CS 240
Foundations of Computer Systems

WELLESLEY

x86 Basics

Translation tools: C -> assembly <-> machine code

x86 registers, data movement instructions, memory addressing, arithmetic instructions

CSAPP book is **highly useful** and well-aligned with class for the remainder of the course.

https://cs.wellesley.edu/~cs240/

Big picture: Turning C into *Actual* Machine Code

Real-world security & performance implications

60-70%

of the world's servers/desktops!

```
void sumstore(long x, long y,
              long *dest) {
    long t = x + y;
    *dest = t;
}
```

sum.c

compiler (CS 301)

```
gcc -Og -S sum.c
```

Generated x86 Assembly Code

Human-readable language close to machine code.

```
sum:
    addq %rdi,%rsi
    movq %rsi,(%rdx)
    retq
```

sum.s

assembler

Object Code

```
0101010110001001111001011
0010110100010100001100000
0011010001010000100010010
0111101100010111011000011
```

sum.o

linker

Executable: `sum`

Resolve references between object files, libraries, (re)locate data

Machine Instruction Example

```
*dest = t;
```

C Code
Store value t where indicated by dest

```
movq %rsi, (%rdx)
```

Assembly Code
Move 8-byte value to memory
t: Register %rsi
dest: Register %rdx
*dest: Memory **M**[%rdx]

```
0x400539:  48 89 32
```

Object Code
3-byte instruction encoding
Stored at address 0x400539

CISC vs. RISC

x86: real ISA, widespread

CISC: maximalism

Complex Instruction Set Computer
Many instructions, specialized.
Variable-size encoding, complex/slow decode.
Gradual accumulation over time.
Original goal:

- humans program in assembly
- or simple compilers generate assembly by template
- hardware supports many patterns as single instructions
- fewer instructions per SLOC

Usually fewer registers.
We will stick to a small subset.

HW: toy, but based on real MIPS ISA

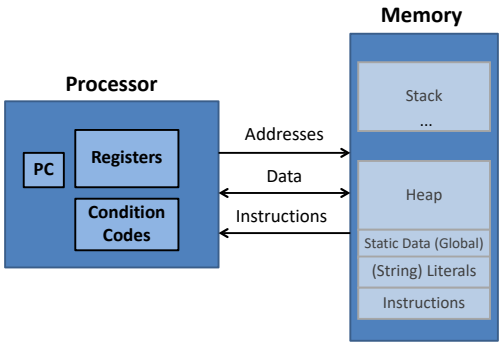
RISC: minimalism

Reduced Instruction Set Computer
Few instructions, general.
Regular encoding, simple/fast decode.
1980s+ reaction to bloated ISAs.
Original goal:

- humans use high-level languages
- smart compilers generate highly optimized assembly
- hardware supports fast basic instructions
- more instructions per SLOC

Usually many registers.

ISA View



x86-64 registers

16 named registers

Each 64 bits (8 bytes)

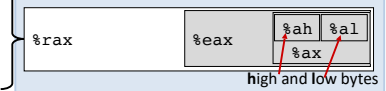
%rax	Return Value
%rbx	
%rcx	Argument 4
%rdx	Argument 3
%rsi	Argument 2
%rdi	Argument 1
%rsp	Special Purpose: Stack Pointer
%rbp	Special Purpose: Base Pointer
%r8	Argument 5
%r9	Argument 6
%r10	
%r11	
%r12	
%r13	
%r14	
%r15	

64-bits / 8 bytes

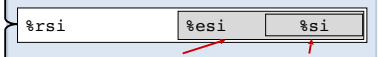
sub-registers

1985: 32-bit extended register %eax

1978: 16-bit register %ax



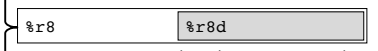
high and low bytes of %ax



Low 32 bits of %rsi

Low 16 bits of %rsi

historical artifacts



32-bit sub-register to match

Some have special uses for particular instructions

x86-64 registers: function arguments and return value

%rax	Return Value
%rbx	
%rcx	Argument 4
%rdx	Argument 3
%rsi	Argument 2
%rdi	Argument 1
%rsp	Special Purpose: Stack Pointer
%rbp	Special Purpose: Base Pointer
%r8	Argument 5
%r9	Argument 6
%r10	
%r11	
%r12	
%r13	
%r14	
%r15	

64-bits / 8 bytes

Mnemonic:

	register
Diana's	%rdi
silk	%rsi
dress	%rdx
costs	%rcx
$89	%r8
	%9

Arguments 7 and above are passed via stack, not in registers.

x86: Three Basic Kinds of Instructions

1. Data movement between memory and register

Load data from memory into register

%reg ← Mem[*address*]

Store register data into memory

Mem[*address*] ← *%reg*

Memory is conceptually an array[] of bytes!

2. Arithmetic/logic on register or memory data

```
c = a + b;      z = x << y;    i = h & g;
```

Next lecture:

3. Comparisons and Control flow to choose next instruction

Unconditional jumps to/from procedures

Conditional branches

Data movement instructions

(Like LW and SW in the HW ISA)

`mov_` *Source, Dest* *"copy the contents of **source** operand into **dest** operand"*

data size `_` is one of {`b, w, l, q`}

- `movq`: move 8-byte "quad word"
- `movl`: move 4-byte "long word"
- `movw`: move 2-byte "word"
- `movb`: move 1-byte "byte"

Historical terms based on the 16-bit days, **not** the current machine word size (64 bits)

Source, Dest operand types:

- ***Immediate:*** Literal integer data
  - Examples: `$0x400` `$-533`
- ***Register:*** One of 16 registers
  - Examples: `%rax` `%rdx`
- ***Memory:*** consecutive bytes in memory, at address held by register
  - Direct addressing: `(%rax)`
  - With displacement/offset: `8(%rsp)`

`mov` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`a = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`d = a;`
		Mem	`movq %rax,(%rdx)`	`*q = a;`
	Mem	*Reg*	`movq (%rax),%rdx`	`d = *p;`

Cannot do memory-memory transfer with a single instruction.

How would you do it?

Memory Addressing Modes

Indirect (R) Mem[Reg[R]]

Register R specifies memory address: `movq (%rcx), %rax`

Displacement D(R) Mem[Reg[R]+D]

Register R specifies **base** memory address (e.g. base of an object)

Displacement D specifies literal **offset** (e.g. a field in the object)

`movq %rdx, 8(%rsp)`

General Form: D(Rb,Ri,S) Mem[Reg[*Rb*] + *S**Reg[*Ri*] + *D*]

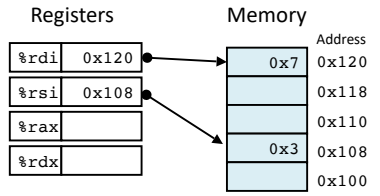
- D: Literal "displacement" value represented in 1, 2, or 4 bytes
- Rb: Base register: Any register
- Ri: Index register: Any except `%rsp`
- S: Scale: 1, 2, 4, or 8

Pointers and Memory Addressing

```
void swap(long* xp, long* yp){
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq (%rdi),%rax
   movq (%rsi),%rdx
   movq %rdx,(%rdi)
   movq %rax,(%rsi)
   retq
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

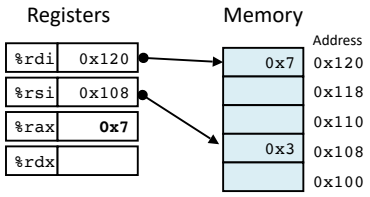


Pointers and Memory Addressing

```
void swap(long* xp, long* yp){
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq (%rdi),%rax
   movq (%rsi),%rdx
   movq %rdx,(%rdi)
   movq %rax,(%rsi)
   retq
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

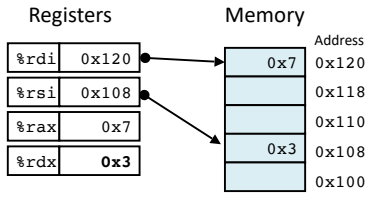


Pointers and Memory Addressing

```
void swap(long* xp, long* yp){
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq (%rdi),%rax
   movq (%rsi),%rdx
   movq %rdx,(%rdi)
   movq %rax,(%rsi)
   retq
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

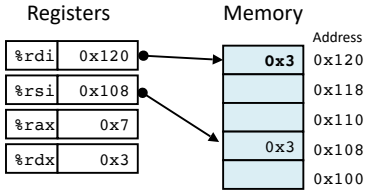


Pointers and Memory Addressing

```
void swap(long* xp, long* yp){
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq (%rdi),%rax
   movq (%rsi),%rdx
   movq %rdx,(%rdi)
   movq %rax,(%rsi)
   retq
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

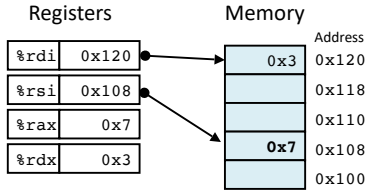


Pointers and Memory Addressing

```
void swap(long* xp, long* yp){
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq (%rdi),%rax
   movq (%rsi),%rdx
   movq %rdx,(%rdi)
   movq %rax,(%rsi)
   retq
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1



Address Computation Examples

ex

General Addressing Modes

D(Rb,Ri,S) Mem[Reg[*Rb*]+*S**Reg[*Ri*] + *D*]

Special Cases: **Implicitly:**

(Rb,Ri)	Mem[Reg[*Rb*]+Reg[*Ri*]]	(S=1,D=0)
D(Rb,Ri)	Mem[Reg[*Rb*]+Reg[*Ri*]+*D*]	(S=1)
(Rb,Ri,S)	Mem[Reg[*Rb*]+*S**Reg[*Ri*]]	(D=0)

Register contents

%rdx	0x6000
%rcx	0x100

Address Expression	Address Computation	Address
0x8(%rdx)	0x8 + 0x6000	
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

Address Computation Examples

ex

General Addressing Modes

D(Rb,Ri,S) Mem[Reg[*Rb*]+*S**Reg[*Ri*] + *D*]

Special Cases: **Implicitly:**

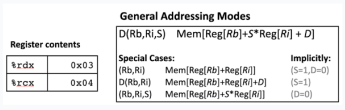
(Rb,Ri)	Mem[Reg[*Rb*]+Reg[*Ri*]]	(S=1,D=0)
D(Rb,Ri)	Mem[Reg[*Rb*]+Reg[*Ri*]+*D*]	(S=1)
(Rb,Ri,S)	Mem[Reg[*Rb*]+*S**Reg[*Ri*]]	(D=0)

Register contents

%rdx	0x6000
%rcx	0x100

Address Expression	Address Computation	Address
0x8(%rdx)	0x8 + 0x6000	6008
(%rdx,%rcx)	0x6000 + 0x100 * 1	6100
(%rdx,%rcx,4)	0x6000 + 0x100 * 4	6400
0x80(,%rdx,2)	0x80 + 0x0 + 0x6000 * 2	C080

What memory address will the following access? 0x1(%rdx,%rcx,2)



- 0x8 (8)
- 0xA (10)
- 0xB (11)
- 0xC (12)
- 0xD (13)
- 0xE (14)

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Compute address given by this addressing mode expression and **store it here**.

`leaq` *Src*, *Dest*

Load effective address

!!!

DOES NOT ACCESS MEMORY

Uses: "address of" "Lovely Efficient Arithmetic"

`p = &x[i];` x + k*I, where k = 1, 2, 4, or 8

leaq vs. movq

Registers	
%rax	
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

Assembly Code

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

Load effective address !!!

Compute address given by this addressing mode expression and **store it here**.

`leaq` *Src*, *Dest*

DOES NOT ACCESS MEMORY

Uses: "address of" "Lovely Efficient Arithmetic"

`p = &x[i];` x + k*I, where k = 1, 2, 4, or 8

leaq vs. movq

Registers	
%rax	**0x110**
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

Assembly Code

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

21

Load effective address !!!

Compute address given by this addressing mode expression and **store it here**.

`leaq` *Src*, *Dest*

DOES NOT ACCESS MEMORY

Uses: "address of" "Lovely Efficient Arithmetic"

`p = &x[i];` x + k*I, where k = 1, 2, 4, or 8

leaq vs. movq

Registers	
%rax	0x110
%rbx	**0x8**
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

Assembly Code

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

22

Load effective address !!!

Compute address given by this addressing mode expression and **store it here**.

`leaq` *Src*, *Dest*

DOES NOT ACCESS MEMORY

Uses: "address of" "Lovely Efficient Arithmetic"

`p = &x[i];` x + k*I, where k = 1, 2, 4, or 8

leaq vs. movq

Registers	
%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	**0x100**
%rsi	

Memory	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

Assembly Code

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

23

Load effective address !!!

Compute address given by this addressing mode expression and **store it here**.

`leaq` *Src*, *Dest*

DOES NOT ACCESS MEMORY

Uses: "address of" "Lovely Efficient Arithmetic"

`p = &x[i];` x + k*I, where k = 1, 2, 4, or 8

leaq vs. movq

Registers	
%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	0x100
%rsi	**0x1**

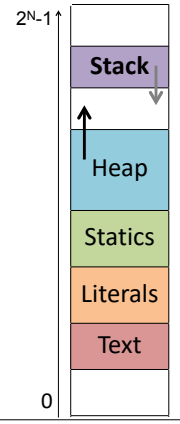
Memory	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

Assembly Code

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

24

Memory address-space layout



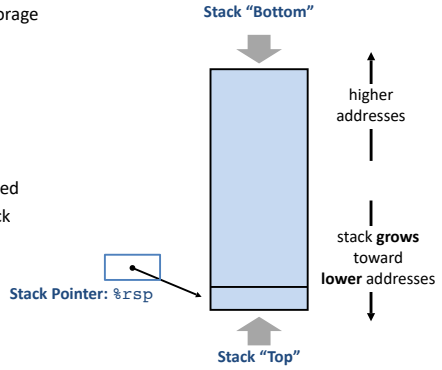
Perm	Contents	Managed by	Initialized
RW	**Procedure context**	**Compiler**	**Run time**
RW	Dynamic data structures	Programmer, malloc/free, new/ GC	Run time
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

Call **Stack**

Memory region for temporary storage managed with stack discipline.

`%rsp` holds lowest stack address (address of "top" element)

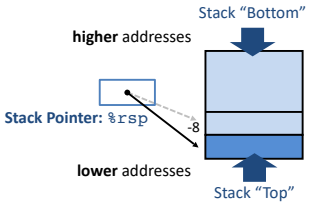
Local variables (that can't be stored in registers) are stored in the stack frame for their function.



Call Stack: **Push, Pop**

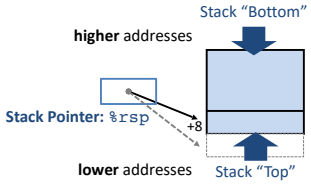
`pushq` ***Src***

1. Fetch value from *Src*
2. Decrement `%rsp` by 8 *(why 8?)*
3. Store value at new address given by `%rsp`



`popq` ***Dest***

1. Load value from address `%rsp`
2. Write value to *Dest*
3. Increment `%rsp` by 8



Assume the current stack pointer is at 0x00000000ffffff08 and %rdi is 0x0000000000000010. What is %rsp after the x86 instruction "pushq %rdi"?

0

- 0x0000000000000010
- 0x00000000ffffff18
- 0x00000000ffffff10
- 0x00000000ffffff00
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Procedure Preview (more soon)

`call, ret, push, pop`
Procedure arguments passed in 6 registers:

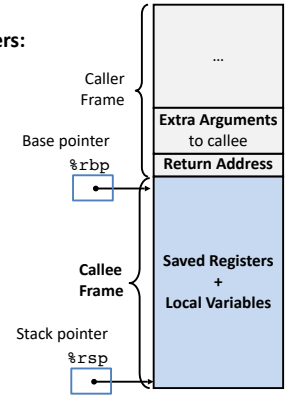
`%rax`	Return Value
`%rbx`	
`%rcx`	Argument 4
`%rdx`	Argument 3
`%rsi`	Argument 2
`%rdi`	Argument 1
`%rsp`	Stack pointer
`%rbp`	Base pointer

`%r8`	Argument 5
`%r9`	Argument 6
`%r10`	
`%r11`	
`%r12`	
`%r13`	
`%r14`	
`%r15`	

Return value in `%rax`.

Allocate/push new *stack frame* for each procedure call.
- Some local variables, saved register values, extra arguments

Deallocate/pop frame before return.



x86: Three Basic Kinds of Instructions

1. Data movement between memory and register
   - ***Load*** data from memory into register
     - *%reg* ← Mem[*address*]
   - ***Store*** register data into memory
     - Mem[*address*] ← *%reg*

Memory is an array[] of bytes!

2. Arithmetic/logic on register or memory data

```
c = a + b;      z = x << y;    i = h & g;
```

3. Comparisons and Control flow to choose next instruction
   - Unconditional jumps to/from procedures
   - Conditional branches

Arithmetic Operations

(Unlike the HW ISA, combines 1 operand and the destination)

Two-operand instructions:

Format	*Computation*	
`addq` *Src,Dest*	*Dest = Dest + Src*	
`subq` ***Src,Dest***	***Dest = Dest – Src***	***note the unexpected argument order***
`imulq` *Src,Dest*	*Dest = Dest * Src*	
`shlq` *Src,Dest*	*Dest = Dest << Src*	*a.k.a salq*
`sarq` *Src,Dest*	*Dest = Dest >> Src*	*Arithmetic*
`shrq` *Src,Dest*	*Dest = Dest >> Src*	*Logical*
`xorq` *Src,Dest*	*Dest = Dest ^ Src*	
`andq` *Src,Dest*	*Dest = Dest & Src*	
`orq` *Src,Dest*	*Dest = Dest | Src*	

One-operand (unary) instructions

`incq` *Dest*	*Dest = Dest + 1*	increment
`decq` *Dest*	*Dest = Dest – 1*	decrement
`negq` *Dest*	*Dest = –Dest*	*negate*
`notq` *Dest*	*Dest = ~Dest*	*bitwise complement*

See CSAPP 3.5.5 for: `mulq`, `cqto`, `idivq`, `divq`

`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
`%rdi`	Argument x
`%rsi`	Argument y
`%rdx`	Argument z
`%rax`	
`%rcx`	

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1
%rcx	

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```


`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	~~t1~~ t2
%rcx	

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```


`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ 3*y
%rax	~~t1~~ t2
%rcx	

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```


`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ ~~3*y~~ t4
%rax	~~t1~~ t2
%rcx	

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx              == %rdx << 4
  leaq    4(%rdi,%rdx), %rcx    == %rdx * 2^4
  imulq   %rcx, %rax            == %rdx * 16
  ret                           == (3*y) * 16
                                == y * 48
```

`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ ~~3*y~~ t4
%rax	~~t1~~ t2
%rcx	4+x+t4 = t5

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```


`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ ~~3*y~~ t4
%rax	~~t1~~ ~~t2~~ rval
%rcx	4+x+t4 = t5

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```


`leaq` for arithmetic

```
long arith(long x, long y,
           long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ ~~3*y~~ t4
%rax	~~t1~~ ~~t2~~ rval
%rcx	4+x+t4 = t5

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Same x86 code as compiling:

```
(x+y+z)*(x+4+48*y)
```


Compiler optimization example

```
long logical(long x, long y){
  long t1 = x^y;
  long t2 = t1 >> 17;
  long mask = (1<<13) - 7;
  long rval = t2 & mask;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```

Compiler optimization example

```
long logical(long x, long y){
  long t1 = x^y;
  long t2 = t1 >> 17;
  long mask = (1<<13) - 7;
  long rval = t2 & mask;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	x

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```


Compiler optimization example

```
long logical(long x, long y){
  long t1 = x^y;
  long t2 = t1 >> 17;
  long mask = (1<<13) - 7;
  long rval = t2 & mask;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	~~x~~ t1

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```


Compiler optimization example

```
long logical(long x, long y){
  long t1 = x^y;
  long t2 = t1 >> 17;
  long mask = (1<<13) - 7;
  long rval = t2 & mask;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	~~x~~ t1 t2

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```


Compiler optimization example

```
long logical(long x, long y){
  long t1 = x^y;
  long t2 = t1 >> 17;
  long mask = (1<<13) - 7;
  long rval = t2 & mask;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	~~x~~ t1 t2 rval

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```