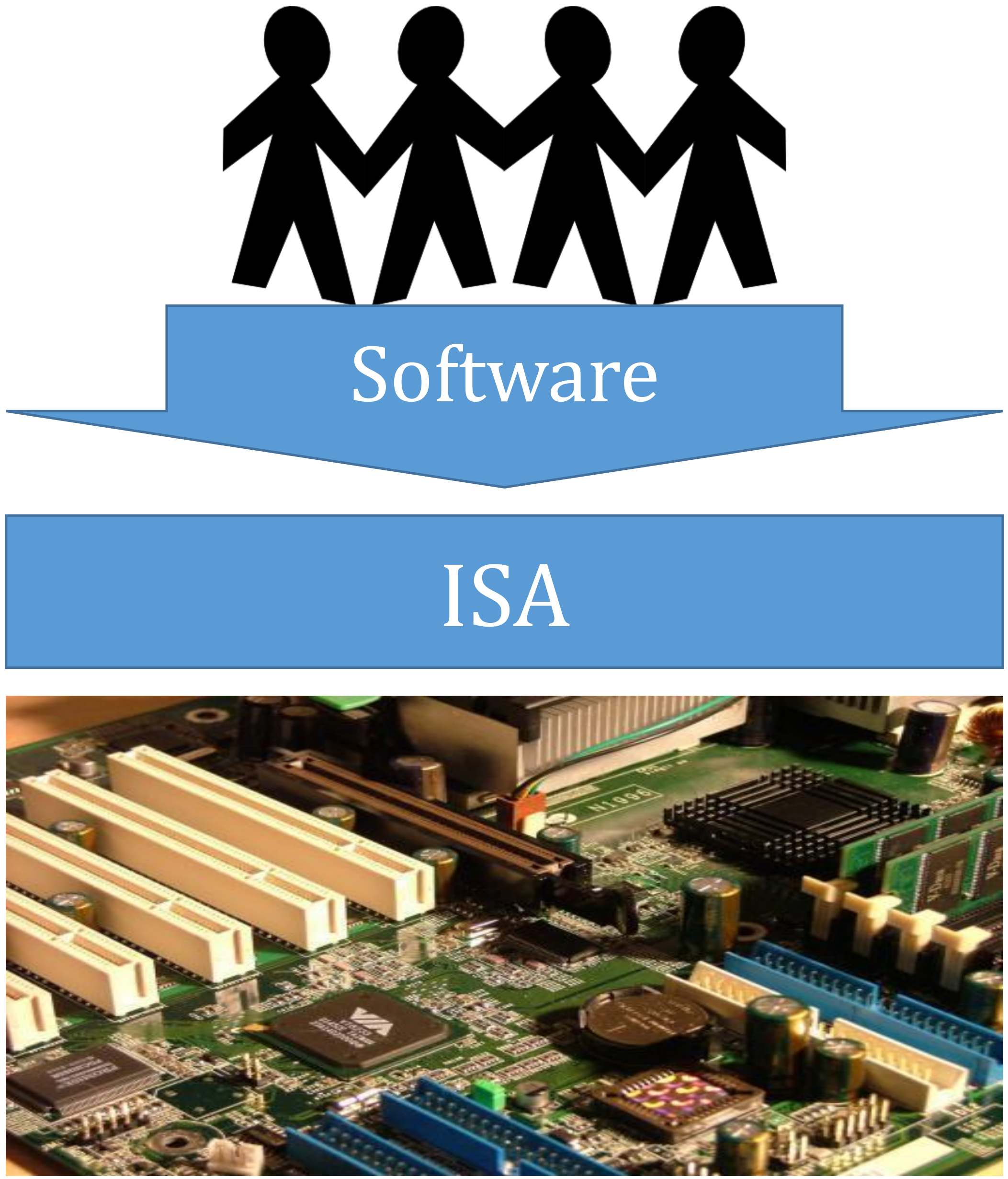
ISA Instructions

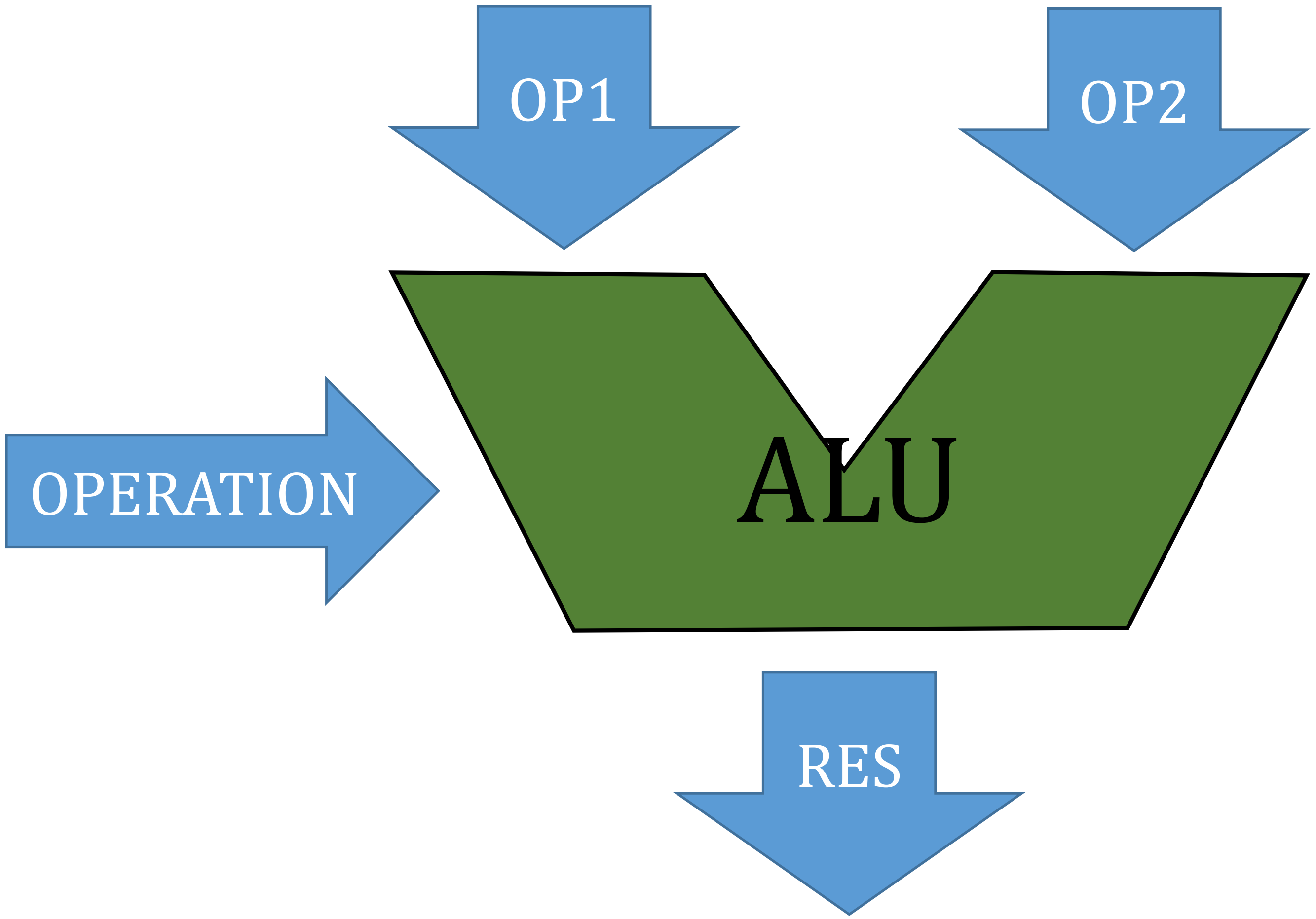
Computer Systems: Section 4.1

ISA Contents

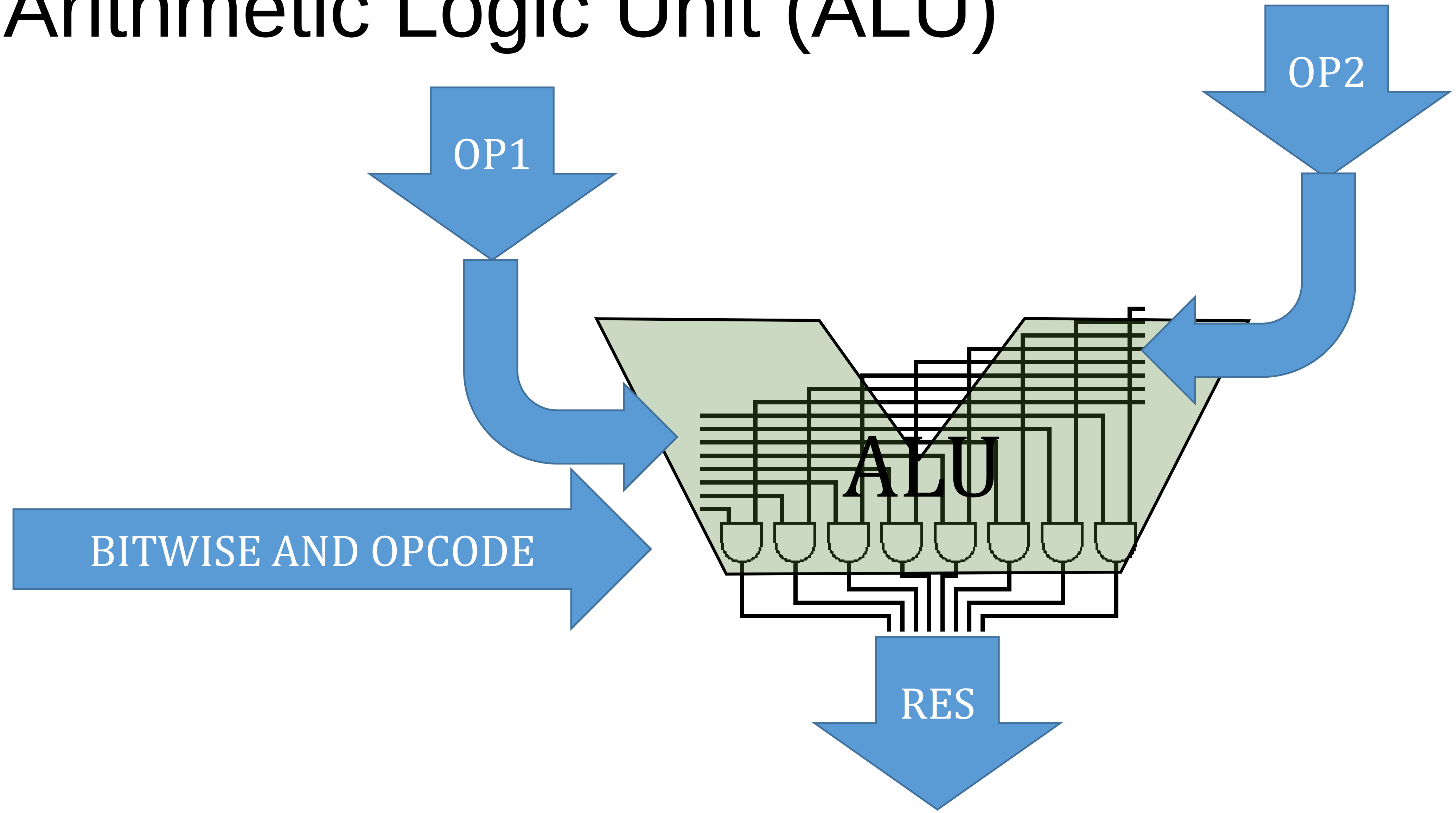
- The data types the instructions can work on
  - two's complement binary, ascii character, unsigned binary, etc.
- The instructions the hardware recognizes
  - add, move, get, ...
- The data the instructions can work on
  - Registers
  - Memory
- The external interfaces supported by the instructions
  - File I/O
  - Exception Handling and Interrupts



Arithmetic Logic Unit (ALU)

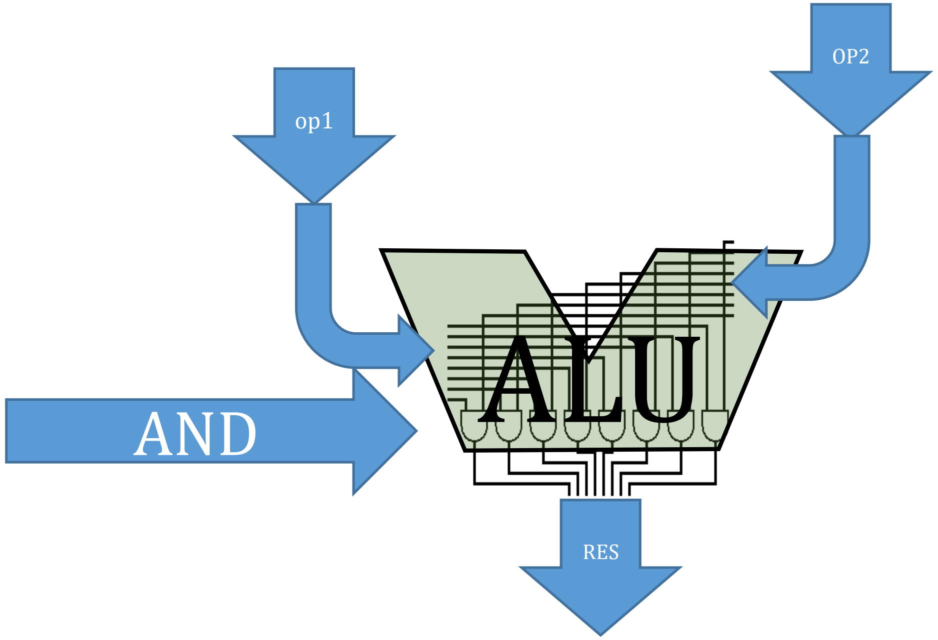


Arithmetic Logic Unit (ALU)



Instructions specify:

- Where to get the operand data
- What type of data are the operands
- What the ALU should do with those operands
- Where to put the results

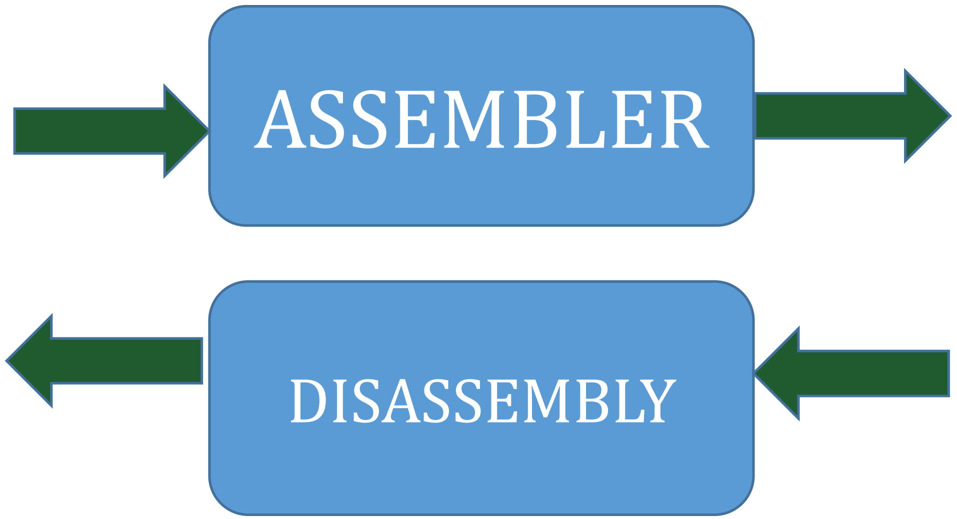


X86 Instructions

- Smallest (Atomic) directive to x86 "hardware"
- Consist of Opcode and Operands
- Two Flavors
  - Man-readable "Assembly"
  - Machine Readable "Object Code" or "Machine Code" or "Binary"
- Translation…

ASSEMBLY CODE

```
push %ebp
mov%esp,%ebp
and $0xfffffff0,%esp
sub $0xa0,%esp
mov0x8(%ebp),%eax
mov%eax,0x8(%esp)
```



OBJECT CODE

```
0x55
0x89e5
0x83e4f0
0x81eca0000000
0x8b4508
0x89442408
```

x86 Assembler Syntax

label: *mnemonic* arg1,*arg2*… ; *comment*

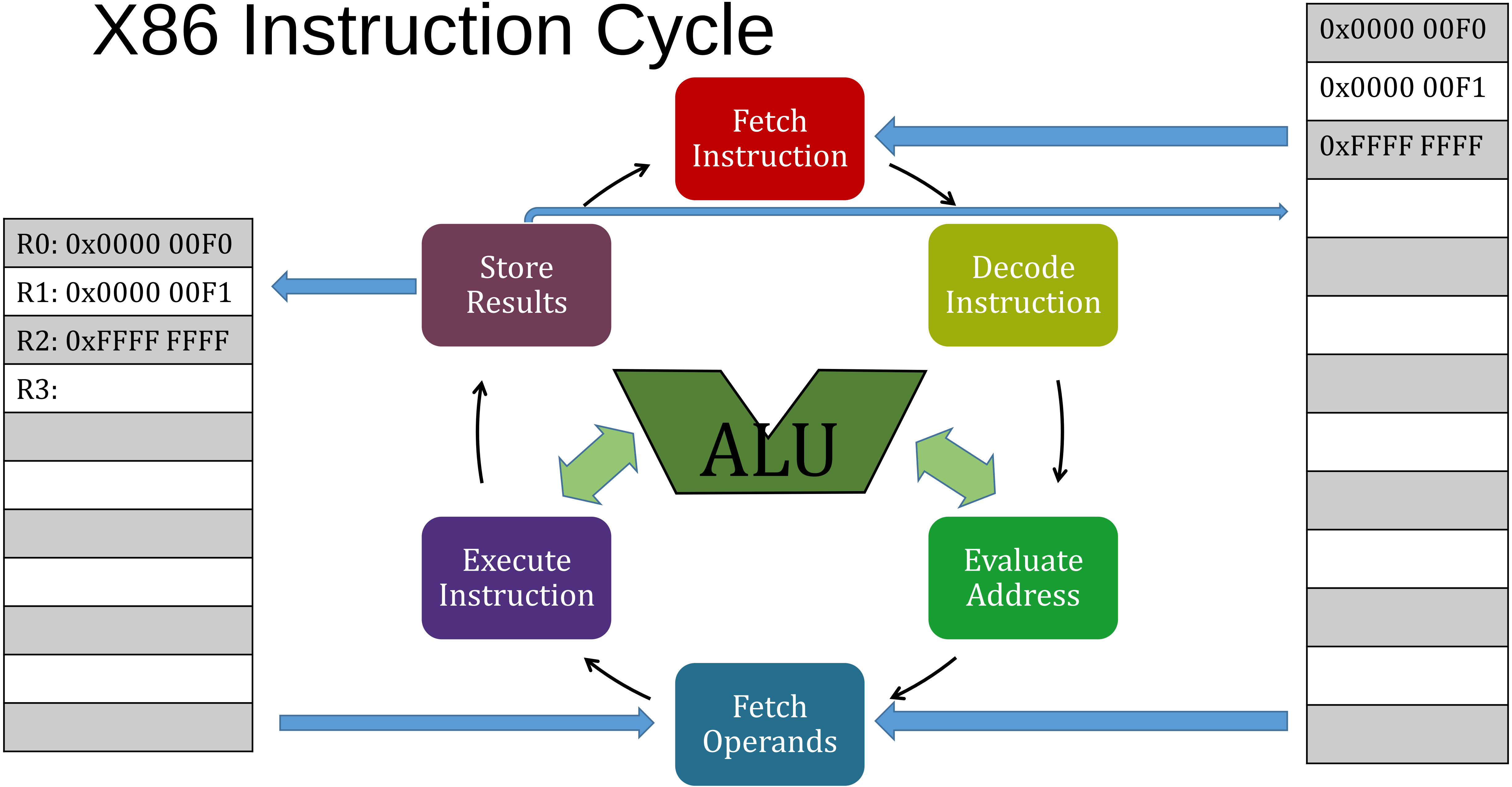
- *label* - optional – identifies start of this instruction
- *mnemonic* – See http://ref.x86asm.net/ for a complete list
- Up to 4 arguments
- Comment ends at the end of this line

```
and %ebx,%eax ; eax=eax & ebx
```

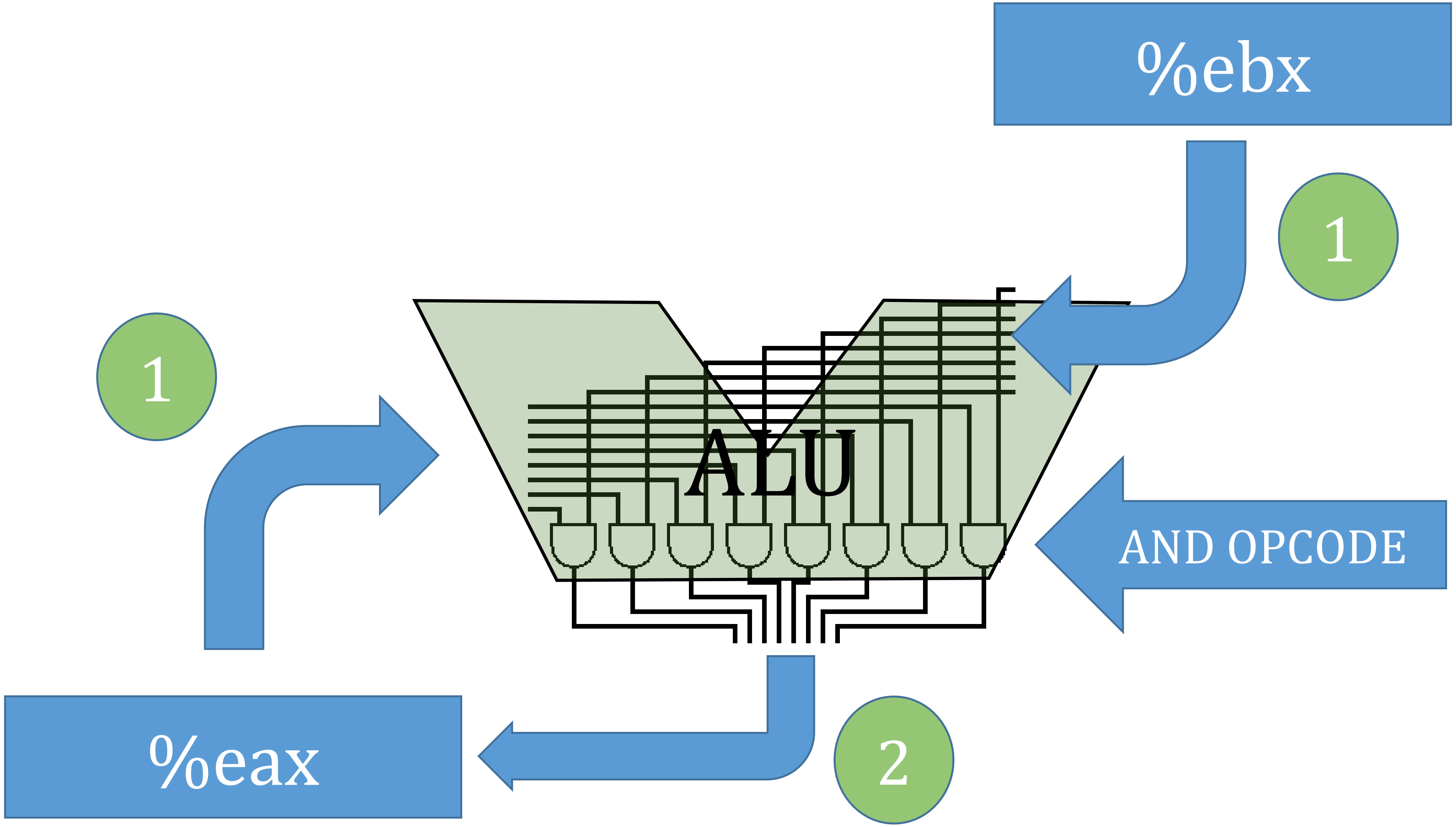
How instructions work

- Machine code consists of:
  - an “op-code” – one+ byte indicating what operation to perform
  - operand info – where / how to get operands and store the result – more in next lecture
- Instructions are stored in memory
- Hardware performs the instruction processing cycle for each instruction

X86 Instruction Cycle

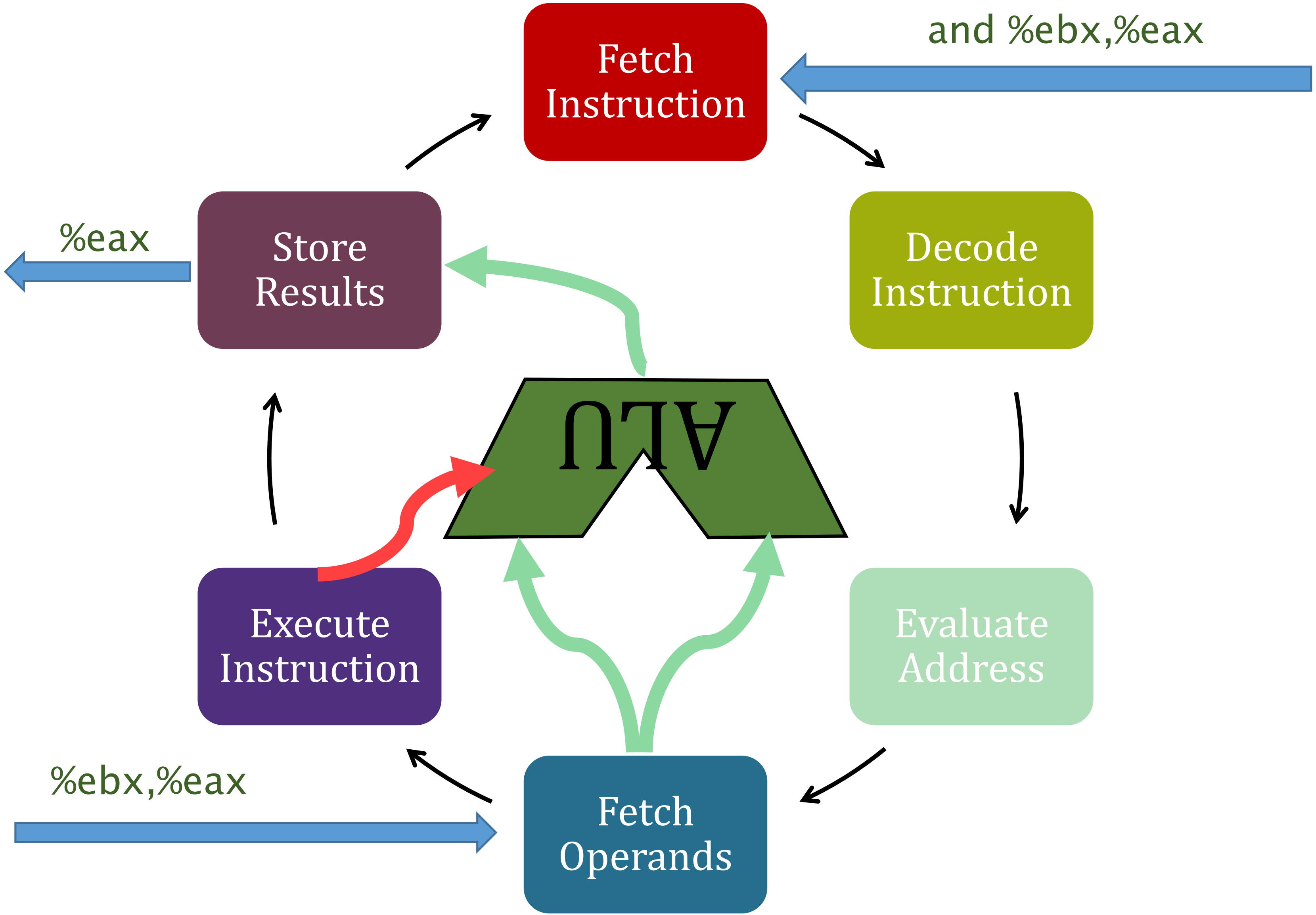


and %ebx,%eax ; in ALU



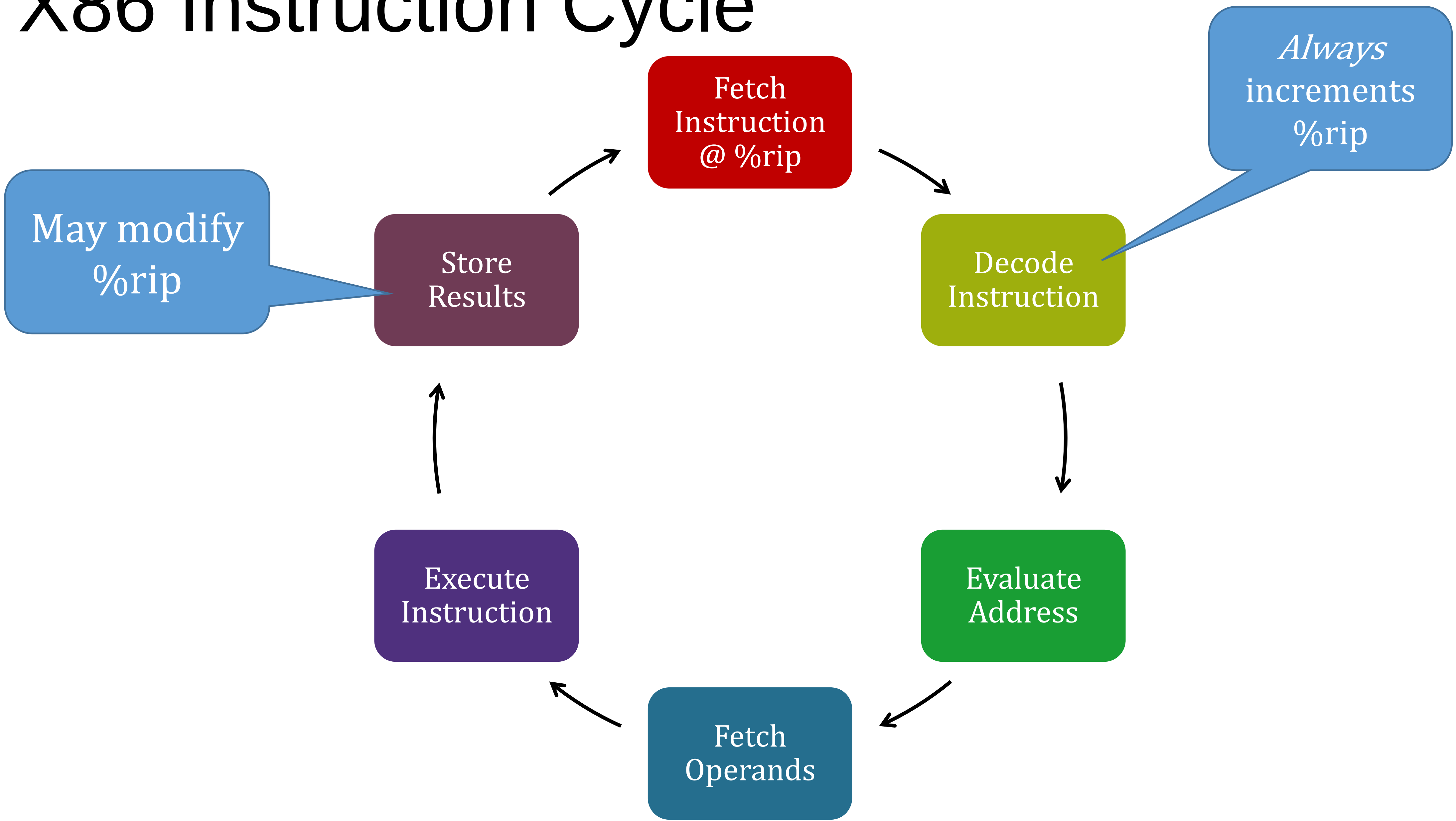
X86 Instruction Cycle Example

0x0000 00F0
0x0000 00F1
0xFFFF FFFF



R0: 0x0000 00F0
R1: 0x0000 00F1
R2: 0xFFFF FFFF
R3:

X86 Instruction Cycle



Instruction Results

- x86 convention – last argument is both operand and result
  - add %eax,%ebx; means %ebx = %eax + %ebx
  - Like the C statement: ebx+=eax;
- Warning: There are two dialects of x86 assembler, “AT&T” and “Intel”… we will be using the AT&T dialect
  - In AT&T dialect, the last argument is the target
  - In Intel dialect, the first argument is the target

Assembler Argument Generalities

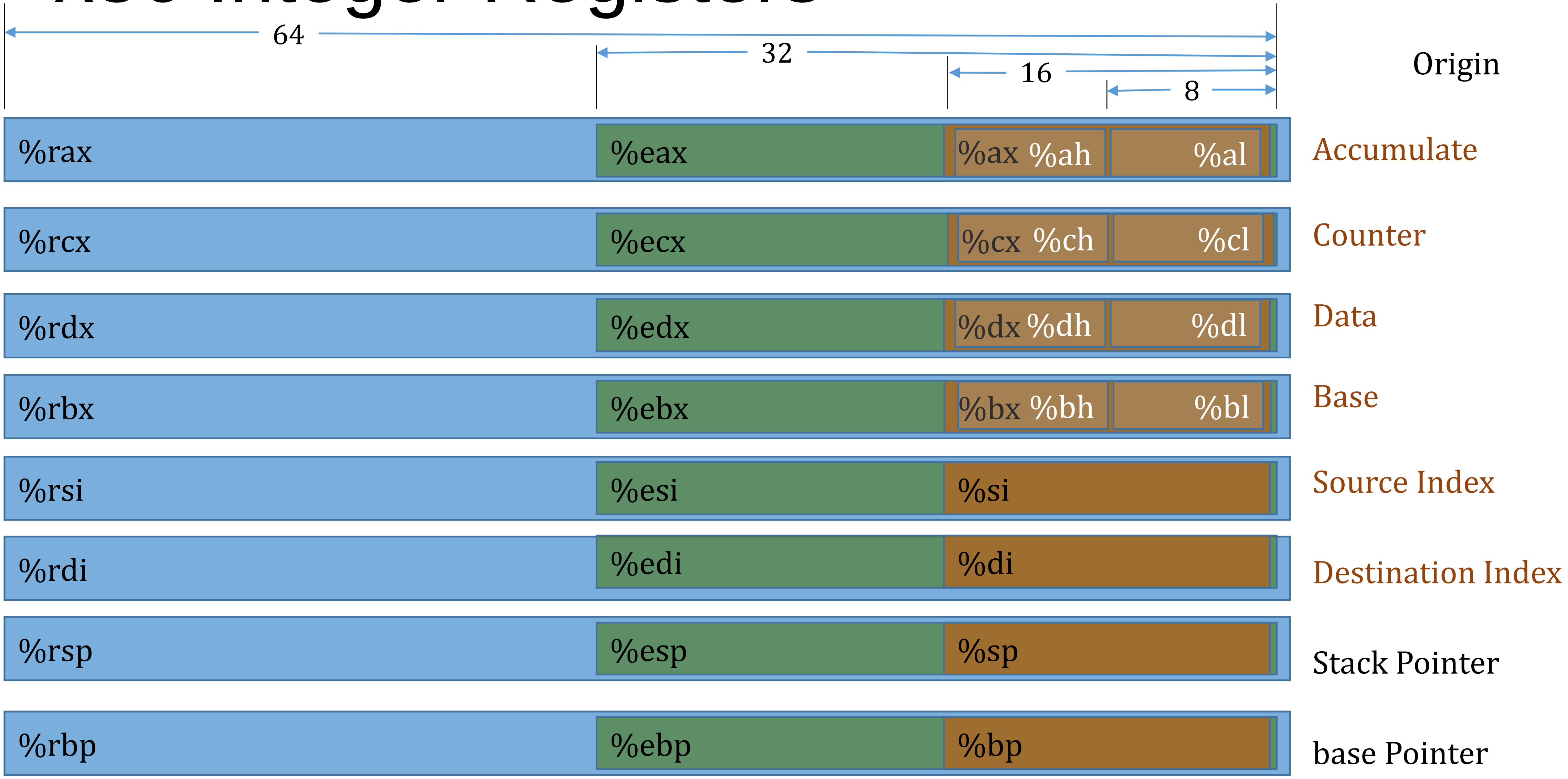
- Arguments may be:
  - a constant value,
  - a register,
  - a memory reference
- Only ONE argument may be a memory reference!
  - But if it’s the last argument, memory can be both read and written
- Optional argument prefixes
  - % - register e.g. “`mov 5,%eax`”
  - $ - constant value e.g. “`mov $5,%eax`”

Constant (literal) values

After optional $ prefix, similar to C Conventions….

- Numbers are decimal by default,
  - octal if preceded by 0,
  - hex if preceded by 0x
- Single characters are enclosed in single quotes,
  - including special characters such as ‘\n’, ‘\t’
- Strings are arrays of characters enclosed in double quotes
- Labels may be used in place of addresses

x86 Integer Registers



x86 Data “Types”

No type checking - Instruction and/or context implies data type

- Arithmetic instructions treat operands as numbers
  - Either signed or unsigned!
- Optional Opcode suffix used to identify width of arguments
  - b – 1 byte (8 bits)
  - w – word (2 bytes, 16 bits)
  - l – long word (4 bytes, 32 bits)
  - q – quad word (8 bytes, 64 bits)
- With no suffix, register implies width of arguments
  - %ah/%al – b – 8 bits
  - %ax – w – 16 bits
  - %eax – l – 32 bits
  - %rax – q – 64 bits
- Floating point – 4, 8, or 10 bytes

The MOV instruction

- Most often used instruction!
- More “copy” than “move”
- Copies 1,2,4, or 8 bytes from ARG1 to ARG2

```
mov $-12,%eax ; put -12 into 4 byte eax register
mov %eax,%ebx; copy value of %eax register into %ebx register
```

- Replaces target value

Memory Reference: indirection

- Simple indirection : ($0x000000000000000000C04)
  - Get the value at the literal address in parenthesis

```
mov ($0x0C04),%ebx
```

Reg	Value	
rax	???? ????	???? ????
rbx	???? ????	0018 0100

Address	Value
0xFFFF FFFF	
0xFFFF FFFE	0xDA
0xFFFF FFFD	0xED
0xFFFF FFFC	0xBE
0xFFFF FFFB	0xEF
…	
0x0000 0C07	0x00
0x0000 0C06	0x01
0x0000 0C05	0x18
0x0000 0C04	0x00
….	
0x0000 0003	0x00
0x0000 0002	0x00
0x0000 0001	0x00
0x0000 0000	0x03

Memory Reference: indirection

- Use a Register: (%rax)
  - The value of the register is the address in memory to use

```
mov $0x0C04,%rax
mov (%rax),%ebx
```

Reg	Value	
rax	0000 0000 0000 0C04	
rbx	???? ????	0018 0100

Address	Value
0xFFFF FFFF	
0xFFFF FFFE	0xDA
0xFFFF FFFD	0xED
0xFFFF FFFC	0xBE
0xFFFF FFFB	0xEF
…	
0x0000 0C07	0x00
0x0000 0C06	0x01
0x0000 0C05	0x18
0x0000 0C04	0x00
….	
0x0000 0003	0x00
0x0000 0002	0x00
0x0000 0001	0x00
0x0000 0000	0x03

Memory Reference: base/offset

- Indirection w/ offset: 8(%rax)
  - Get the value at the address in the %rax register + offset

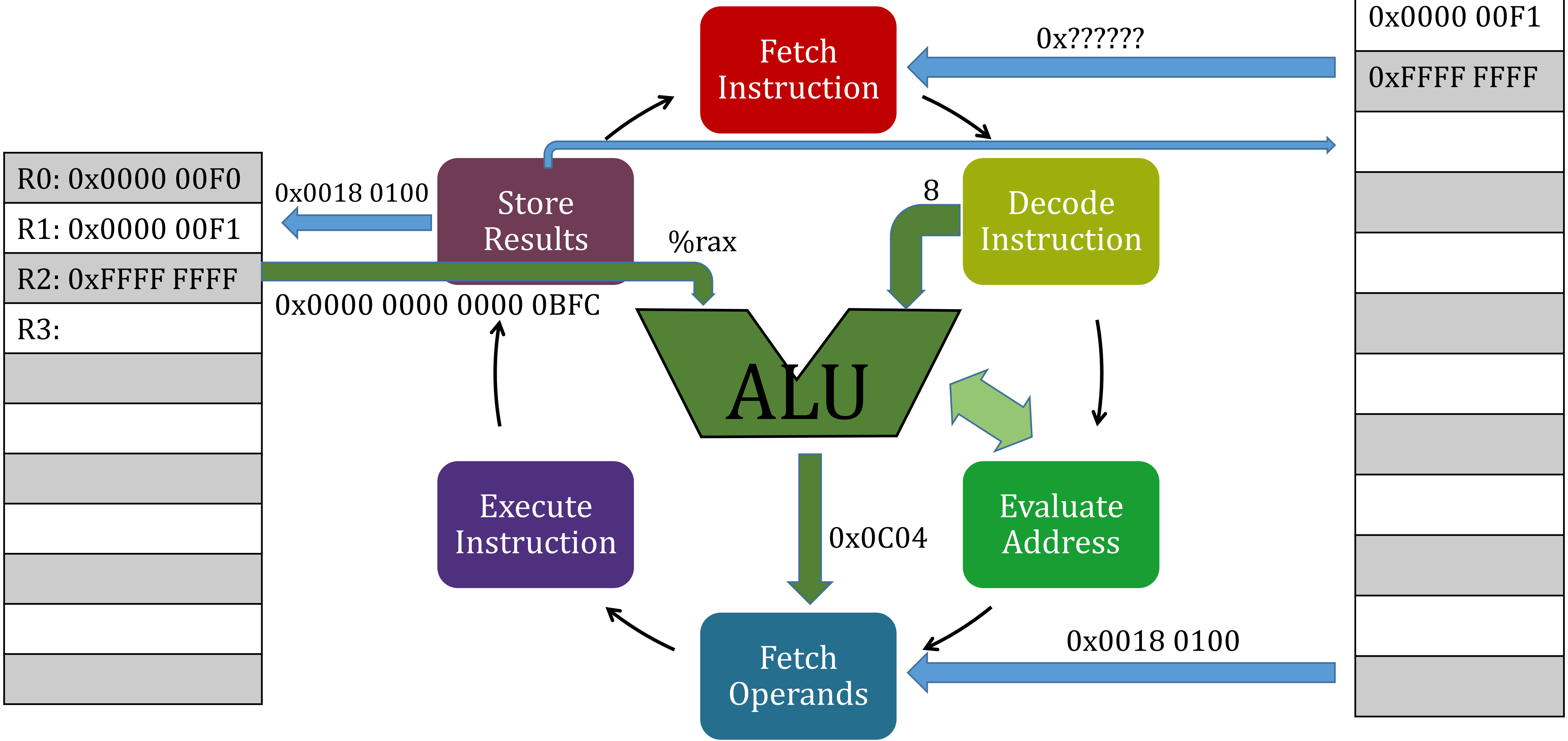
```
mov $0x0BFC,%rax
mov 8(%rax),%ebx
```

  - Offset may be negative, and may be expressed in hex

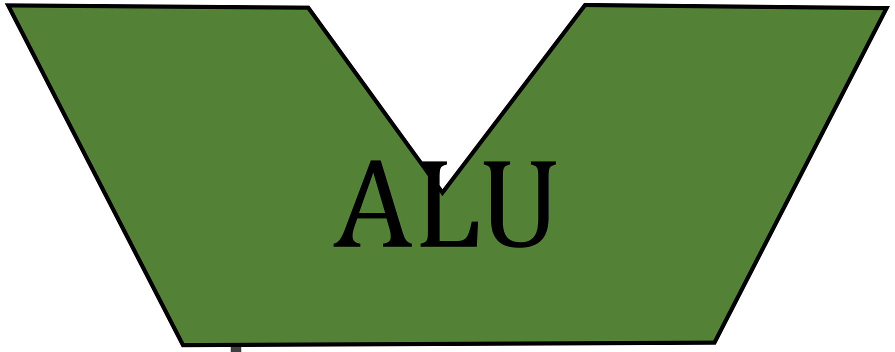
Reg	Value	
rax	0000 0000 0000 0BFC	
rbx	???? ????	0018 0100

Address	Value
0xFFFF FFFF	
0xFFFF FFFE	0xDA
0xFFFF FFFD	0xED
0xFFFF FFFC	0xBE
0xFFFF FFFB	0xEF
…	
0x0000 0C07	0x00
0x0000 0C06	0x01
0x0000 0C05	0x18
0x0000 0C04	0x00
….	
0x0000 0003	0x00
0x0000 0002	0x00
0x0000 0001	0x00
0x0000 0000	0x03

X86 Instruction Cycle: mov 8(%rax),%ebx



Condition Code Registers



- CF Carry Flag =1 if most significant bit overflows (unsigned)
- ZF Zero Flag = 1 if result bits are all zero
- SF Sign Flag = 1 if leftmost result bit is 1 (signed negative)
- OF Overflow Flag = 1 if result sign bit is incorrect
  (op1+, op2+ res- or op1-, op2-, res+)

Condition Codes (Implicit Setting)

- Implicitly set by arithmetic operations. e.g. sub b,a ; $a' = a - b$

Flag	Set to 1 if…	Interpretation
CF	Carry out from high order bit	Unsigned arithmetic overflow
ZF	a’ is all zeroes	a==b
SF	The sign bit is on in a’	a<b
OF	The sign bit is incorrect a>0, -b>0, a’<0 or a<0, -b<0, a’>0	Signed arithmetic overflow

- Not set by `lea` instruction
- Full documentation (nice summary) or Wikibooks X86 Control Flow

Invocation Record

In a C function

- %rsp -> start of the invocation record
- %rbp -> end of the invocation record
- Local vars are at the end of the record
- In x86, reference locals as -4(%rbp) or -0xc(%rpb)

Reg	Value
rsp	FFFF FFFF AAC4 0C00
rbp	FFFF FFFF AAC4 0C14

	Address	Value
	0xFFFF FFFC	0xDEADBEEF
	0xFFFF FFF8	0xDEADBEEF
	…	
	0xAAC4 0C18	0xDEADBEEF
%rbp->	0xAAC4 0C14	0x0000000D
local->	0xAAC4 0C10	0x0000000B
	0xAAC4 0C0C	0x0000000A
	0xAAC4 0C08	0x00000002
	0xAAC4 0C04	0x00000001
%rsp->	0xAAC4 0C00	0x00000000
	….	
	0x0000 0010	0xFFFFFE80
	0x0000 000c	0x00001A04
	0x0000 0004	0x0000001C
	0x0000 0000	0x03000000

Arithmetic Instructions

- Standard integer arithmetic: add sub

  add $10,(%eax); (*eax)=(*eax)+10

  sub $4,%esp ; esp=esp-4 (move stack pointer down)
- "Special" integer arithmetic: imul idiv
  - imul cannot write to memory
  - idiv divides register pair (EDX:EAX) and puts quotient/remainder back
- Single argument: inc dec

  inc %eax; eax=eax+1 – same as add eax,1

  dec (%esp) ; decrement the value at the top of the stack by 1
- Floating Point Instructions

Unsigned vs. Two's Complement Addition

Addition is Addition

1	1	1	1			1			UNS	SGN
	0	1	1	1	0	0	1	1	115	115
+	1	1	1	1	0	0	1	0	+242	+ -14
	0	1	1	0	0	1	0	1	101 OVFL	101

Overflow is Different!

Overflow with Addition

Unsigned

- Carry out of the high order bit
- CF condition code

Two's Complement

- Sign Bit Incorrect…
  - POS + POS = NEG or
  - NEG + NEG = POS
  - Note… Opposite signs never overflow!
    POS + NEG = No Overflow
- OF Condition code

C to X86 : Integer Arithmetic

C Code	X86 Implementation
int a=6;	movl $0x6,-0x4(%rbp)
int b=21;	movl $0x15,-0x8(%rbp)

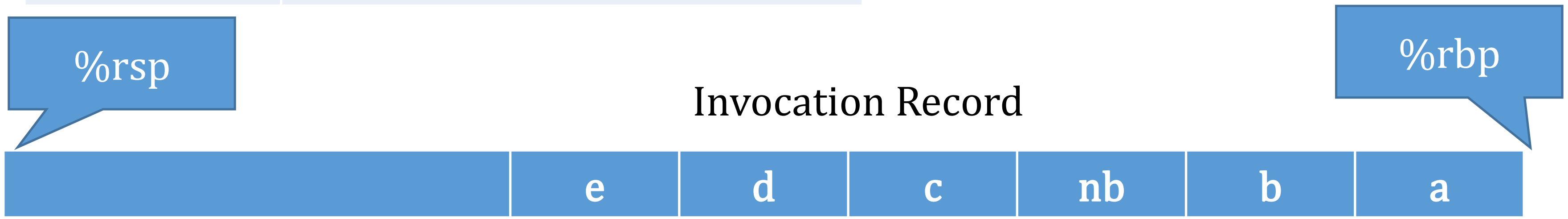
int nb=-b;	mov -0x8(%rbp),%eax
	neg %eax
	mov %eax,-0xc(%rbp)

int c=a+b;	mov -0x4(%rbp),%edx
	mov -0x8(%rbp),%eax
	add %edx,%eax
	mov %eax,-0x10(%rbp)

C Code	X86 Implmentation

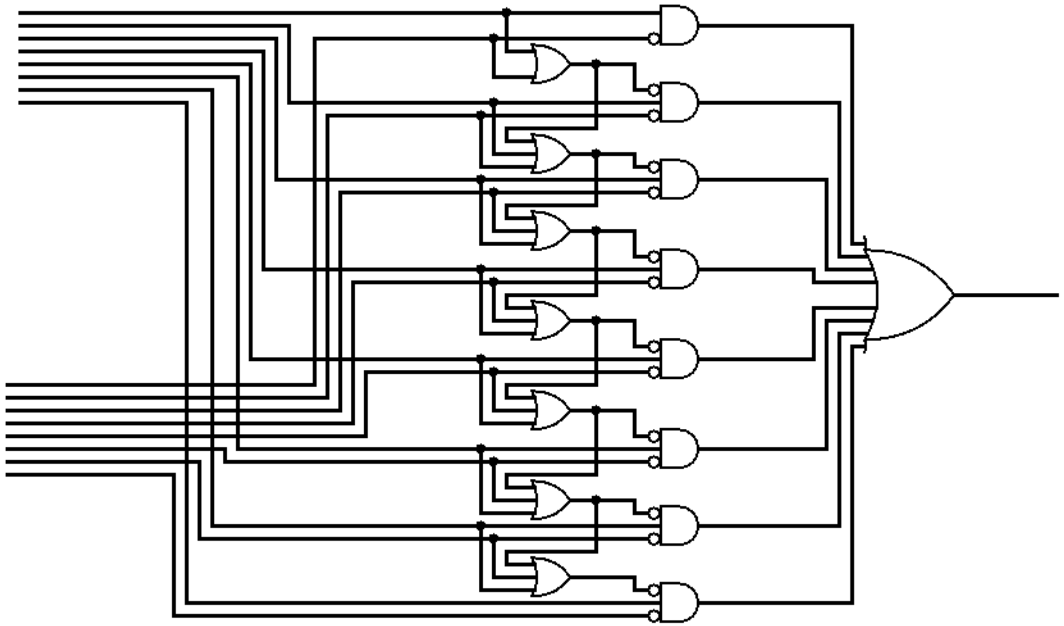
int d=a*b;	mov -0x4(%rbp),%eax
	imul -0x8(%rbp),%eax
	mov %eax,-0x14(%rbp)

int e=a-b;	mov -0x4(%rbp),%eax
	sub -0x8(%rbp),%eax
	mov %eax,-0x18(%rbp)



a @ -0x4(%rbp)
b @ -0x8(%rbp)
nb @ -0xc(%rbp)
c @ -0x10(%rbp)
d @ -0x14(%rbp)
e @ -0x18(%rbp)

Comparison: A vs B



- Instead of a hardware compare…
  - Requires ripple from MSB to LSB
  - Takes lots of time and gates
- (Signed) Arithmetic Compare: A-B
  - A-B>0 means A>B (SF=0, ZF=0, OF=0) OR (SF=1, ZF=0, OF=1)
  - A-B=0 means A==B (ZF=1)
  - A-B<0 means A<B (SF=1, ZF=0, OF=0) OR (SF=0, ZF=0, OF=1)

C to X86 : Comparison

C Code	X86 Implementation
int a=6; int b=-3;	movl $0x6,-0x4(%rbp) movl $0xfffffffd,-0x8(%rbp)

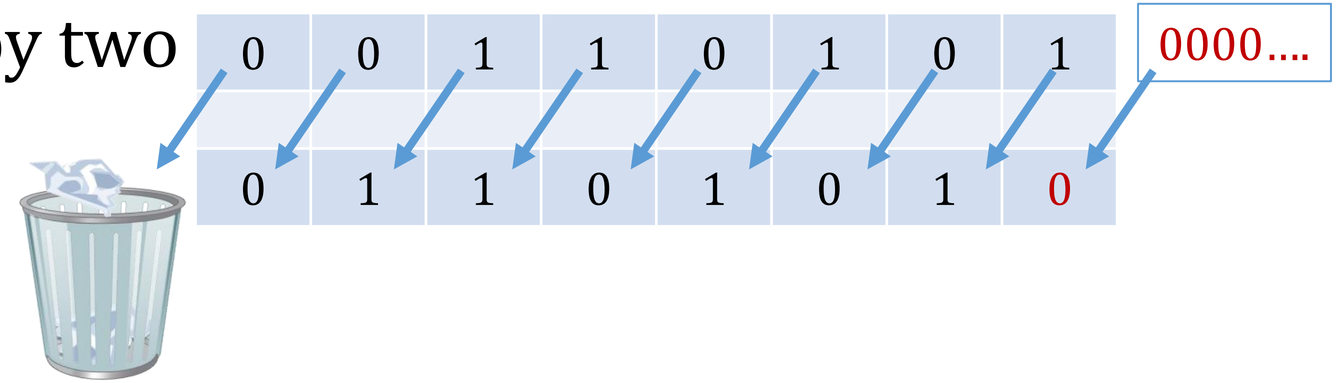
int c=(a==b);	mov -0x4(%rbp),%eax
	cmp -0x8(%rbp),%eax
	sete %al
	movzbl %al,%eax
	mov %eax,-0xc(%rbp)

int d=(a>b);	mov -0x4(%rbp),%eax
	cmp -0x8(%rbp),%eax
	setg %al
	movzbl %al,%eax
	mov %eax,-0x10(%rbp)

Bit Shifting

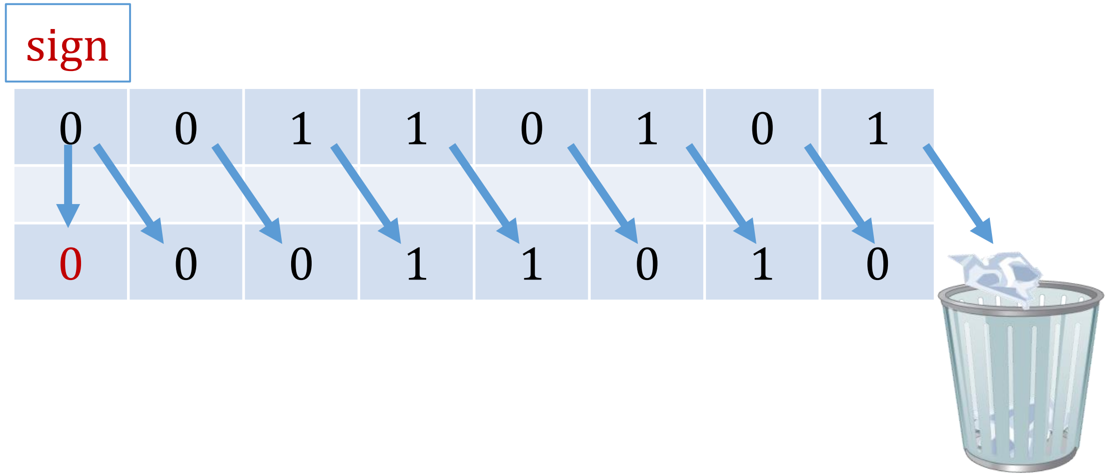
- Shift Left – Same as multiply by two

```
signed char x=53;
signed char y=x<<1;
```



- Shift Right – Same as divide by two (almost)

```
signed char x=53;
signed char y=x>>1;
```



See xmp_shift/shift.c

Bit Shifting… Signed vs. Unsigned

- Shift left… no difference – pad on right with 0
- Shift right…
  - Signed (arithmetic)… pad on left with sign bit
  - Unsigned (logical) … pad on left with “sign” bit… always 0
- In lower level languages…
  - “shift right logical” same as unsigned shift – pad on left with 0
  - “shift right arithmetic” same as signed shift – pad on left with sign bit

C to X86 : Shifting

C Code	X86 Implementation
int a=21;	movl $0x15,-0x4(%rbp)

int b=a<<2;	mov -0x4(%rbp),%eax
	shl $0x2,%eax
	mov %eax,-0x8(%rbp)

unsigned int c=-30000;
unsigned int d=c>>10;
	movl $0xffff8ad0,-0xc(%rbp)
	mov -0xc(%rbp),%eax
	shr $0xa,%eax
	mov %eax,-0x10(%rbp)

int e=a>>2;	mov -0x4(%rbp),%eax
	sar $0x2,%eax
	mov %eax,-0x14(%rbp)

C to X86 : Bitwise Operations

C Code	X86 Implementation
int a=12; int b=-42;	movl $0xc,-0x4(%rbp) movl $0xffffffd6,-0x8(%rbp)

int c = a & b;	mov -0x4(%rbp),%eax
	and -0x8(%rbp),%eax
	mov %eax,-0xc(%rbp)

int d = a ^ b;	mov -0x4(%rbp),%eax
	xor -0x8(%rbp),%eax
	mov %eax,-0x10(%rbp)

Table Addressing Mode

C Table Example

```
int mat[3][2]={{0,1},{10,11},{20,21}};
int i=1;
…
++mat[i][1];
```

Label	Address	Value
	0xFFFF FFFC	0xDEADBEEF
	0xFFFF FFF8	0xDEADBEEF
	…	
	0xAAC4 0C18	0xDEADBEEF
mat[2][1]	0xAAC4 0C14	0x00000015
mat[2][0]	0xAAC4 0C10	0x00000014
mat[1][1]	0xAAC4 0C0C	0x0000000B
mat[1][0]	0xAAC4 0C08	0x0000000A
mat[0][1]	0xAAC4 0C04	0x00000001
mat[0][0]	0xAAC4 0C00	0x00000000
	….	
	0x0000 0010	0xFFFFFE80
	0x0000 000c	0x00001A04
	0x0000 0004	0x0000001C
	0x0000 0000	0x03000000

Table Addressing Mode

- *Offset*(*Base*, *Row*, *Width*)   e.g. $4(%rbx,%rax,$8)
  - *Offset*=4, *Base*=%rbx, *Row*=%rax, *Width*=8
- $Address = (Base) + (Row \times Width) + Offset$
  - (%rbx)+(%rax*8)+4
  - 0xAAC40C00 + 1*8 + 4
  - 0xAAC40C0C

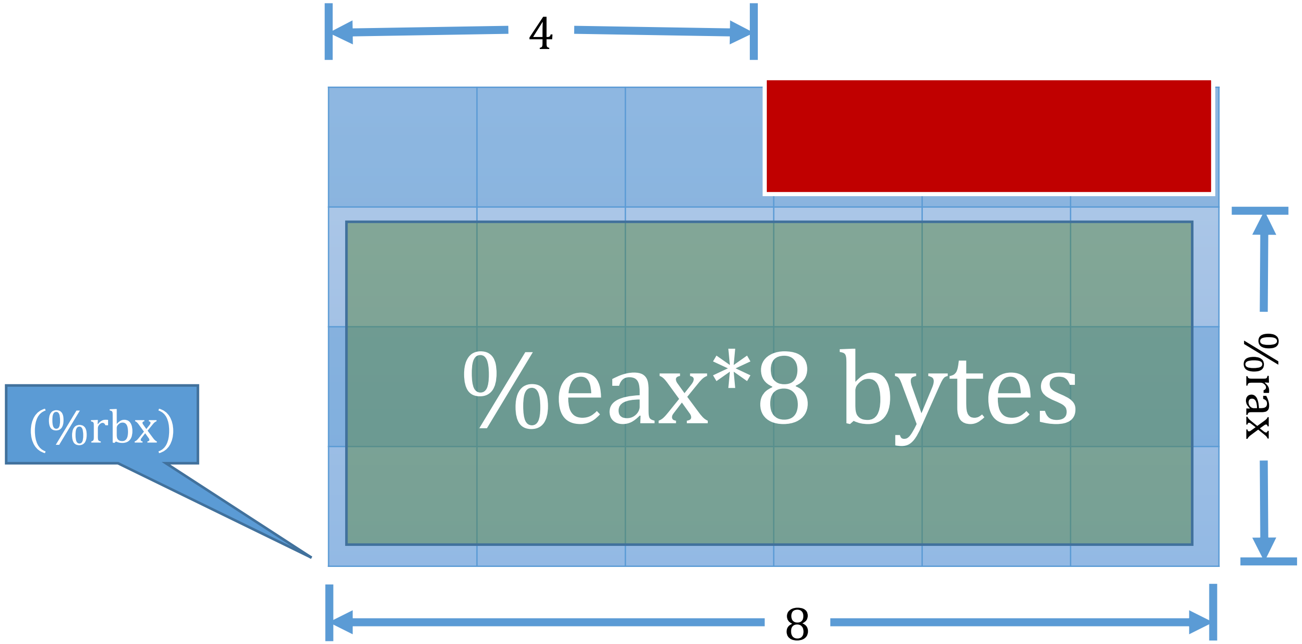


Table Addressing Mode Restrictions

- Offset must be a literal (or label)
- Base must be a 64 bit register
- Row must be a 64 bit register
- Width must be a literal: 1, 2, 4, or 8
- If Offset, Base, or Row are blank, assume default of 0.

- Because of width restriction, not really used for C tables as much as for C vectors (row major order) or structures

C Table Example

*Width=2*4*

```
int mat[3][2]={{0,1},{10,11},{20,21}};
int i=1;
...
++mat[i][1];
```

Row

*Offset= 1*4*

```
mov $1,%rax
movq $0x0AAC40C00,%rbx
addl $1,$4(%rbx,%rax,$8)
```

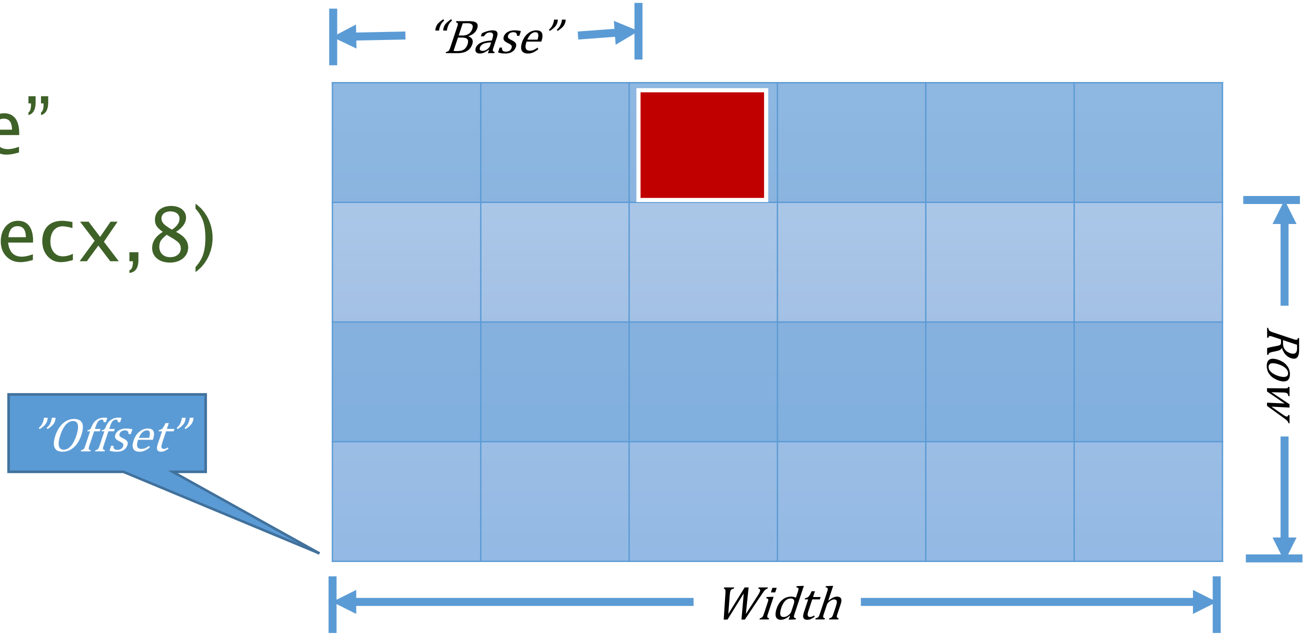
Base

Label	Address	Value
	0xFFFF FFFC	0xDEADBEEF
	0xFFFF FFF8	0xDEADBEEF
	…	
	0xAAC4 0C18	0xDEADBEEF
mat[2][1]	0xAAC4 0C14	0x00000015
mat[2][0]	0xAAC4 0C10	0x00000014
mat[1][1]	0xAAC4 0C0C	0x0000000B
mat[1][0]	0xAAC4 0C08	0x0000000A
mat[0][1]	0xAAC4 0C04	0x00000001
mat[0][0]	0xAAC4 0C00	0x00000000
	….	
	0x0000 0010	0xFFFFFE80
	0x0000 000c	0x00001A04
	0x0000 0004	0x0000001C
	0x0000 0000	0x03000000

Table Addressing Mode : Alternate view

- Also: $Offset(Base, Row, Width)$
  - e.g. mat(%rbx,%rcx,8)
- $Address = Base + Offset + (Row \times Width)$

```
mov $1,%ecx ; row
mov $4,%ebx ; "base"
addl $1,mat(%ebx,%ecx,8)
```



Dealing with Pointers

- Load effective address: lea
  - Used for implicit arrays/structures, etc.
  - Calculates address from first argument, and writes that address to second
  - Sometimes used as a cheap register to register “add” using addr/offset or table address mode

```
lea $-0x1c(%rbp),%rax ; %rax = &counter
lea $3(,$rax,2),$rax ; $rax = ($rax*2) + 3
```