CS429: Computer Organization and Architecture

Instruction Set Architecture II

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: February 26, 2020 at 14:01

Topics of this Slideset

- Assembly Programmer's Execution Model
- Accessing Information
- Registers
- Memory
- Arithmetic operations

BTW: We're through with Y86 for a while, and starting the x86. We'll come back to the Y86 later for pipelining.

Intel x86 Processors

x86 processors totally dominate the laptop/desktop/server market.

Evolutionary Design

- Starting in 1978 with 8086
- Added more features over time.

Complex Instruction Set Computer (CISC)

- Still support many old, now obsolete, features.
- There are many different instructions with many different formats, but only a small subset are encountered with Linux programs.
- Hard to match performance of Reduced Instruction Set Computers (RISC), though Intel has done just that!

x86 Evolution: Programmer's View

Machine Evolution

Model	Date	Trans.
386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
Pentium Pro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M
Core i7-8086K	2018	3B

Added Features

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

x86 Clones: Advanced Micro Devices (AMD)

Historically

- AMD has followed behind Intel
- A little bit slower, a lot cheaper

Then

- Recruited top circuit designers from Digital Equipment Corp. (DEC) and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

Recent Years

- Intel got its act together; leads the world in semiconductor technology
- AMD has fallen behind; relies on external semiconductor manufacturers

x86 Evolution: Clones

Transmeta

Radically different approach to implementation.

- Translate x86 code into "very long instruction word" (VLIW) code.
- Very high degree of parallelism.

Centaur / Via

- Continued evolution from Cyrix, the 3rd x86 vendor. Low power, design team in Austin.
- 32-bit processor family.
  - At 2 GHz, around 2 watts; at 600 MHz around 0.5 watt.
- 64-bit processor family, used by HP, Lenovo, OLPC, IBM.
  - Very low power, only a few watts at 1.2 GHz.
  - Full virtualization and SSE support.

Definitions:

Architecture: (also ISA or instruction set architecture). The parts of a processor design one needs in order to understand or write assembly/machine code.

- Examples: instruction set specification, registers

Microarchitecture: implementation of the architecture.

- Examples: cache sizes and core frequency

Code Forms:

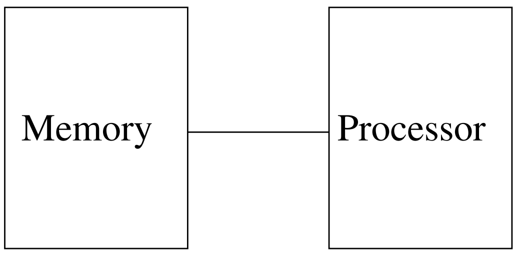
- Machine code: the byte-level programs that a processor executes
- Assembly code: a human-readable textual representation of machine code

Example ISAs:

- Intel: x86, IA32, Itanium, x86-64
- ARM: used in almost all mobile phones

Machine Models

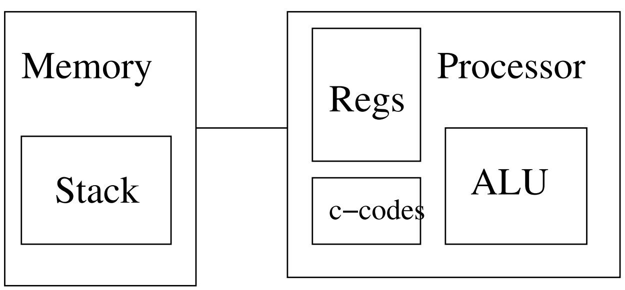
C



Data
1) char
2) int, float
3) double
4) struct, array
5) pointer

Control
1) loops
2) conditionals
3) switch
4) proc. call
5) proc. return

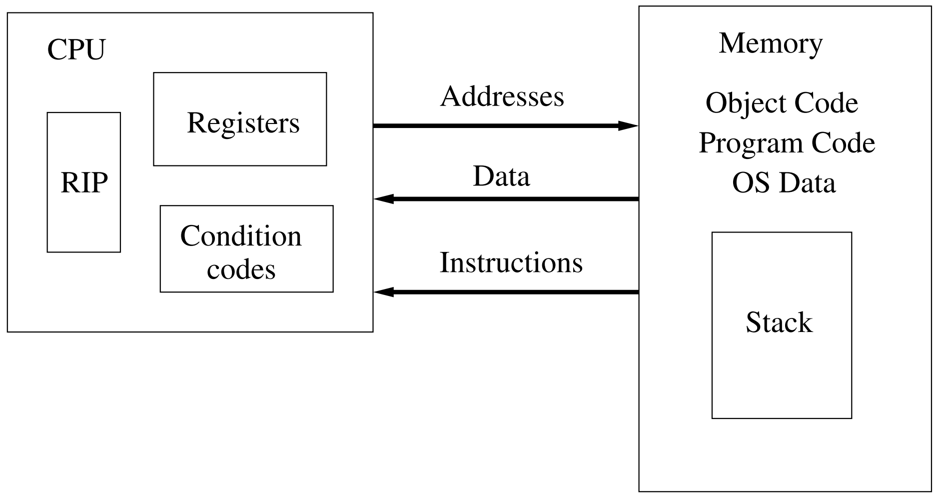
Assembly



1) byte
2) 2-byte word
3) 4-byte long word
4) 8-byte quad word
5) contiguous byte allocation
6) address of initial byte

1) branch/jump
2) call
3) ret

Assembly Programmer's View



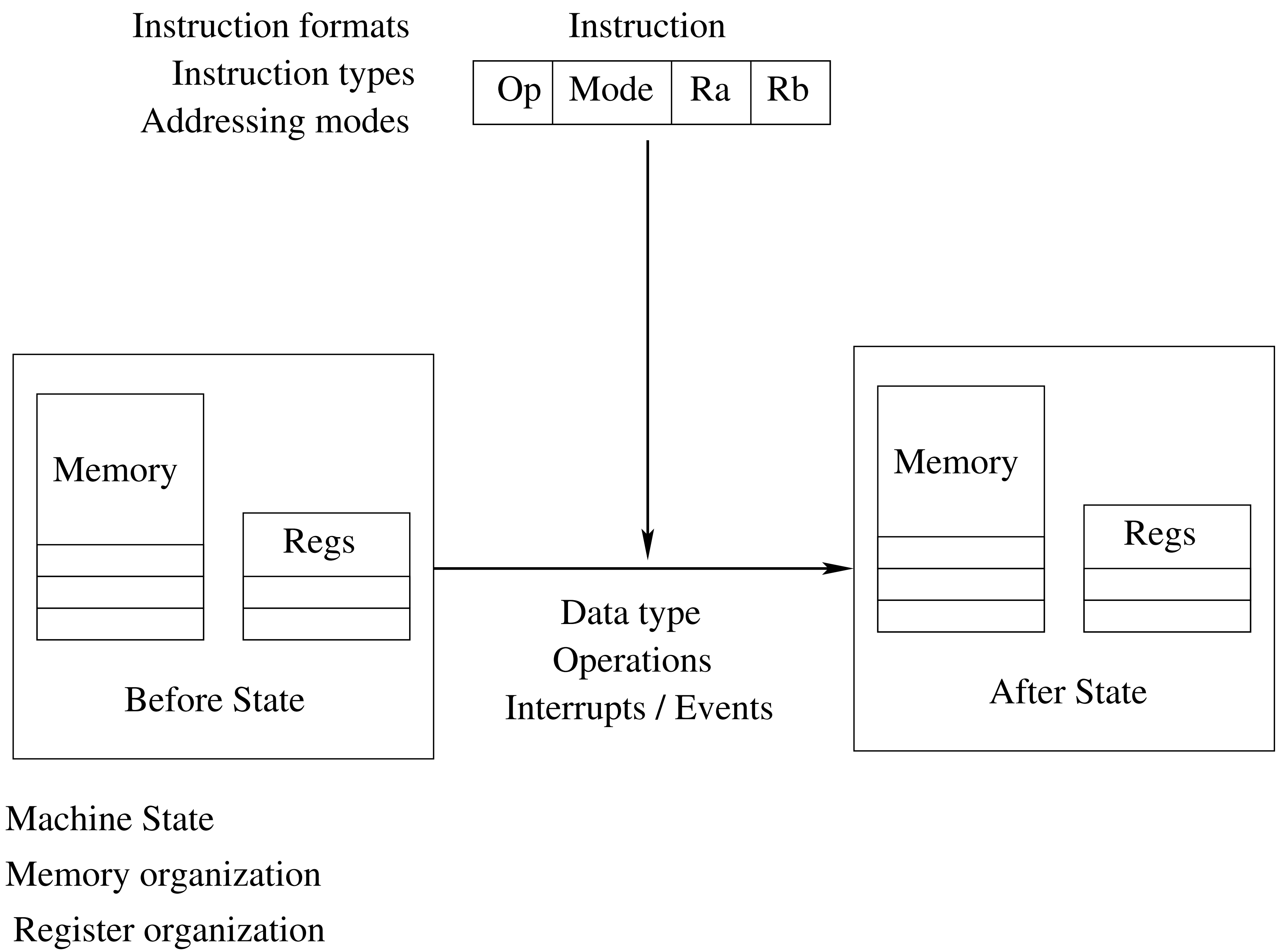
Programmer Visible State

- PC (Program Counter): address of next instruction. Called `%rip` in x86-64.
- Condition codes:
  - Store status info about most recent arithmetic operation.
  - Used for conditional branching.
- Register file: heavily used program data.
- Memory
  - Byte addressable array.
  - Code, user data, (some) OS data.
  - Includes stack.

ISA Principles

- Contract between programmer and the hardware.
  - Defines visible state of the system.
  - Defines how state changes in response to instructions.
- For Programmer: ISA is model of how a program will execute.
- For Hardware Designer: ISA is formal definition of the correct way to execute a program.
  - With a stable ISA, SW doesn't care what the HW looks like under the hood.
  - Hardware implementations can change drastically.
  - As long as the HW implements the same ISA, all prior SW should still run.
  - Example: x86 ISA has spanned many chips; instructions have been added but the SW for prior chips still runs.
- ISA specification: the binary encoding of the instruction set.

ISA Basics


Instruction formats
Instruction types
Addressing modes
Instruction
Op
Mode
Ra
Rb
Memory
Regs
Before State
Data type
Operations
Interrupts / Events
Memory
Regs
After State
Machine State
Memory organization
Register organization

Architecture vs. Implementation

Architecture: defines *what* a computer system does in response to a program and set of data.

- *Programmer visible* elements of computer system.

Implementation (microarchitecture): defines *how* a computer does it.

- Sequence of steps to complete operations.
- Time to execute each operation.
- Hidden "bookkeeping" function.

If the architecture changes, some programs may no longer run or return the same answer. If the implementation changes, some programs may run faster/slower/better, but the answers won't change.

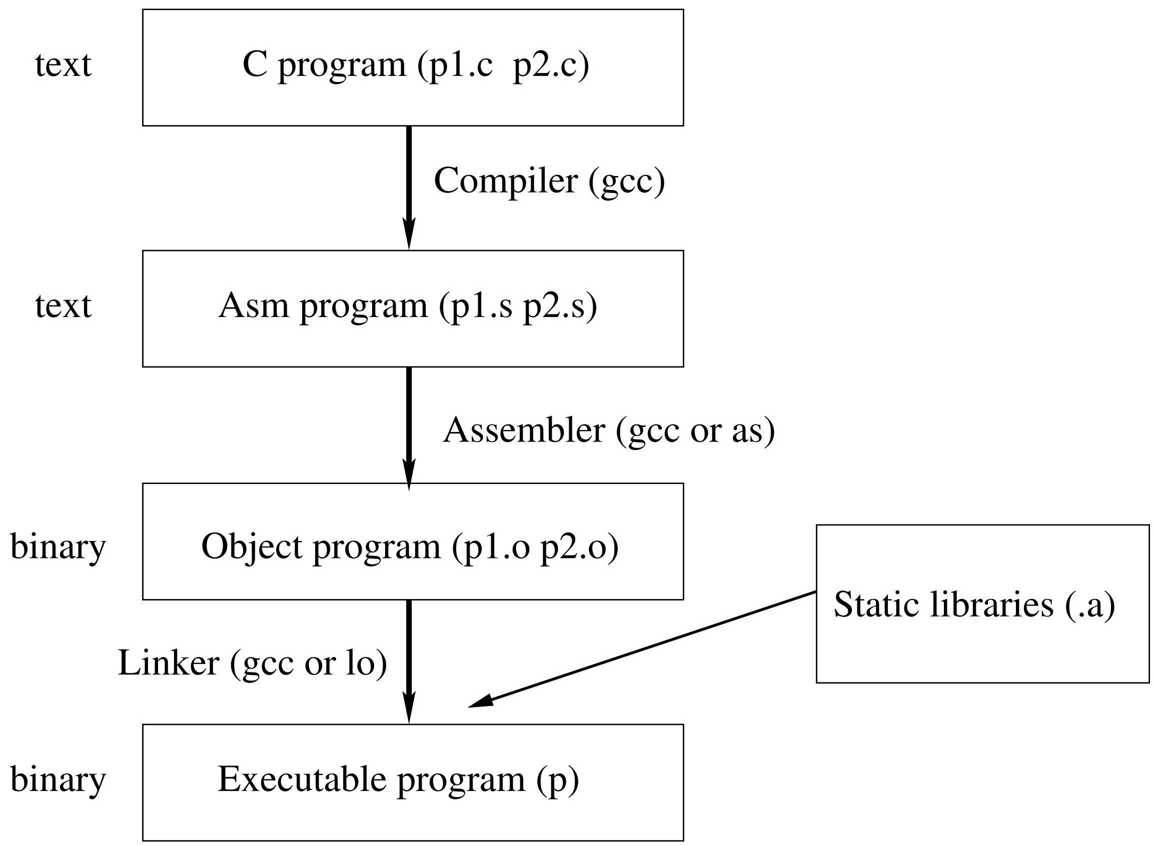
Examples

Which of the following are part of the architecture and which are part of the implementation? Hint: if the programmer can see/use it (directly) in a program, it's part of the architecture.

- Number/names of general purpose registers
- Width of memory bus
- Binary representation of each instruction
- Number of cycles to execute a FP instruction
- Condition code bits set by a `move` instruction
- Size of the instruction cache
- Type of FP format

Turning C into Object Code

- Code in files: `p1.c, p2.c`
- For minimal optimization, compile with command:
  `gcc -Og p1.c p2.c -o p`
- Use optimization (`-Og`); new to recent versions of gcc
- Put resulting binary in file `p`



Compiling into Assembly

C Code (sum.c):

```
long plus(long x, long y);

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

Run command: `gcc -Og -S sum.c`
produces file `sum.s`.

```
sumstore:
   pushq %rbx                # save %rbx
   movq  %rdx, %rbx          # temp <-- dest
   call  plus
   movq  %rax, (%rbx)        # *dest <-- t
   popq  %rbx                # restore %rbx
   ret
```

Warning: you may get different results due to variations in gcc and compiler settings.

Assembly Characteristics

Minimal Data Types

- "Integer" data of 1, 2, 4 or 8 bytes
- Addresses (untyped pointers)
- Floating point data of 4, 8 or 10 bytes
- No aggregate types such as arrays or structures
- Just contiguously allocated bytes in memory

Primitive Operations

- Perform arithmetic functions on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

```
0x0400595:
    0x53
    0x48     # total of
    0x89     # 14 bytes
    0xd3
    0xe8     # each
        inst
    0xf2     # 1, 3, or
    0xff     # 5 bytes
    0xff
    0xff     # starts
        at
    0x48     # addr
    0x89     # 0
        x0x00595
    0x03
    0x5b
    0xc3
```

Assembler

- Translates .s into .o
- Binary encoding of each inst.
- Nearly complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries; e.g., code for `malloc`, `printf`
- Some libraries are dynamically linked (just before execution)

Machine Instruction Example

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x040059e:    48 89 03
```

C Code

- Store value `t` where designated by `dest`

Assembly

- Move 8-byte value to memory (quad word in x86 parlance).
- Operands:

  t: Register `%rax`
  dest: Register `%rbx`
  *dest: Memory `M[%rbx]`

Object Code

- 3-byte instruction
- Stored at address `0x40059e`

Disassembling Object Code

Disassembly using `objdump`. Offsets are relative.

```
> objdump -d sumstore.o

sumstore.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <sumstore>:
   0:   53                      push   %rbx
   1:   48 89 d3                mov    %rdx,%rbx
   4:   e8 00 00 00 00          callq  9 <sumstore+0x9>
   9:   48 89 03                mov    %rax,(%rbx)
   c:   5b                      pop    %rbx
   d:   c3                      retq
```

- `objdump -d sum`
- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a.out (complete executable) or .o file

Alternate Disassembly

Disassembly using gdb. Offsets are relative.

```
Dump of assembler code for function sumstore:
   0x0000000000000000 <+0>:   push   %rbx
   0x0000000000000001 <+1>:   mov    %rdx,%rbx
   0x0000000000000004 <+4>:   callq  0x9 <sumstore+9>
   0x0000000000000009 <+9>:   mov    %rax,(%rbx)
   0x000000000000000c <+12>:  pop    %rbx
   0x000000000000000d <+13>:  retq
End of assembler dump.
```

Within gdb debugger:

```
gdb sum
disassemble sumstore
x/14xb sumstore
```

Examine the 14 bytes starting at sumstore.

What Can be Disassembled?

- Anything that can be interpreted as executable code.
- Disassembler examines bytes and reconstructs assembly source.

```
% objdump -d WINWORD.EXE

WINWORD.EXE:        file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55                push %ebp
30001001:  8b ec             mov  %esp, %ebp
30001003:  6a ff             push $0xffffffff
30001005:  68 90 10 00 30    push $0x30001090
3000100a:  68 91 dc 4c 30    push $0x304cdc91
```

Which Assembler?

Intel/Microsoft Format

```
lea rax, [rcx+rcx*4]
sub rsp, 8
cmp quad ptr[ebp-8], 0
mov rax, quad ptr[rax*4+10h]
```

GAS/Gnu Format

```
leaq (%rcx,%rcx,4), %rax
subq $8,%rsp
cmpq $0,-8(%rbp)
movq $0x10(,%rax,4),%rax
```

Intel/Microsoft Differs from GAS

- Operands are listed in opposite order:

  `mov` Dest, Src   `movq` Src, Dest

- Constants not preceded by '$'; denote hex with 'h' at end.

  `10h`   `$0x10`

- Operand size indicated by operands rather than operator suffix.

  `sub`   `subq`

- Addressing format shows effective address computation.

  `[rax*4+10h]`   `$0x10(,%rax,4)`

From now on we'll always use GAS assembler format.

x86-64 Integer Registers

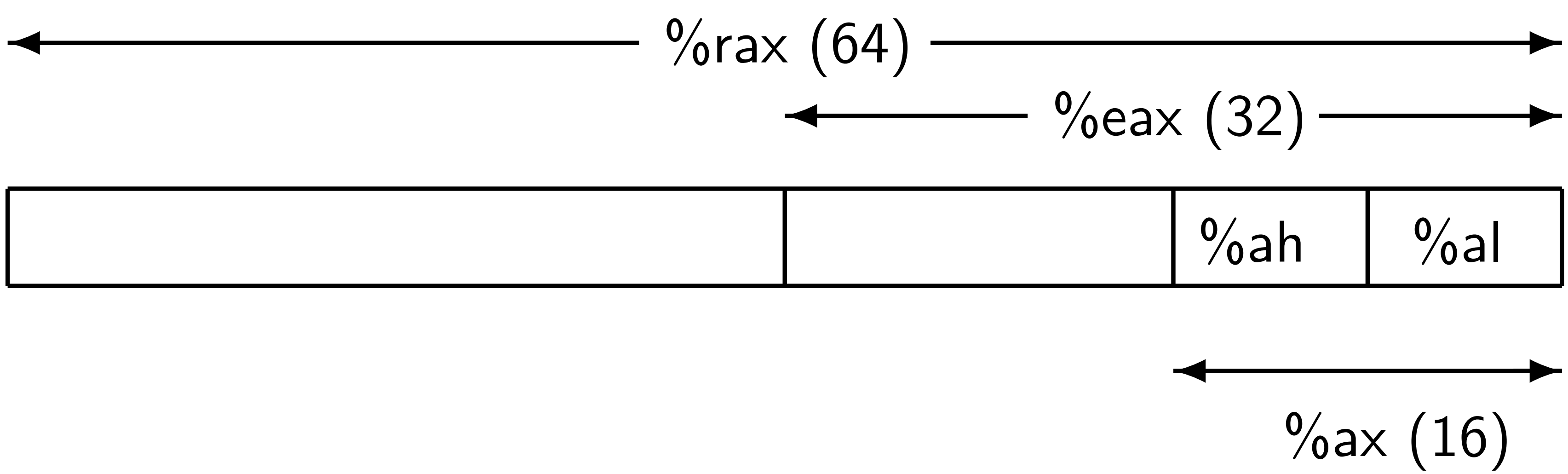
For each of the 64-bit registers, the LS 4 bytes are named 32-bit registers.

Reg.	**LS 4 bytes**	**Reg.**	**LS 4 bytes**
`%rax`	`%eax`	`%r8`	`%r8d`
`%rbx`	`%ebx`	`%r9`	`%r9d`
`%rcx`	`%ecx`	`%r10`	`%r10d`
`%rdx`	`%edx`	`%r11`	`%r11d`
`%rsi`	`%esi`	`%r12`	`%r12d`
`%rdi`	`%edi`	`%r13`	`%r13d`
`%rsp`	`%esp`	`%r14`	`%r14d`
`%rbp`	`%ebp`	`%r15`	`%r15d`

You can also reference the LS 16-bits (2 bytes) and LS 8-bits (1 byte). For the numbered registers (%r8–%r15) the components are named e.g., %r8d (32-bits), %r8w (16-bits), %r8b (8-bits).

Decomposing the %rax Register

All of the x86's 64-bit registers have 32-bit, 16-bit and 8-bit accessible internal structure. It varies slightly among the different registers. Example, only `%rax`, `%rbx`, `%rcx`, `%rdx` allow direct access to byte 1 (`%ah`).



Some History: IA32 Registers

32-bit reg	16-bit reg	8-bit reg	8-bit Reg	Use
%eax	%ax	%ah	%al	accumulator
%ecx	%cx	%ch	%cl	counter
%edx	%dx	%dh	%dl	data
%ebx	%bx	%bh	%bl	base
%esi	%si		%sil*	source index
%edi	%di		%dil*	dest. index
%esp	%sp		%spl*	stack pointer
%ebp	%bp		%bpl*	base pointer

*These are only available in 64-bit mode.

Simple Addressing Modes (Same as Y86)

- **Immediate:** `value`

```
movq  $0xab, %rbx
```

- **Register:** `Reg[R]`

```
movq  %rcx, %rbx
```

- **Normal (R):** `Mem[Reg[R]]`
  - Register R specifies memory address.
  - This is often called *indirect* addressing.
  - 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(%rcb),%rdx
```

Moving Data

Moving Data:

- Form: `movq` Source, Dest
- Move 8-byte "long" word
- Lots of these in typical code

Operand Types

- **Immediate**: Constant integer data
  - Like C constant, but prefixed with '$'
  - E.g., `$0x400`, `$-533`
  - Encoded with 1, 2, or 4 bytes
- **Register**: One of 16 integer registers
  - Example: `%rax`, `%r13`
  - But `%rsp` is reserved for special use
  - Others have special uses for particular instructions
- **Memory**: source/dest is first address of block
  - Example: `(%rax)`, `0x20(%rbx)`
  - Various "addressing modes"

movq Operand Combinations

Unlike the Y86, we don't distinguish the operator depending on the operand addressing modes.

Source	Dest.	Assembler	C Analog
Immediate	Register	`movq  $0x4,%rax`	`temp = 0x4;`
Immediate	Memory	`movq $-147,(%rax)`	`*p = -147;`
Register	Register	`movq %rax,%rdx`	`temp2 = temp1;`
Register	Memory	`movq %rax,(%rdx)`	`*p = temp;`
Memory	Register	`movq (%rax),%rdx`	`temp = *p`

Direct memory-memory transfers are not supported.

Addresses and Pointers in C

C programming model is close to machine language.

- Machine language manipulates memory addresses.
  - For address computation;
  - To store addresses in registers or memory.
- C employs pointers, which are just addresses of primitive data elements or data structures.

Examples of operators * and &:

- `int a, b;` /* declare integers a and b */
- `int *a_ptr;` /* a is a pointer to an integer */
- `a_ptr = a;` /* illegal, types don't match*/
- `a_ptr = &a;` /* a_ptr holds address of a */
- `b = *a_ptr;` /* dereference a_ptr and assign value to b */

Using Simple Addressing Modes

```c
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 (1)

```
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
```

Register	Value	comment
%rdi	xp	points into memory
%rsi	yp	points into memory
%rax	t0	temporary storage
%rdx	t1	temporary storage

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

Initial State:

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

Value	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

Understanding Swap (3)

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

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

Memory

123	0x120
	0x118
	0x110
	0x108
456	0x100

Understanding Swap (4)

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

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

123	0x120
	0x118
	0x110
	0x108
456	0x100

Understanding Swap (5)

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

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

456	0x120
	0x118
	0x110
	0x108
456	0x100

Understanding Swap (6)

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

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

456	0x120
	0x118
	0x110
	0x108
123	0x100

Simple Addressing Modes

- **Immediate:** `value`

```
movq  $0xab, %rbx
```

- **Register:** `Reg[R]`

```
movq  %rcx, %rbx
```

- **Normal (R):** `Mem[Reg[R]]`
  - Register R specifies memory address.
  - This is often called *indirect* addressing.
  - 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(%rcb),%rdx
```

Indexed Addressing Modes

Most General Form:

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

- D: Constant "displacement" of 1, 2, 4 or 8 bytes
- Rb: Base register, any of the 16 integer registers
- Ri: Index register, any except `%rsp` (and probably not `%rbp`)
- S: Scale, must be 1, 2, 4 or 8.

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]]
```

Addressing Modes

Type	Form	Operand value	Name
Immediate	\$D	D	Immediate
Register	E_a	$R[E_a]$	Register
Memory	D	$M[D]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$D(E_b)$	$M[D + R[E_b]]$	Base + displacement
Memory	(E_b, E_i),	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$D(E_b, E_i)$,	$M[D + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$D(, E_i, s)$	$M[D + R[E_i] \cdot s]$	Scaled indexed
Memory	(E_b, E_i, s),	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$D(E_b, E_i, s)$	$M[D + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

The scaling factor s *can only be* 1, 2, 4, or 8.

Address Computation Example

%rdx	0xf000
%rcx	0x100

Expression	**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
0x80(%rdx, 2)	Illegal. Why?	
0x80(,%rdx, 3)	Illegal. Why?	

Addressing Mode Example

Indexed addressing modes are extremely useful when iterating over an array.

```
long sumArray ( long A[], int len) {
  long i;
  long sum = 0;

  for (i = 0; i < len; i++)
    sum += A[i];
  return sum;
}
```

- What is the type of A?
- Why do we need len? Could we just call `len(A)`?

Addressing Mode Example

`> gcc -S -Og test.c`

causes `sumArray` on the previous slide to compile to:

```
sumArray:
        movl    $0, %eax
        movl    $0, %edx
        jmp     .L2
.L3:
        addq    (%rdi,%rdx,8), %rax
        addq    $1, %rdx
.L2:
        movslq  %esi, %rcx
        cmpq    %rcx, %rdx
        jl      .L3
        rep ret
```

Another Example

Suppose we want to add `val` to each of the elements of array.

```
# include <stdio.h>

void addnum ( int array[], int val, int len ) {
  int i;
  for (i = 0; i < len; i++)
    array[i] += val;
}
```

Assembly

```
addnum:
        movl    $0, %eax
        jmp     .L2
.L3:
        movslq  %eax, %rcx
        addl    %esi, (%rdi,%rcx,4)
        addl    $1, %eax
.L2:
        cmpl    %edx, %eax
        jl      .L3
        rep ret
```

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`	same as `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!
- There's no distinction between signed and unsigned. Why?
- For shift operations Src must be a constant or `%cl`.

Some Arithmetic Operations

One operand instructions:

Format	**Computation**
`incq` Dest	`Dest = Dest + 1`
`decq` Dest	`Dest = Dest - 1`
`negq` Dest	`Dest = -Dest`
`notq` Dest	`Dest = ~Dest`

More instructions in the book.

Address Computation Instruction

Form: `leaq  Src, Dest`

- Src is address mode expression.
- Sets Dest to *address* denoted by the expression

LEA stands for “load effective address.”

After the effective address computation, place the *address*, not the contents of the address, into the destination.

Address Computation Instruction: movq vs. leaq

Consider the following computation:

Reg.	Value
%rax	0x100
%rbx	0x200

```
movq   0x10(%rbx, %rax, 4), %rcx
leaq   0x10(%rbx, %rax, 4), %rdx
```

After this sequence,

- `%rcx` will contain the *contents* of location `0x610`;
- `%rdx` will contain the number (address) 0x610.

Neither LEA nor MOV set condition codes.

What should the following do?

```
leaq   %rbx, %rdx
```

Address Computation Instruction: movq vs. leaq

Consider the following computation:

Reg.	Value
%rax	0x100
%rbx	0x200

```
movq   0x10(%rbx, %rax, 4), %rcx
leaq   0x10(%rbx, %rax, 4), %rdx
```

After this sequence,

- %rcx will contain the *contents* of location 0x610;
- %rdx will contain the number (address) 0x610.

Neither LEA nor MOV set condition codes.

What should the following do?

```
leaq   %rbx, %rdx
```

It really shouldn't be legal since %rbx doesn't have an address. However, the semantics makes it equal to `movq %rbx, %rdx`.

Address Computation Instruction

The `leaq` instruction is widely used for address computations *and* for some general arithmetic computations.

Uses:

- Computing address without doing a memory reference:
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form $x + k \times y$, where $k \in \{1, 2, 4, 8\}$

Example:

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

Converted to ASM by compiler:

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

Arithmetic Expression Example

```c
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 our 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

ISA II: Summary

History of Intel processors and architectures

- Evolutionary design leads to many quirks and artifacts

C, assembly, machine code

- New forms of visible state: program counter, registers, etc.
- Compiler must transform statements, expressions, procedures into low-level instruction sequences

Assembly Basics: Registers, operands, move

- The x86-64 move instructions cover a wide range of data movement forms

Arithmetic

- C compiler will figure out different instruction combinations to carry out computation