Introduction to Processor Architecture (EC2.204)

LECTURE 2 – INSTRUCTION SET ARCHITECTURE

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Based on materials from text books and other public sources

ISA: Hardware – Software Interface

Programming Languages and Abstractions

A Programming Language provides

- Data Abstractions
  - int, float, bool etc. data types.
  - Mechanism for hierarchical composition of new Data Abstractions
    - Structures, arrays, Unions etc.
- Data Processing Abstractions
  - Arithmetic and Boolean Operations, String Operations etc.
- Control Abstractions – while, if, for constructs etc.

Key Idea: We should be able to realize these abstractions through the ISA of a given processor!

ISA Design Philosophy

- Two ISA design philosophies
  - RISC – Reduced Instruction Set Computer
  - CISC – Complex Instruction Set Computer

RISC versus CISC

RISC	CISC
Each instruction does one simple task.	Each instruction can do multiple tasks.
Amount of work done in each instruction is roughly the same.	Amount of work done in each instruction could have huge variance.
Fixed Length instruction format.	Variable length instruction format.
Load-Store Architecture. Instruction operands should always reside in registers.	Instruction Operands can reside in memory also.
Large bank of general purpose registers.	Many special purpose registers.
Simple Addressing Modes.	Can have complex addressing modes.
Few Data Types (typically integer and float)	Could provide support for more data types like Strings.
Berkeley RISC , Stanford MIPS, ARM, HP's PA-RISC	Intel x86 line of processors

RISC or CISC – Which way should we go?

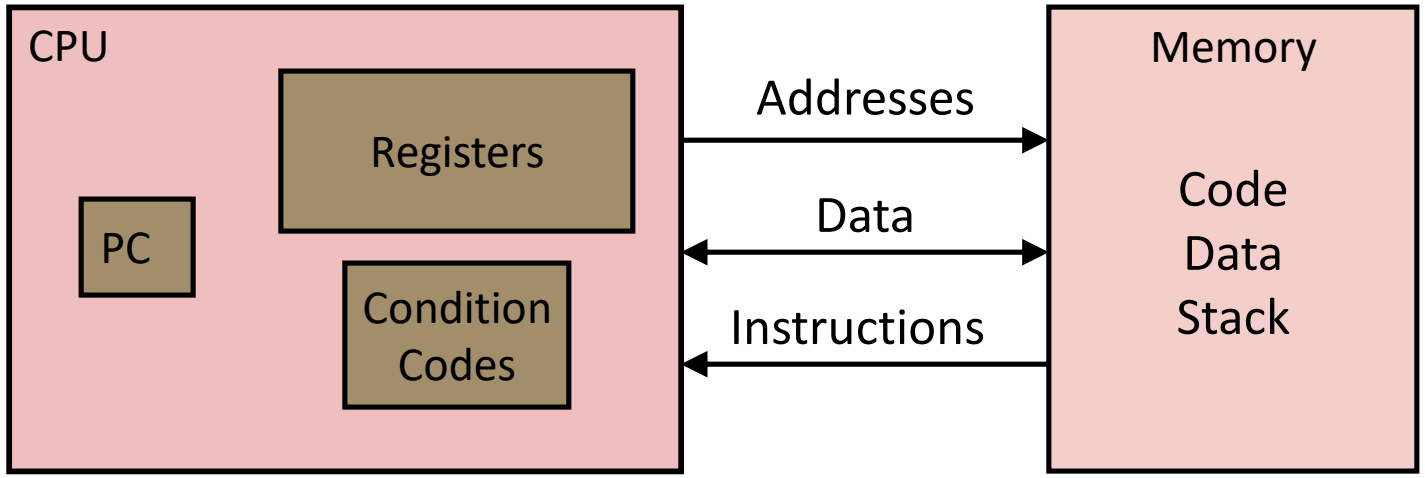
RISC	CISC
Simple, fast (pipelined) and power efficient hardware implementations.	Complex hardware, not so Power efficient, hard to come up with pipelined implementations.
Not so good for an Assemble Language Programmer when compared with CISC ISAs.	Good for an Assemble Language Programmer.
Good for compiler writer.	Compiler writer has to work hard to use the underlying CISC ISA features.
Less code density	Good code density

Assembly Language Example

```
long mult2(long, long);

void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
multstore:
  pushq   %rbx
  movq    %rdx, %rbx
  call    mult2
  movq    %rax, (%rbx)
  popq    %rbx
  ret
```

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

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

8-bit, 16-bit, 32-bit, 64-bit ISAs

A processor supporting a 32-bit ISA

- Registers are 32 bits in length
- 32-bit ALU
- Data bus width between processor and main memory 32 bits
- Address space size 2^32 bytes [4 GB]

A 64-bit processor may support a 32-bit ISA also for compatibility reasons [but don’t expect performance gains]

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

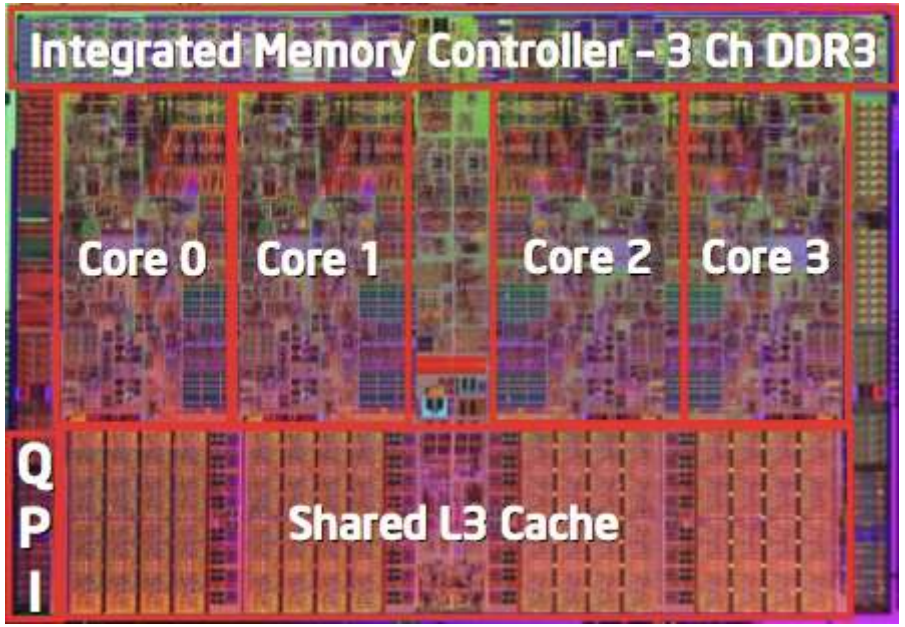
Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10
◦ First 16-bit Intel processor. Basis for IBM PC & DOS			
◦ 1MB address space			
386	1985	275K	16-33
◦ First 32 bit Intel processor , referred to as IA32			
◦ Added “flat addressing”, capable of running Unix			
Pentium 4E	2004	125M	2800-3800
◦ First 64-bit Intel x86 processor, referred to as x86-64			
Core 2	2006	291M	1060-3500
◦ First multi-core Intel processor			
Core i7	2008	731M	1700-3900
◦ Four cores			

Intel x86 Processors, cont.

Machine Evolution

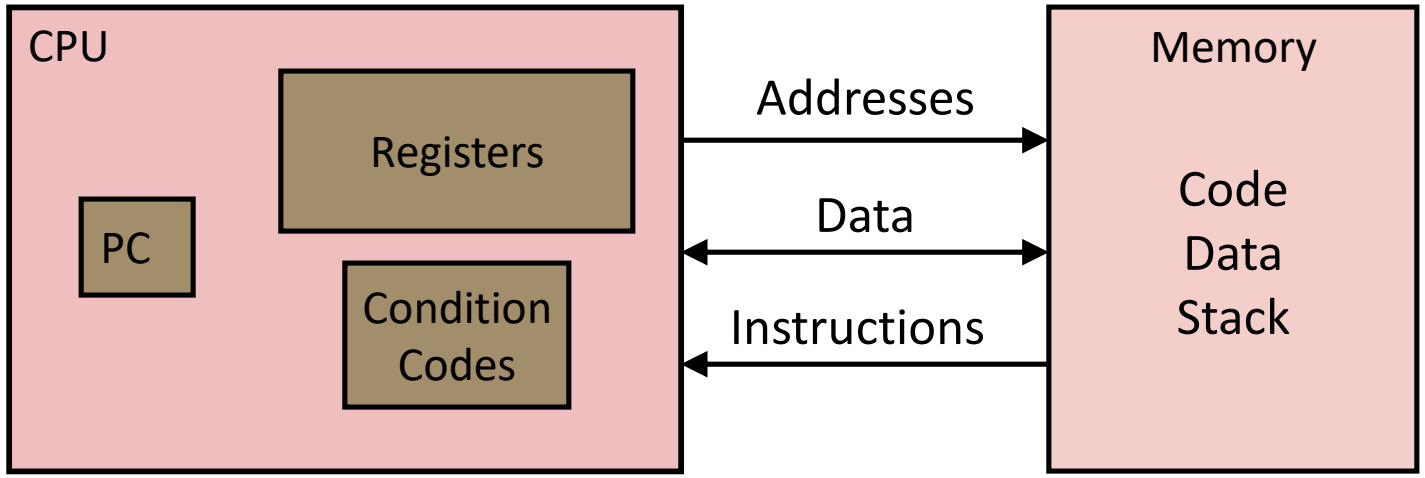
◦ 386	1985	0.3M
◦ Pentium	1993	3.1M
◦ Pentium/MMX	1997	4.5M
◦ PentiumPro	1995	6.5M
◦ Pentium III	1999	8.2M
◦ Pentium 4	2001	42M
◦ Core 2 Duo	2006	291M
◦ Core i7	2008	731M



Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

Assembly/Machine Code View



Programmer-Visible State

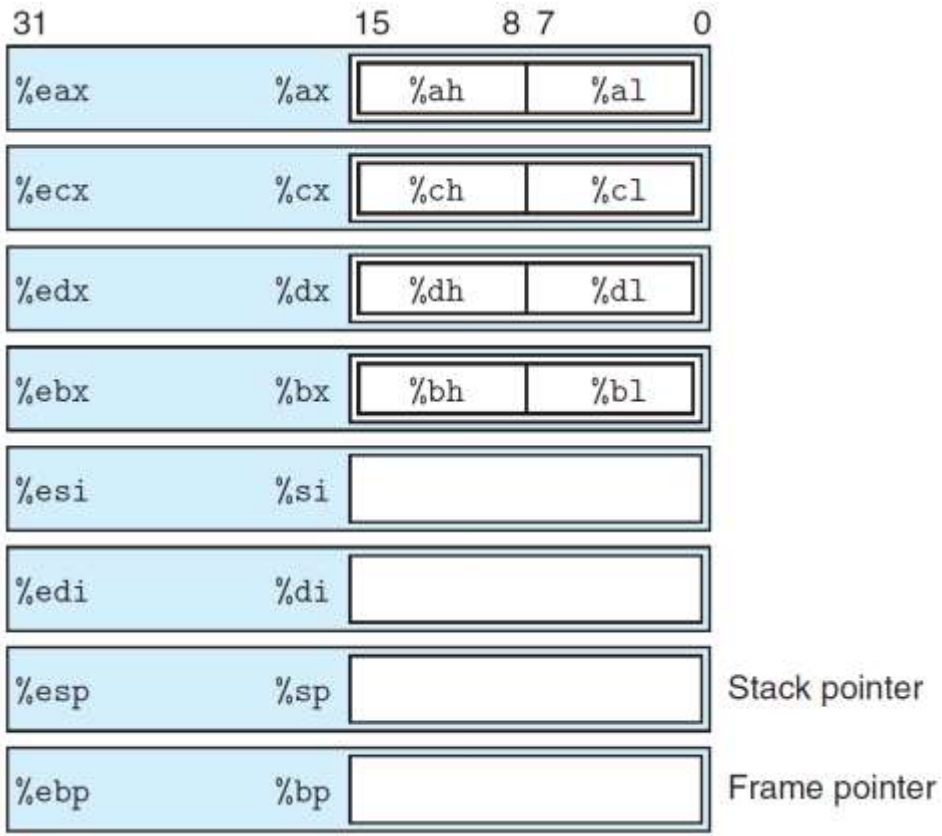
- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

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

Registers

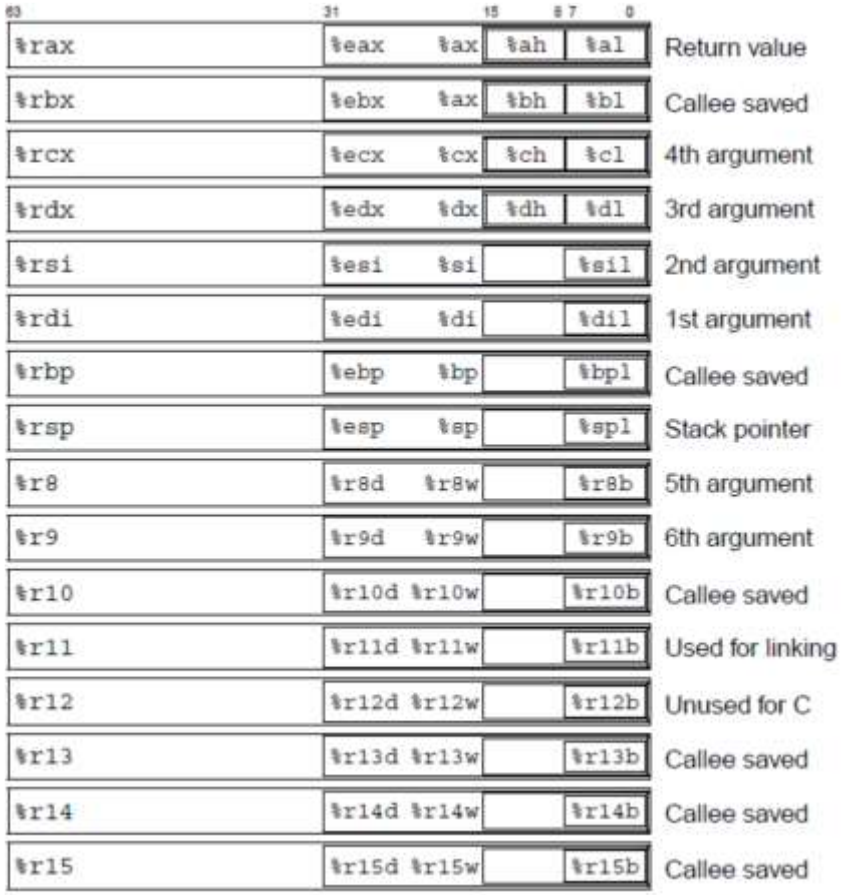
Accessing Information: IA32 Integer Registers

- Used to store integer/pointers
  - First six registers can be considered general purpose registers, except some instructions that uses specific registers as source and/or destination.
  - Last two registers, contain pointers to important places in the program stack
  - The low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions.
  - Similarly, the low-order 16 bits of each register can be read or written by word operation instructions.



Accessing Information: x86-64 Integer Registers

- 16 registers to store integer/pointers
  - First six registers can be considered general purpose registers, except some instructions that uses specific registers as source and/or destination.
  - Next two registers, contain pointers to important places in the program stack
  - The low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions.
  - Similarly, the low-order 16 bits of each register can be read or written by word operation instructions.



General Purpose Registers

- Accumulator register (ax). Used in arithmetic operations
- Counter register (cx). Used in shift/rotate instructions and loops.
- Data register (dx). Used in arithmetic operations and I/O operations.
- Base register (bx). Used as a pointer to data
- Stack Pointer register (sp). Pointer to the top of the stack.
- Stack Base Pointer register (bp). Used to point to the base of the stack.
- Source Index register (si). Used as a pointer to a source in stream operations.
- Destination Index register (di). Used as a pointer to a destination in stream operations.

Source: Wikipedia

Data Transfer Instructions

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

Normal (R) Mem[Reg[R]]

- Register R specifies memory address

```
movq (%rcx),%rax
```

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

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

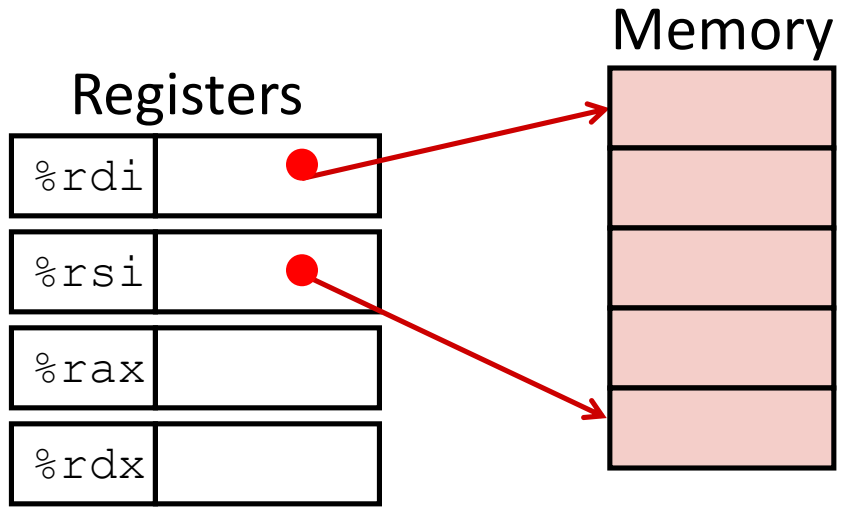
Example of Simple Addressing Modes

```
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)
    ret
```

Understanding Swap()

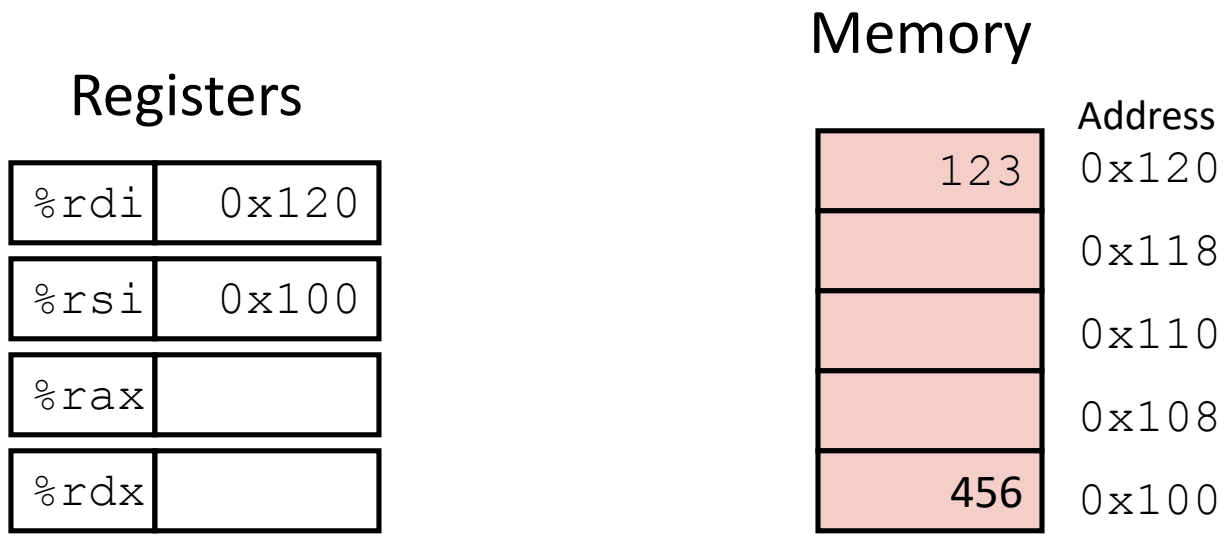
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

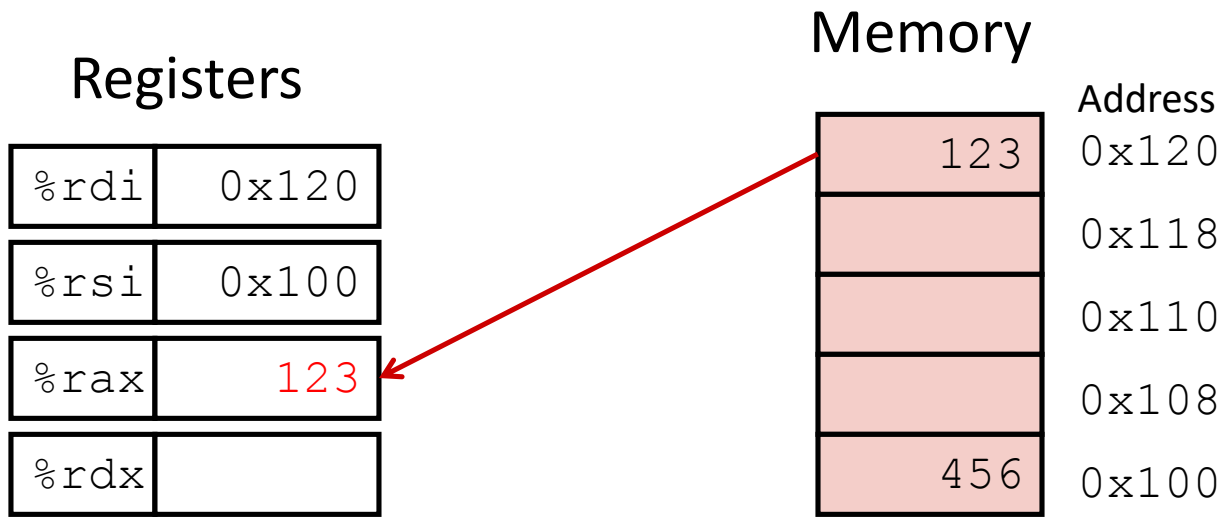
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



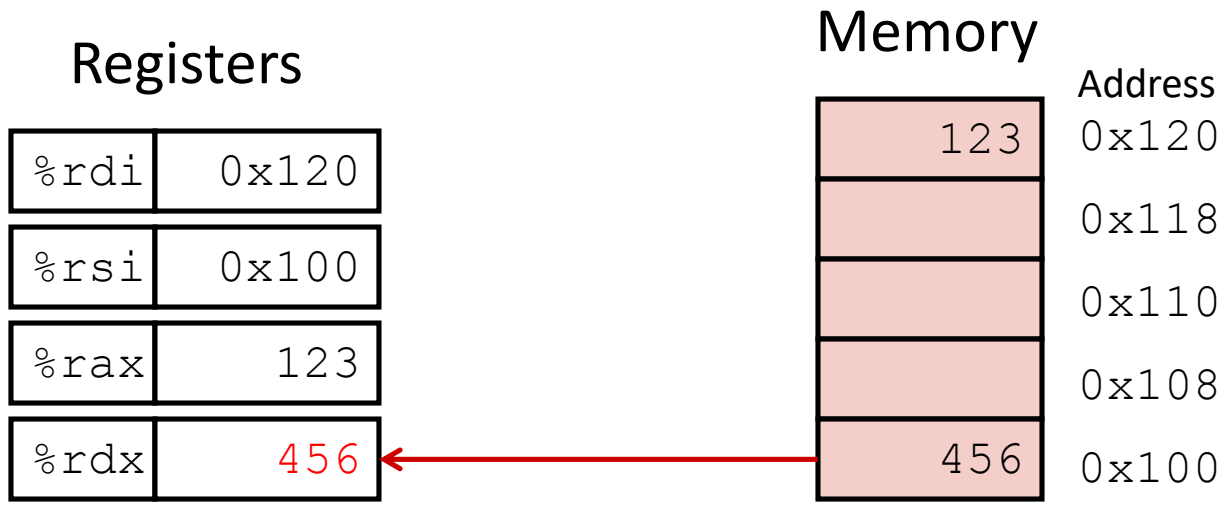

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



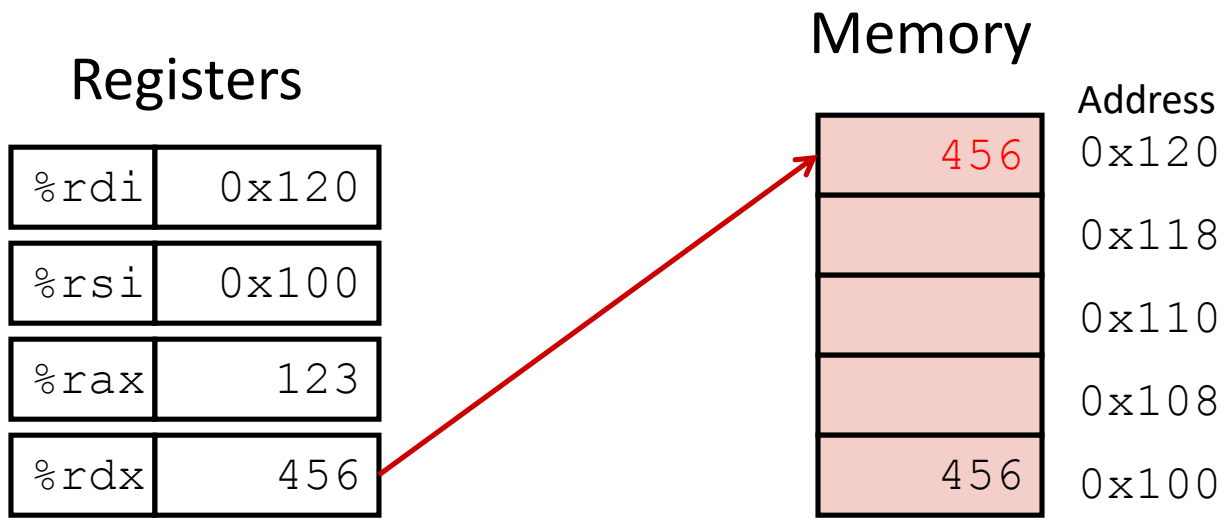

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



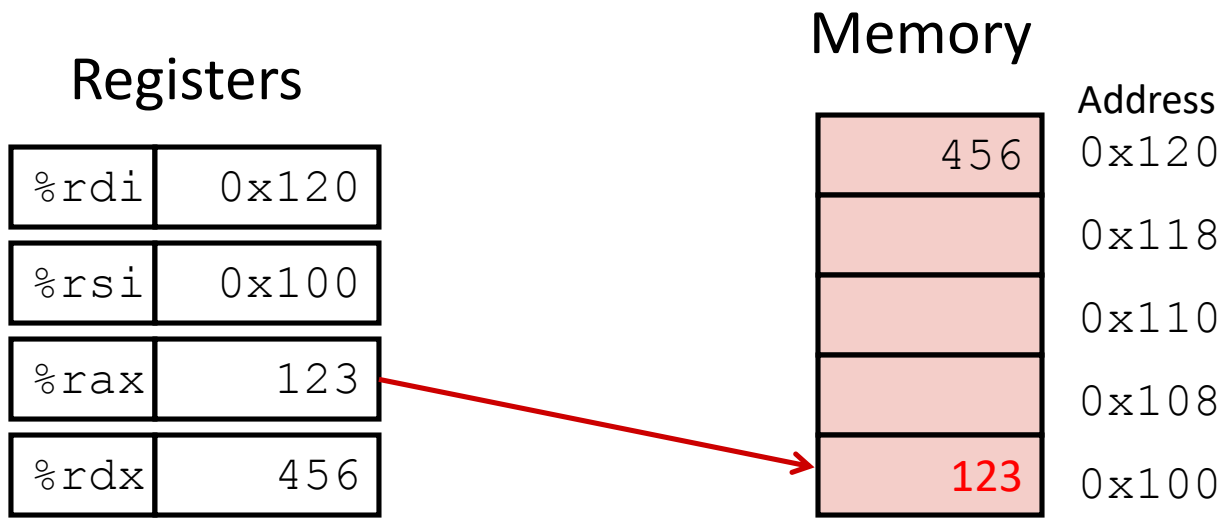

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Simple Memory Addressing Modes

Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

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

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Complete Memory Addressing Modes

Most General Form

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

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Special Cases

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

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Data Transfer Instructions

Example

(a) C code

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

(b) Assembly code

```
    xp in %rdi, y in %rsi
1   exchange:
2     movq    (%rdi), %rax    Get x at xp. Set as return
3     movq    %rsi, (%rdi)    Store y at xp.
4     ret                     Return.
```

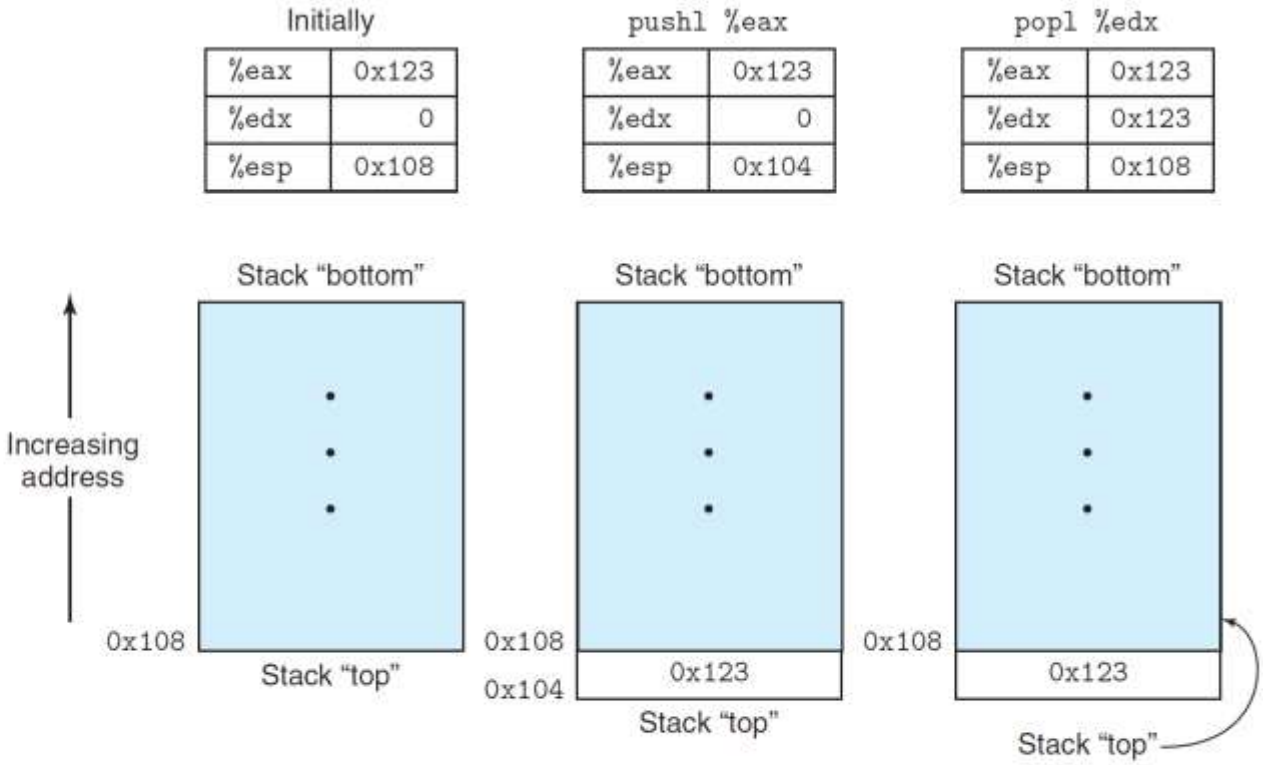
- Dereferencing a pointer → Copying the pointer into a register and using the register in a memory reference
- Local variables are often stored in registers rather than memory locations → Faster access

Push and Pop

Instruction		Effect	Description
`pushl`	S	R[%esp] $\leftarrow$ R[%esp] $-$ 4; M[R[%esp]] $\leftarrow S$	Push double word
`popl`	D	$D \leftarrow$ M[R[%esp]]; R[%esp] $\leftarrow$ R[%esp] $+$ 4	Pop double word

Push and Pop

IA 32 Example



subl $4,%esp
movl %eax,(%esp)

```
movl (%esp),%eax
addl $4,%esp
```

Arithmetic and Logic Operations

Address Computation Instruction

`leaq` *Src*, *Dst*

- *Src* is address mode expression
- Set *Dst* to address denoted by expression

Uses

- Computing addresses without a memory reference
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8
- Compilers can use it for compact arithmetic operations

Exam

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Unary Operations

Instruction		Effect	Description
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement

- The single operand can either be a register or a memory location
- Example:
  incl (%esp) → Increments the 4 byte element on top of the stack

Some Arithmetic Operations

Two Operand Instructions:

Format		Computation	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`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	

Watch out for argument order!

No distinction between signed and unsigned int (why?)

Shift Operations

Instruction		Effect	Description
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Operation	Values	
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[0101*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

Arithmetic Expression Example

```
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;
}
```

```
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
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

```
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;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

Condition Codes

- CPU maintains a set of single-bit condition code registers describing attributes of the most recent arithmetic or logical operation.
- These registers (listed below) can then be tested to perform conditional branches.

`CF`: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero Flag. The most recent operation yielded zero.

`SF`: Sign Flag. The most recent operation yielded a negative value.

`OF`: Overflow Flag. The most recent operation caused a two's-complement overflow—either negative or positive.

Condition Codes

t = a + b

CF:	`(unsigned) t < (unsigned) a`	Unsigned overflow
ZF:	`(t == 0)`	Zero
SF:	`(t < 0)`	Negative
OF:	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

Condition Codes

- All unary and binary arithmetic & logical operations (except leaq) set the single bit condition codes.
- For logical operations, the carry and overflow flags are set to zero.
- Fore shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero.
- INC and DEC instruction set the overflow and zero flags but leave the carry flag unchanged.

Condition Codes

- Two instruction classes (CMP and TEST) set the condition codes.
- CMP behave similar to SUB without altering the destination register.
- TEST behave similar to AND without altering the destination register.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
`cmpb`		Compare byte	
`cmpw`		Compare word	
`cmpl`		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		Test double word	

Condition Codes

- Three common ways to use the condition codes

1. Set a single byte to 0 or 1 depending on some combination of condition codes (SET instructions)
2. Conditionally jump to some other part of the program (Jump instructions)
3. Conditionally move data (CMOVE instructions)

Accessing the Condition Codes

- Instruction suffixes refer to combination of condition codes and not the operand types.
- Operand D refers to a single byte register or single byte memory location.

Instruction		Synonym	Effect	Set condition
sete	*D*	setz	*D* ← ZF	Equal / zero
setne	*D*	setnz	*D* ← ~ZF	Not equal / not zero
sets	*D*		*D* ← SF	Negative
setns	*D*		*D* ← ~SF	Nonnegative
setg	*D*	setnle	*D* ← ~(SF ^ OF) & ~ZF	Greater (signed >)
setge	*D*	setnl	*D* ← ~(SF ^ OF)	Greater or equal (signed >=)
setl	*D*	setnge	*D* ← SF ^ OF	Less (signed <)
setle	*D*	setng	*D* ← (SF ^ OF) | ZF	Less or equal (signed <=)
seta	*D*	setnbe	*D* ← ~CF & ~ZF	Above (unsigned >)
setae	*D*	setnb	*D* ← ~CF	Above or equal (unsigned >=)
setb	*D*	setnae	*D* ← CF	Below (unsigned <)
setbe	*D*	setna	*D* ← CF | ZF	Below or equal (unsigned <=)

Condition Codes

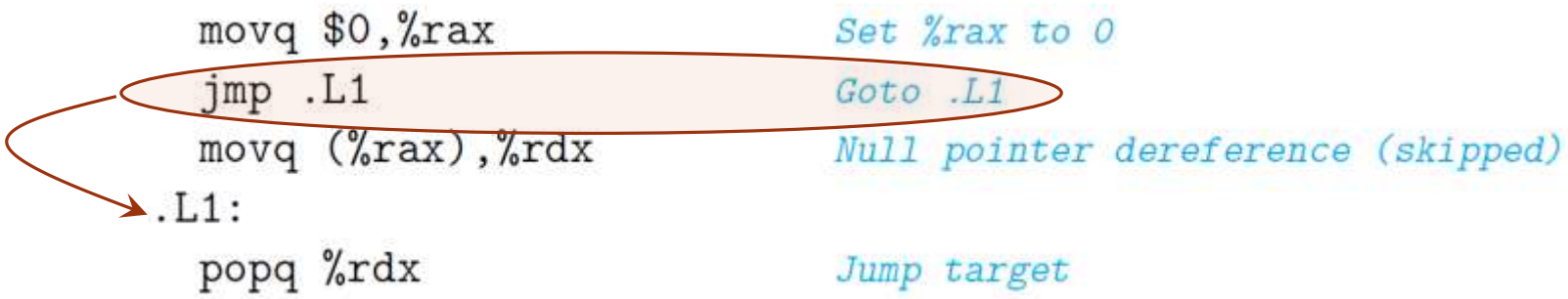
a < *b*

```
    a is in %edx, b is in %eax
1     cmpl    %eax, %edx     Compare a:b
2     setl    %al            Set low order byte of %eax to 0 or 1
3     movzbl  %al, %eax      Set remaining bytes of %eax to 0
```

setl	*D*	setnge	$D \leftarrow$ SF ^ OF	Less (signed <)

JUMP Instruction

- A JUMP instruction can cause the execution to switch to a completely new position in the program
- Generally a label used in assembly code to indicate the jump destination
- Addresses of jump targets determined during object code generation



JUMP Instructions and Condition Codes

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

JUMP Instruction Encoding

Two common encodings used are

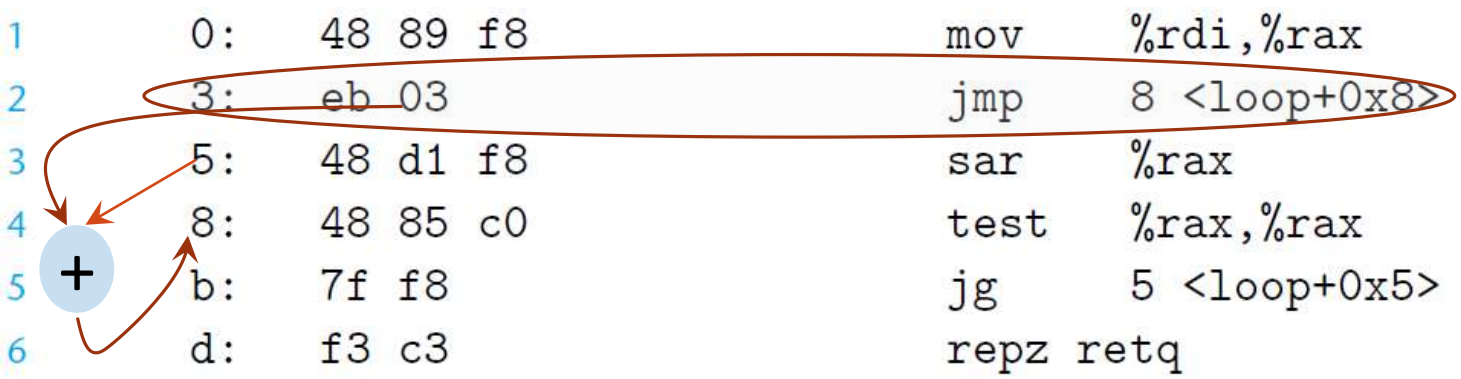
- PC relative: Encodes the difference between the address of target instruction and address of instruction immediately following the jump
- Absolute address

- Why relative to address of instruction immediately following jump?

Assembly code

```
1     movq    %rdi, %rax
2     jmp     .L2
3   .L3:
4     sarq    %rax
5   .L2:
6     testq   %rax, %rax
7     jg      .L3
8     rep; ret
```

Disassembled code



JUMP Instruction Encoding

Disassembled version after linking

```
1    4004d0:   48 89 f8          mov    %rdi,%rax
2    4004d3:   eb 03             jmp    4004d8 <loop+0x8>
3    4004d5:   48 d1 f8          sar    %rax
4    4004d8:   48 85 c0          test   %rax,%rax
5    4004db:   7f f8             jg     4004d5 <loop+0x5>
6    4004dd:   f3 c3             repz retq
```

Advantages:

1) Instructions can be compactly encoded (requiring just 2 bytes)
2) Object code can be shifted to different positions in memory without alteration

Implementing Conditional Branches with Conditional Control

```
1   int absdiff(int x, int y) {
2       if (x < y)
3           return y - x;
4       else
5           return x - y;
6   }
```

```
1   int gotodiff(int x, int y) {
2       int result;
3       if (x >= y)
4           goto x_ge_y;
5       result =  y - x;
6       goto done;
7    x_ge_y:
8       result = x - y;
9    done:
10      return result;
11  }
```

Implementing Conditional Branches with Conditional Control

(c) Generated assembly code

```
    x at %ebp+8, y at %ebp+12
 1      movl    8(%ebp), %edx     Get x
 2      movl    12(%ebp), %eax    Get y
 3      cmpl    %eax, %edx        Compare x:y
 4      jge     .L2               if >= goto x_ge_y
 5      subl    %edx, %eax        Compute result = y-x
 6      jmp     .L3               Goto done
 7    .L2:                      x_ge_y:
 8      subl    %eax, %edx        Compute result = x-y
 9      movl    %edx, %eax        Set result as return value
10    .L3:                      done: Begin completion code
```

Implementing Conditional Branches with Conditional Control

```
if (test-expr)
    then-statement
else
    else-statement
```

```
    t = test-expr;
    if (!t)
        goto false;
    then-statement
    goto done;
false:
    else-statement
done:
```

Conditional Branches with Conditional Move

(a) Original C code

```
1   int absdiff(int x, int y) {
2       return x < y ? y-x : x-y;
3   }
```

(b) Implementation using conditional assignment

```
1   int cmovdiff(int x, int y) {
2       int tval = y-x;
3       int rval = x-y;
4       int test = x < y;
5       /* Line below requires
6          single instruction: */
7       if (test) rval = tval;
8       return rval;
9   }
```

Conditional Branches with Conditional Move

(c) Generated assembly code

```
    x at %ebp+8, y at %ebp+12
1       movl    8(%ebp), %ecx     Get x
2       movl    12(%ebp), %edx    Get y
3       movl    %edx, %ebx        Copy y
4       subl    %ecx, %ebx        Compute y-x
5       movl    %ecx, %eax        Copy x
6       subl    %edx, %eax        Compute x-y and set as return value
7       cmpl    %edx, %ecx        Compare x:y
8       cmovl   %ebx, %eax        If <, replace return value with y-x
```

Conditional Branches with Conditional Move

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	below or equal (unsigned <=)

Conditional Branches: Do-While Loop

(a) C code

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

(b) Equivalent goto version

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
    long fact_do(long n)
    n in %rdi
1   fact_do:
2     movl    $1, %eax       Set result = 1
3   .L2:                   loop:
4     imulq   %rdi, %rax     Compute result *= n
5     subq    $1, %rdi       Decrement n
6     cmpq    $1, %rdi       Compare n:1
7     jg      .L2            If >, goto loop
8     rep; ret               Return
```

Conditional Branches: While Loop

```
1  int fact_while(int n)
2  {
3      int result = 1;
4      while (n > 1) {
5          result *= n;
6          n = n-1;
7      }
8      return result;
9  }
```

```
1  int fact_while_goto(int n)
2  {
3      int result = 1;
4      if (n <= 1)
5          goto done;
6   loop:
7      result *= n;
8      n = n-1;
9      if (n > 1)
10         goto loop;
11  done:
12     return result;
13 }
```

```
   Argument: n at %ebp+8
   Registers: n in %edx, result in %eax
1     movl    8(%ebp), %edx     Get n
2     movl    $1, %eax          Set result = 1
3     cmpl    $1, %edx          Compare n:1
4     jle     .L7               If <=, goto done
5   .L10:                     loop:
6     imull   %edx, %eax        Compute result *= n
7     subl    $1, %edx          Decrement n
8     cmpl    $1, %edx          Compare n:1
9     jg      .L10              If >, goto loop
10  .L7:                      done:
    Return result
```

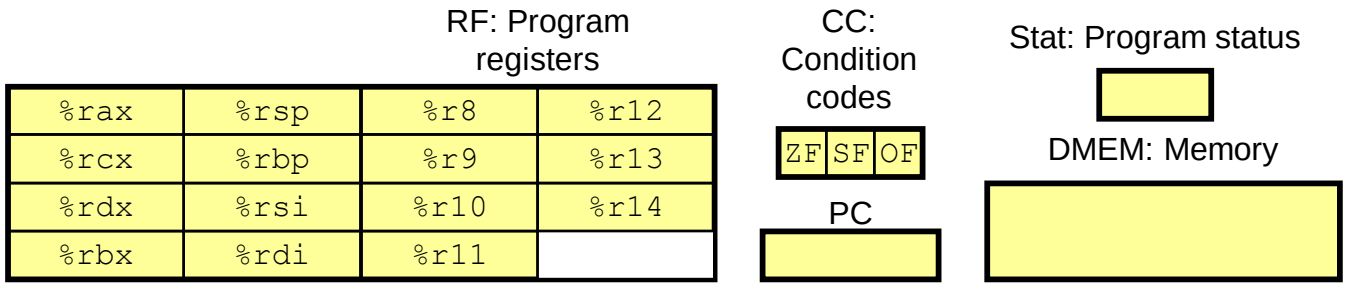
Conditional Branches: Switch Statement

(a) Switch statement

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

    case 100:
        val *= 13;
        break;

    case 102:
        val += 10;
        /* Fall through */

    case 103:
        val += 11;
        break;

    case 104:
    case 106:
        val *= val;
        break;

    default:
        val = 0;
    }
    *dest = val;
}
```

```
    void switch_eg(long x, long n, long *dest)
    x in %rdi, n in %rsi, dest in %rdx
1   switch_eg:
2     subq    $100, %rsi            Compute index = n-100
3     cmpq    $6, %rsi              Compare index:6
4     ja      .L8                   If >, goto loc_def
5     jmp     *.L4(,%rsi,8)         Goto *jg[index]
6   .L3:                          loc_A:
7     leaq    (%rdi,%rdi,2), %rax   3*x
8     leaq    (%rdi,%rax,4), %rdi   val = 13*x
9     jmp     .L2                   Goto done
10  .L5:                          loc_B:
11    addq    $10, %rdi             x = x + 10
12  .L6:                          loc_C:
13    addq    $11, %rdi             val = x + 11
14    jmp     .L2                   Goto done
15  .L7:                          loc_D:
16    imulq   %rdi, %rdi            val = x * x
17    jmp     .L2                   Goto done
18  .L8:                          loc_def:
19    movl    $0, %edi              val = 0
20  .L2:                          done:
21    movq    %rdi, (%rdx)          *dest = val
22    ret                           Return
```

Y86-64 Processor State

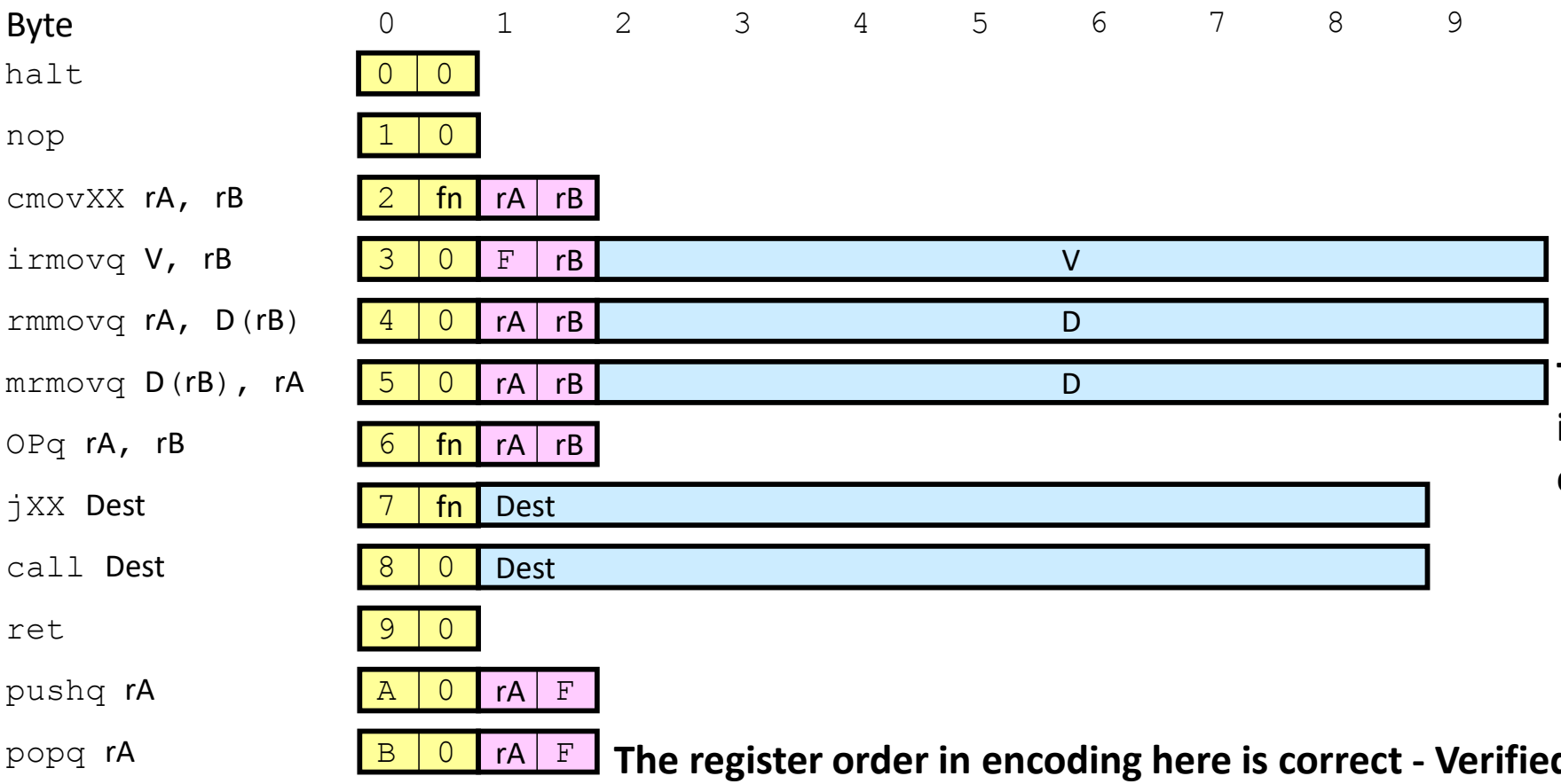


- Program Registers
  - 15 registers (omit `%r15`). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero SF:Negative OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Y86-64 Instructions

- Subset of x86-64 instruction set
  - includes only 8-byte integer operations
  - fewer addressing modes (second index register and scaling not supported)
  - No transfer of immediate data to memory
  - smaller set of operations

Y86-64 Instruction Set #1



The register order in encoding here is correct - Verified

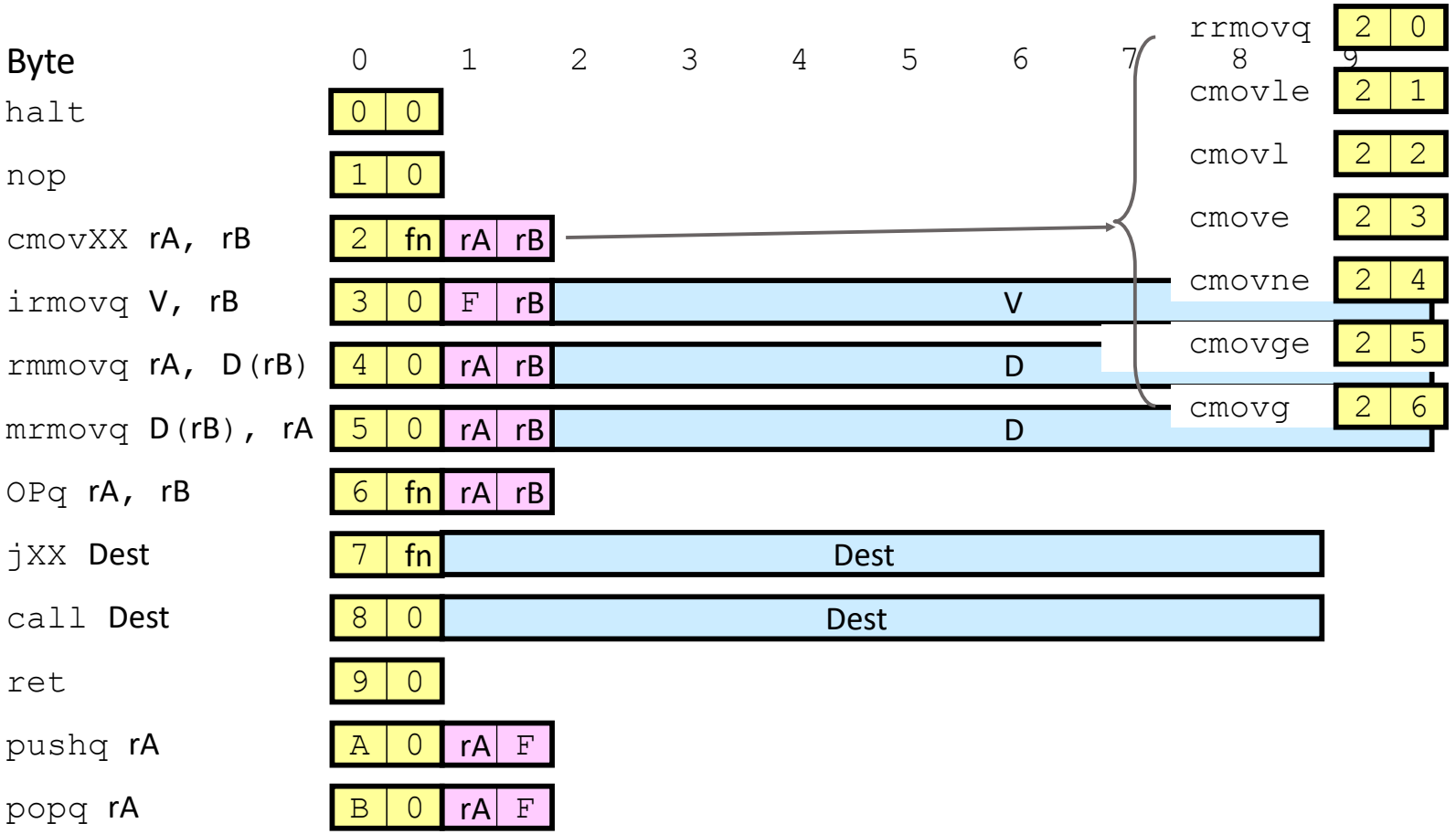
The register order in encoding here is correct - Verified

Y86-64 Instructions

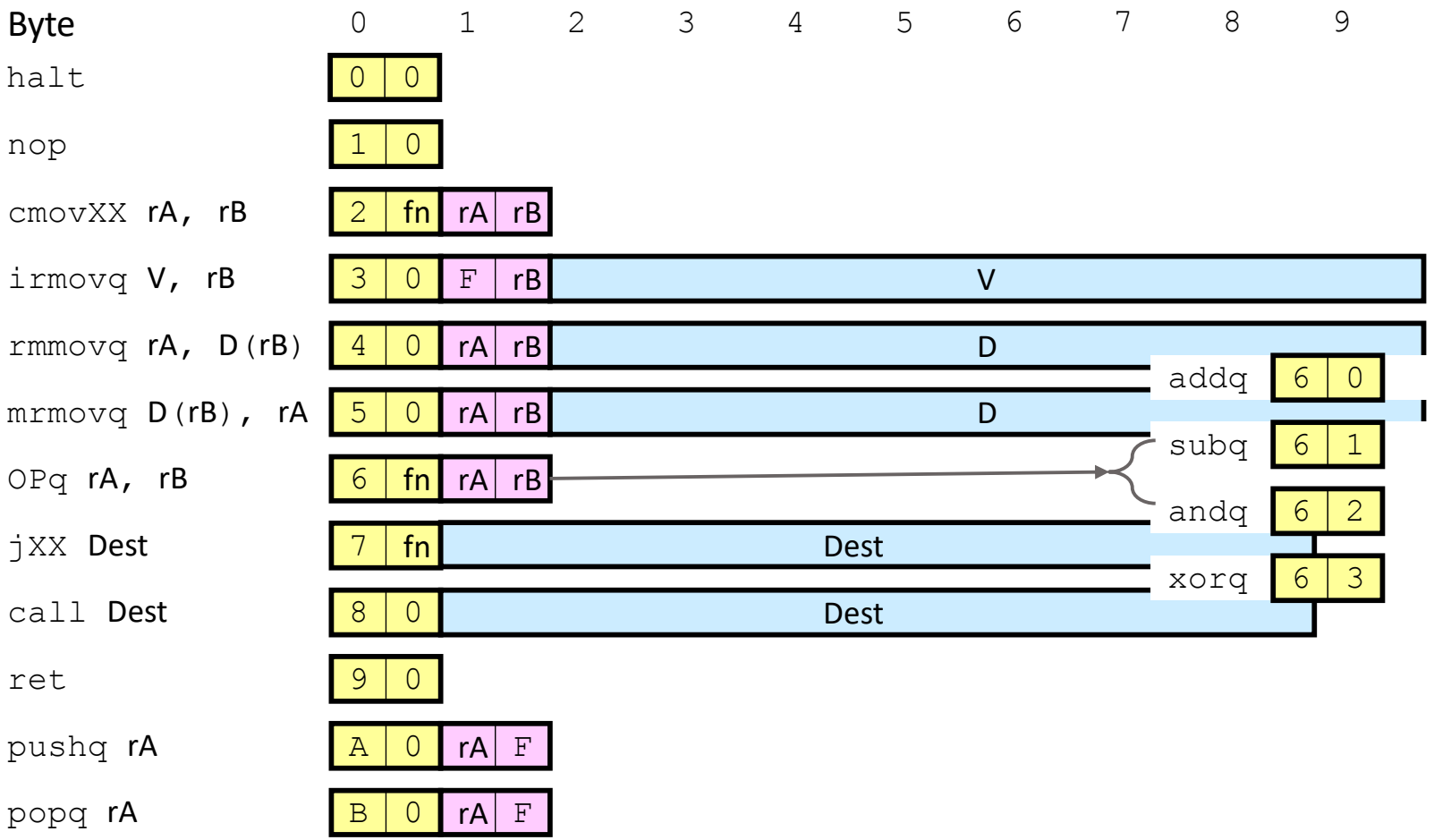
Format

- 1–10 bytes of information read from memory
  - Can determine instruction length from first byte
  - Not as many instruction types, and simpler encoding than with x86-64
- Each accesses and modifies some part(s) of the program state

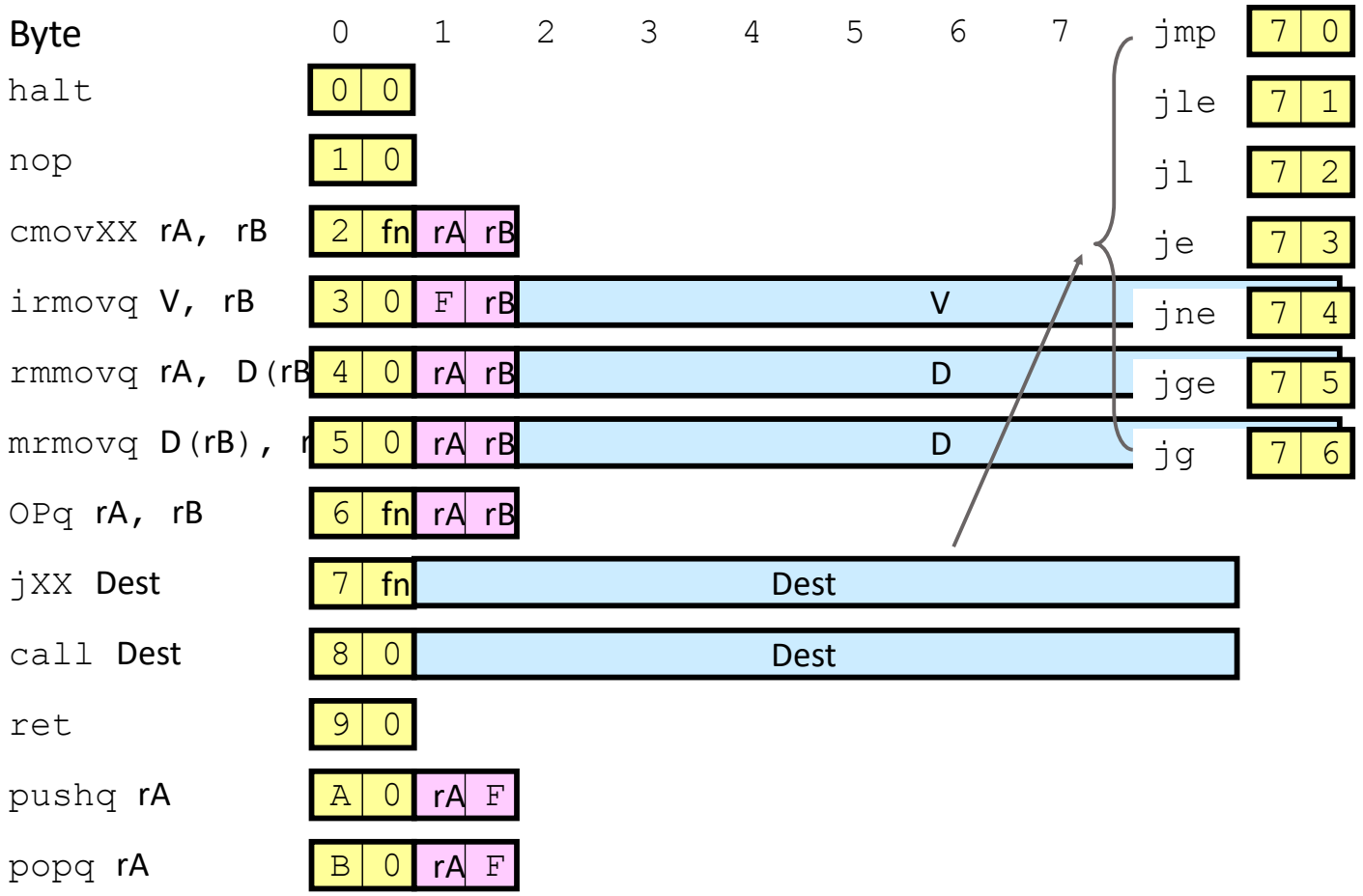
Y86-64 Instruction Set #2



Y86-64 Instruction Set #3



Y86-64 Instruction Set #4



Encoding Registers

Each register has 4-bit ID

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

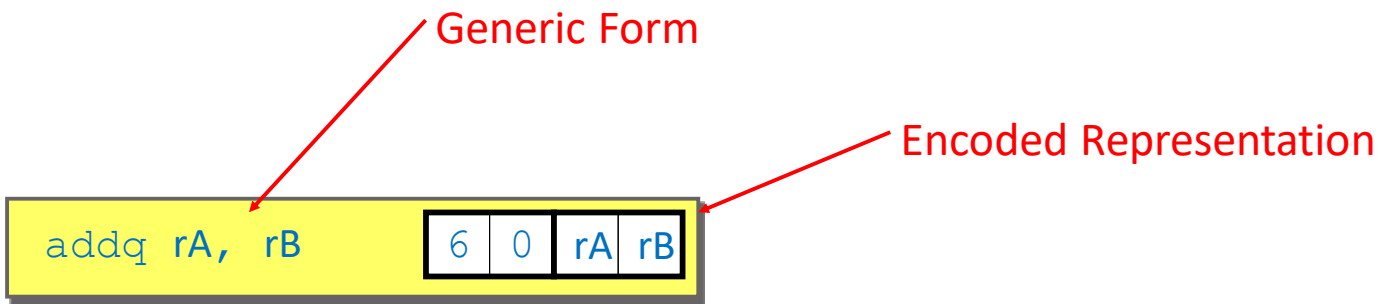
- Same encoding as in x86-64

Register ID 15 (`0xF`) indicates “no register”

- Will use this in our hardware design in multiple places

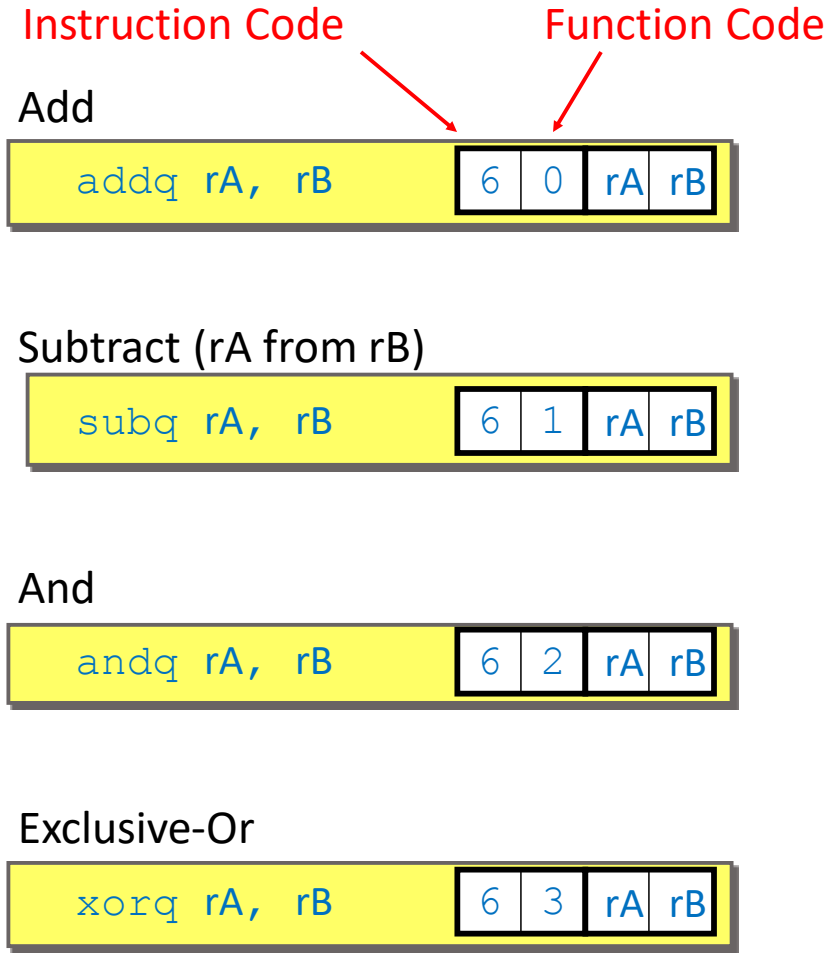
Instruction Example

Addition Instruction



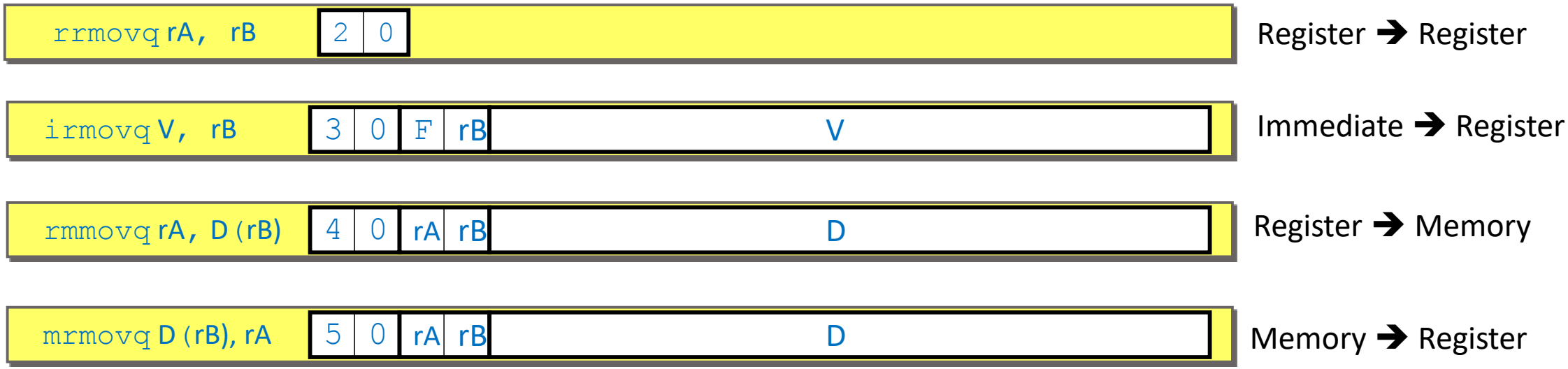
- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., `addq %rax,%rsi` Encoding: `60 06`
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

Arithmetic and Logical Operations



- Refer to generically as "`OPq`"
- Encodings differ only by "function code"
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

Move Operations



- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
Encoding:	`30 82 cd ab 00 00 00 00 00 00`
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
Encoding:	`20 43`
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
Encoding:	`50 15 f4 ff ff ff ff ff ff ff`
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
Encoding:	`40 64 1c 04 00 00 00 00 00 00`

Conditional Move Instructions

Move Unconditionally

Move When Less or Equal

Move When Less

Move When Equal

Move When Not Equal

Move When Greater or Equal

Move When Greater

- Refer to generically as "`cmovXX`"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - (Conditionally) copy value from source to destination register

Jump Instructions

Jump (Conditionally)

- Refer to generically as "jXX"
- Encodings differ only by "function code" fn
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in x86-64

Jump Instructions

Jump Unconditionally

Jump When Less or Equal

Jump When Less

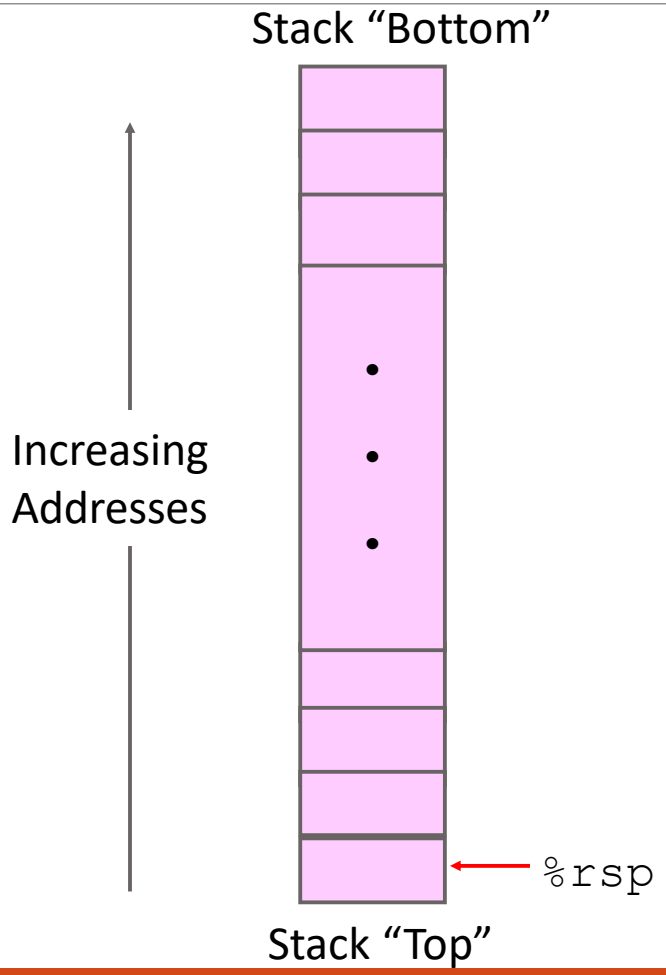
Jump When Equal

Jump When Not Equal

Jump When Greater or Equal

Jump When Greater

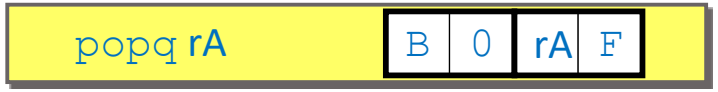
Y86-64 Program Stack



- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by `%rsp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer

Stack Operations

- Decrement `%rsp` by 8
- Store word from rA to memory at `%rsp`
- Like x86-64



- Read word from memory at `%rsp`
- Save in rA
- Increment `%rsp` by 8
- Like x86-64

Subroutine Call and Return

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

- Pop value from stack
- Use as address for next instruction
- Like x86-64

Miscellaneous Instructions

- Don’t do anything

- Stop executing instructions
- x86-64 has comparable instruction, but can’t execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code
AOK	1

- Normal operation

Mnemonic	Code
HLT	2

- Halt instruction encountered

Mnemonic	Code
ADR	3

- Bad address (either instruction or data) encountered

Mnemonic	Code
INS	4

- Invalid instruction encountered

Desired Behavior

- If AOK, keep going
- Otherwise, stop program execution

Y86-64 program

```
1   long sum(long *start, long count)
2   {
3       long sum = 0;
4       while (count) {
5           sum += *start;
6           start++;
7           count--;
8       }
9       return sum;
10  }
```

x86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
1   sum:
2     movl    $0, %eax          sum = 0
3     jmp     .L2               Goto test
4   .L3:                      loop:
5     addq    (%rdi), %rax      Add *start to sum
6     addq    $8, %rdi          start++
7     subq    $1, %rsi          count--
8   .L2:                      test:
9     testq   %rsi, %rsi        Test sum
10    jne     .L3               If !=0, goto loop
11    rep; ret                  Return
```

Y86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
1   sum:
2     irmovq $8,%r8           Constant 8
3     irmovq $1,%r9           Constant 1
4     xorq %rax,%rax          sum = 0
5     andq %rsi,%rsi          Set CC
6     jmp     test            Goto test
7   loop:
8     mrmovq (%rdi),%r10      Get *start
9     addq %r10,%rax          Add to sum
10    addq %r8,%rdi           start++
11    subq %r9,%rsi           count--.  Set CC
12  test:
13    jne     loop            Stop when 0
14    ret                     Return
```

Summary

Y86-64 Instruction Set Architecture

- Similar state and instructions as x86-64
- Simpler encodings
- Somewhere between CISC and RISC

How Important is ISA Design?

- Less now than before
  - With enough hardware, can make almost anything go fast

Thank You!