03_03_Machine-Level Representation of Programs

why we shouldn't use assembly

- compilers generate pretty fast, efficient code
- tedious, easy to screw up
- not portable

why you shouldn’t use assembly

- … and for that matter, why for a lot of things you shouldn’t be using C either …

- Corbató's Law:

  "The number of lines of code a programmer can write in a fixed period of time is the same independent of the language used."

why we're doing assembly here

- learn how the machine works
- what if no compiler?
- processor features not easily accessed by highers level language

the point

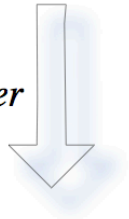
- we’re not trying to prepare you for a job doing assembly programming
- it’s about learning how computers work

how

- look at the assembly generated by GCC
- write some of our own assembly from scratch

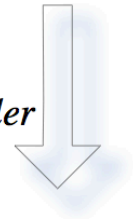
```
int sum(int a, int b, int c)
{
  return a+b+c;
}
```

compiler



```
   .text
.globl _sum
_sum:
   pushl %ebp
   movl  %esp, %ebp
   subl  $8, %esp
   movl  12(%ebp), %eax
   addl  8(%ebp), %eax
   addl  16(%ebp), %eax
   leave
   ret
```

assembler



```
01010101100010011110010110000011111011000000100010001011010001010000110000000011010001010000100000000011010001010001000011001001110000110000000000000000000000000000000000010000000000000000000000000000000000011110000000010000000000000000000000000000000000000000000000000000000000000000000010111110111001101110101011011010000000000000000000000000
```

instruction set

- set of of operations that a processor supports
- examples
  - load *x* bytes from this address into register *y*
  - add what's in register *i* to what's in register *j*
- primitive stuff
- usually takes lots of these primitive ops to do something really useful

Other instruction sets

- IA32, Intel64 (x86-64 or x64), IA64
- SPARC
- ARM
- PowerPC
- MIPS
- Alpha
- JVM

- we're using: IA32
  - not easiest, but popular
  - focusing on GCC output on Linux

some terminology

- x86 name for the chips
- IA32 the name of the instruction set
  - IA = **I**ntel **A**rchitecture

- note difference between:
  - **architecture**: what you need to know to program assembly -- instruction set, registers
  - **microarchitecture**: *implementation*
  - *e.g.*, IA32 on non-Intel chips (*e.g.* AMD)

some tools

- compiler – GCC
- assembler – as *aka* gas
- linker – ld
- debugger – gdb
- disassembler - objdump
- profiler - gprof

Some History: Why should we care?

- Important things to take away from the history lesson in the chapter:
  - Moore's law
  - Evolution of register names
  - Backward compatability:
    - Goal: run progs compiled for earlier versions of chip
    - But: old baggage support features that new OS, compilers rarely use

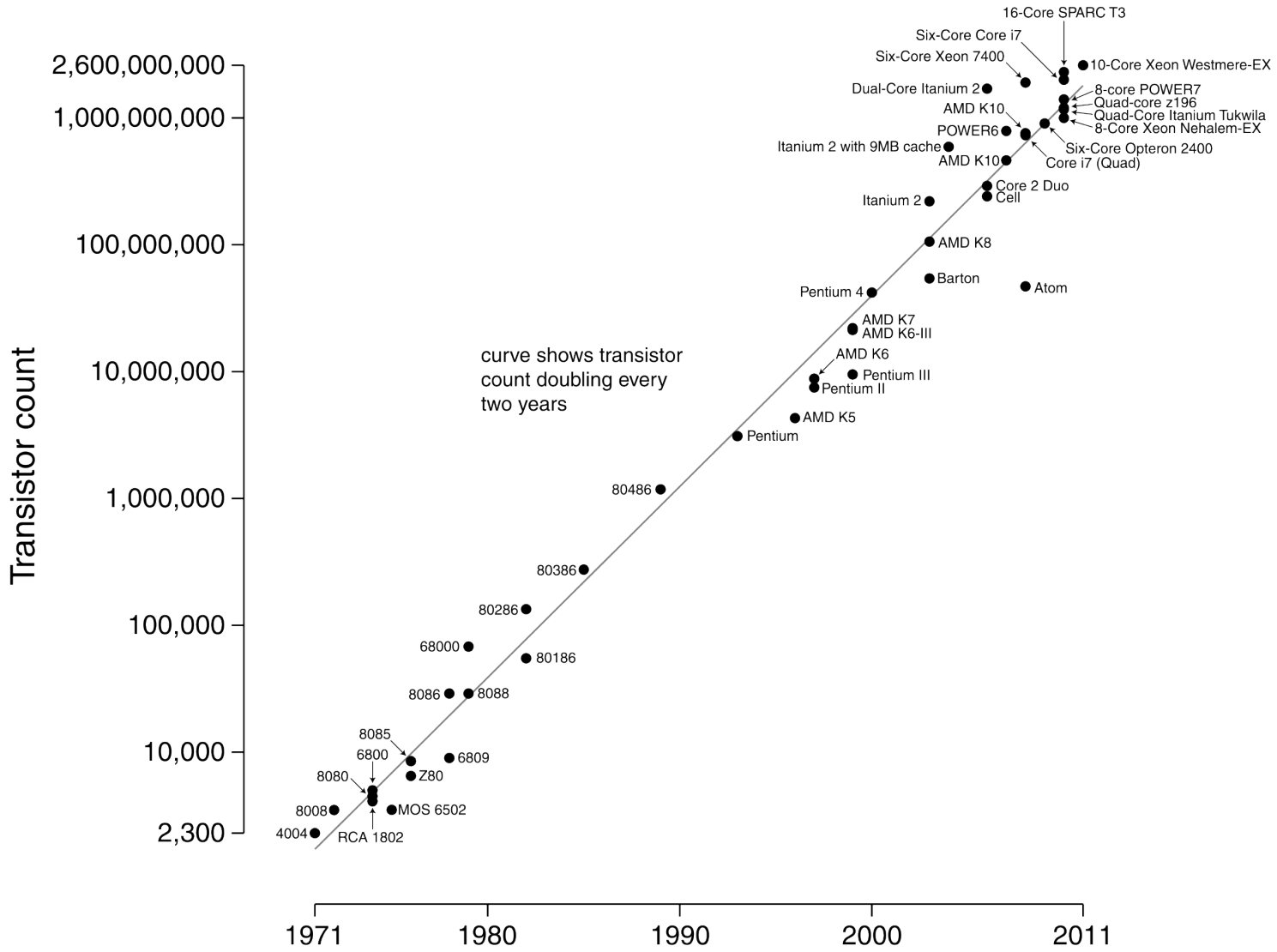
Some History

name	**date**	**transistors**	**notes**
8086	1978	29K	16-bit processor. DOS. 1MB address space. DOS allows 640K
80286	1982	134K	Windows
80386	1985	275K	32-bit registers. Flat addressing. Can run a Unix OS
80486	1989	1.9M	
Pentium	1993	3.1M	
'/MMX	1997	4.5M	instructions helpful for multimedia processing
PentiumPro	1995	6.5M	conditional move instructions
Pentium III	1999	8.2M	
Pentium IV	2001	42M	
Core 2 Duo	2006	291M	
i7	2008	731M	

In parallel. IA64 chips.

name	**date**	**transistors**
Itanium	2001	10M
Itanium 2	2002	221M
Itanium 2 Dual-Core	2006	1.7B

Microprocessor Transistor Counts 1971-2011 & Moore's Law



from [wikipedia]

aside: goals then and now

- big goal then:
  – cram as much processing power on a chip possible
- big goal now:
  – cram as much processing power on a chip possible

  BUT
  - don't use so much power
  - some environments: keep the chip small

  think cell phone

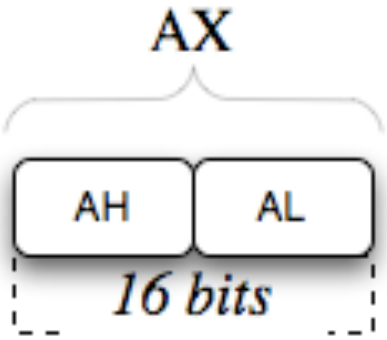
How bad is your electric bill?

Company	**Servers**	**Electricity**	**Cost**
eBay	16K	$\approx 0.6 \times 10^5$ MWh	$\approx$ \$3.7M
Akamai	40K	$\approx 1.7 \times 10^5$ MWh	$\approx$ \$10M
Rackspace	50K	$\approx 2 \times 10^5$ MWh	$\approx$ \$12M
Microsoft	>200K	$> 6 \times 10^5$ MWh	$>$ 36M
Google	>500K	$> 6.3 \times 10^5$ MWh	>38M
USA (2006)	10.9M	610×10^5 MWh	\$4.5B
MIT campus		2.7×10^5 MWh	\$62M

from, *Cutting the Electric Bill for Internet-Scale Systems,* Qureshi et al, CCR 2009.

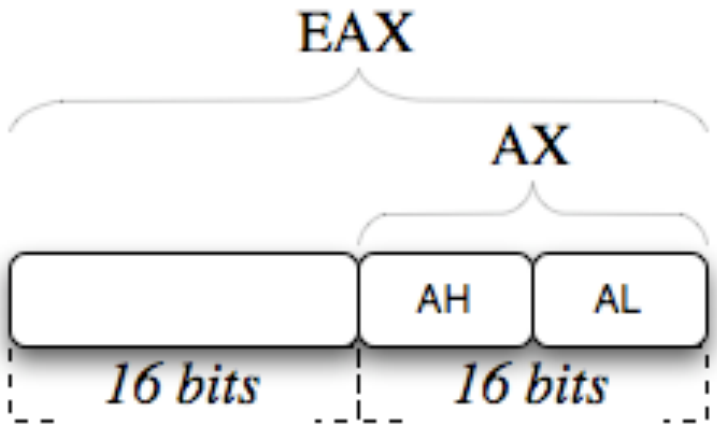
8086 Register

- Example general purpose register

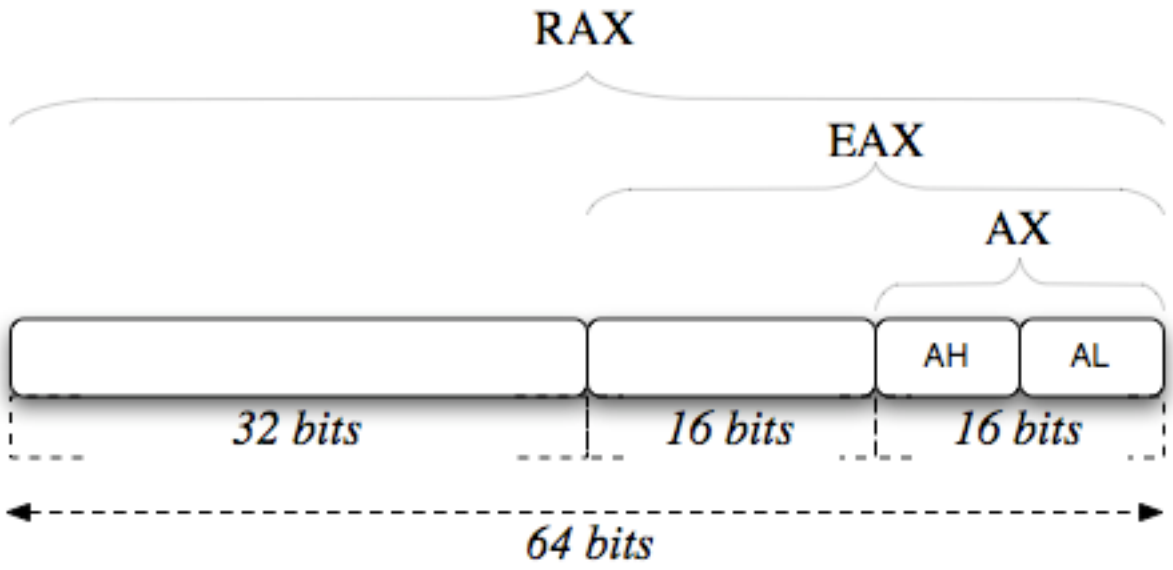


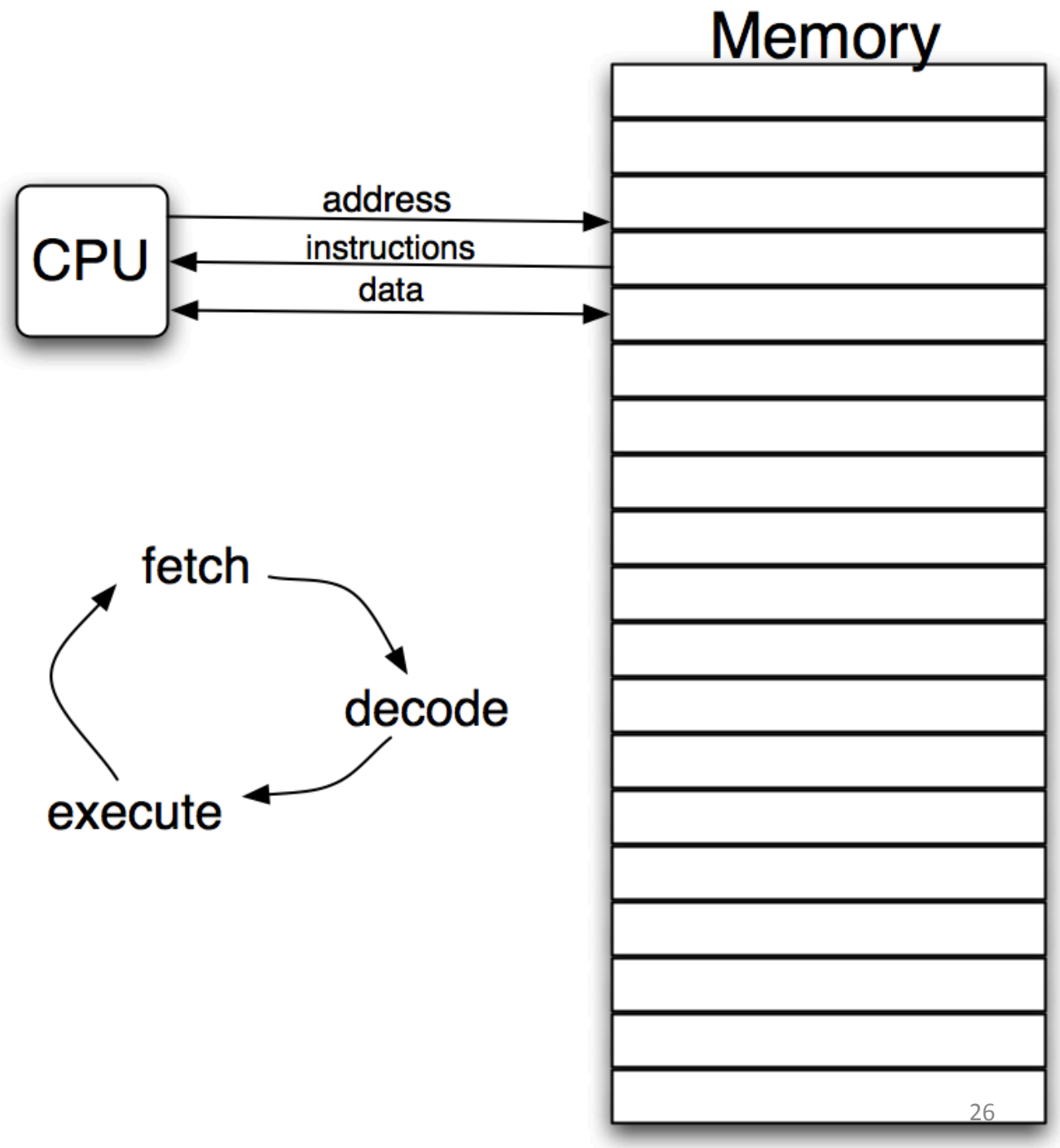
- 8 general purpose

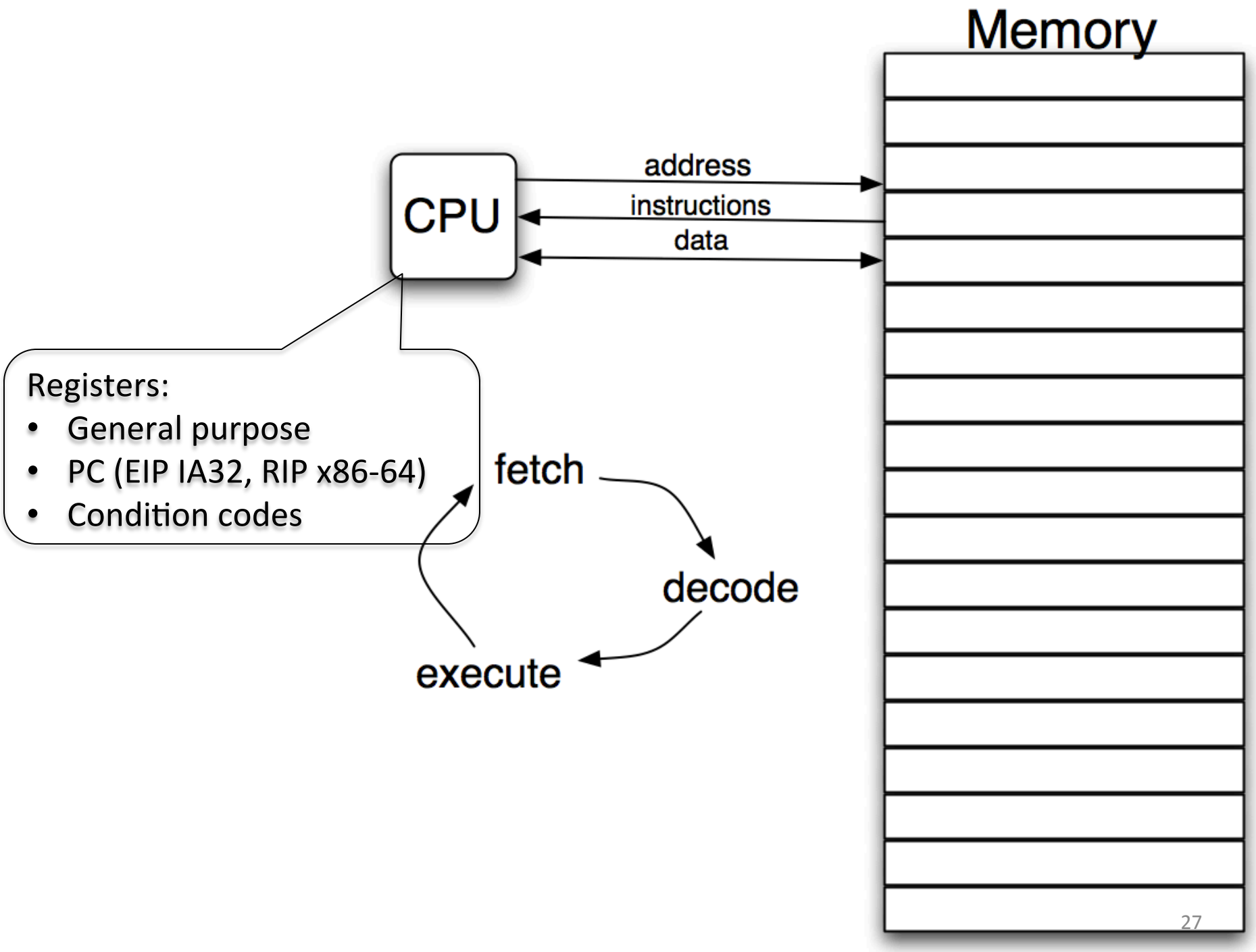
386 registers



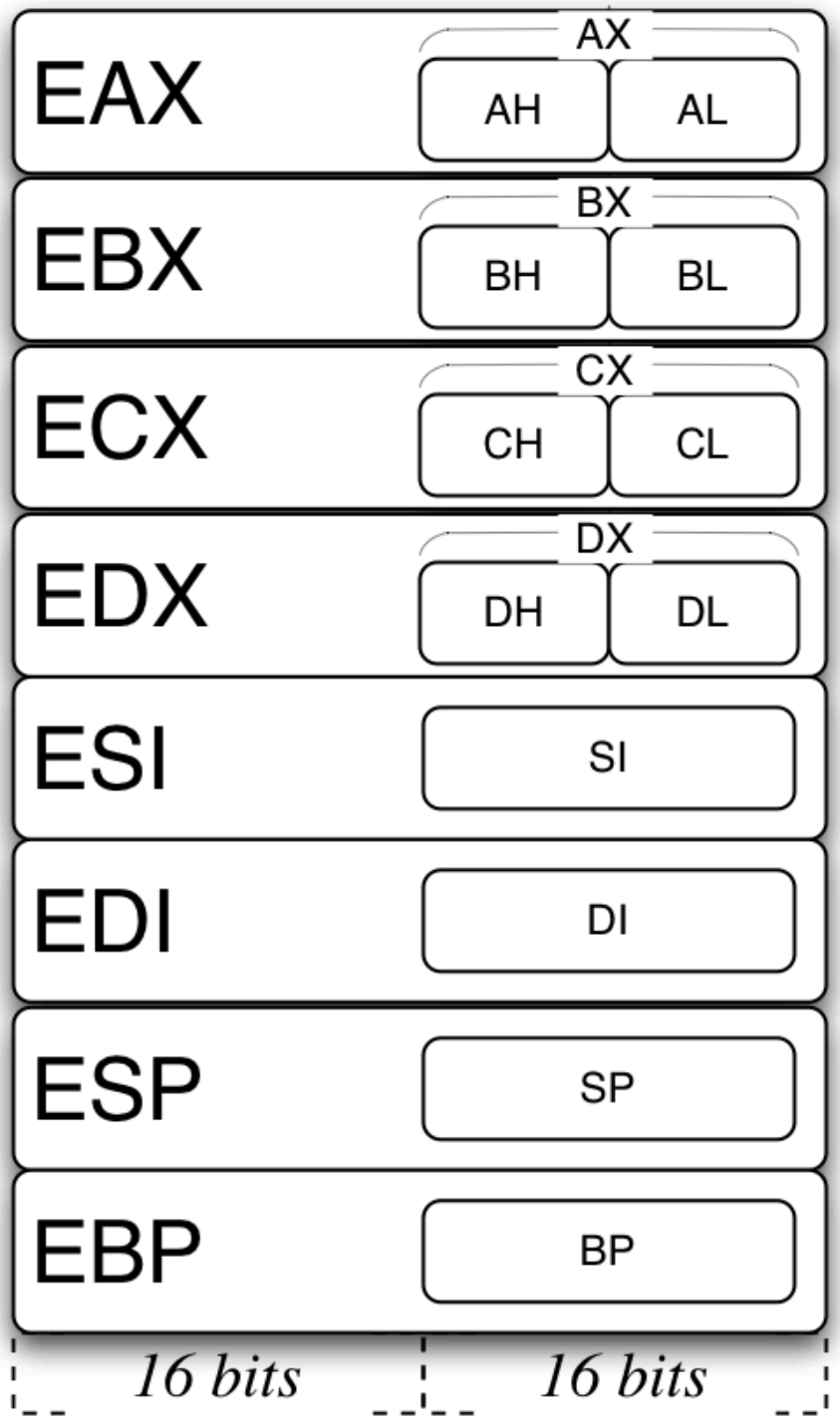
64-bit general purpose registers




Memory
CPU
address
instructions
data
fetch
decode
execute


Memory
CPU
address
instructions
data
Registers:
• General purpose
• PC (EIP IA32, RIP x86-64)
• Condition codes
fetch
decode
execute

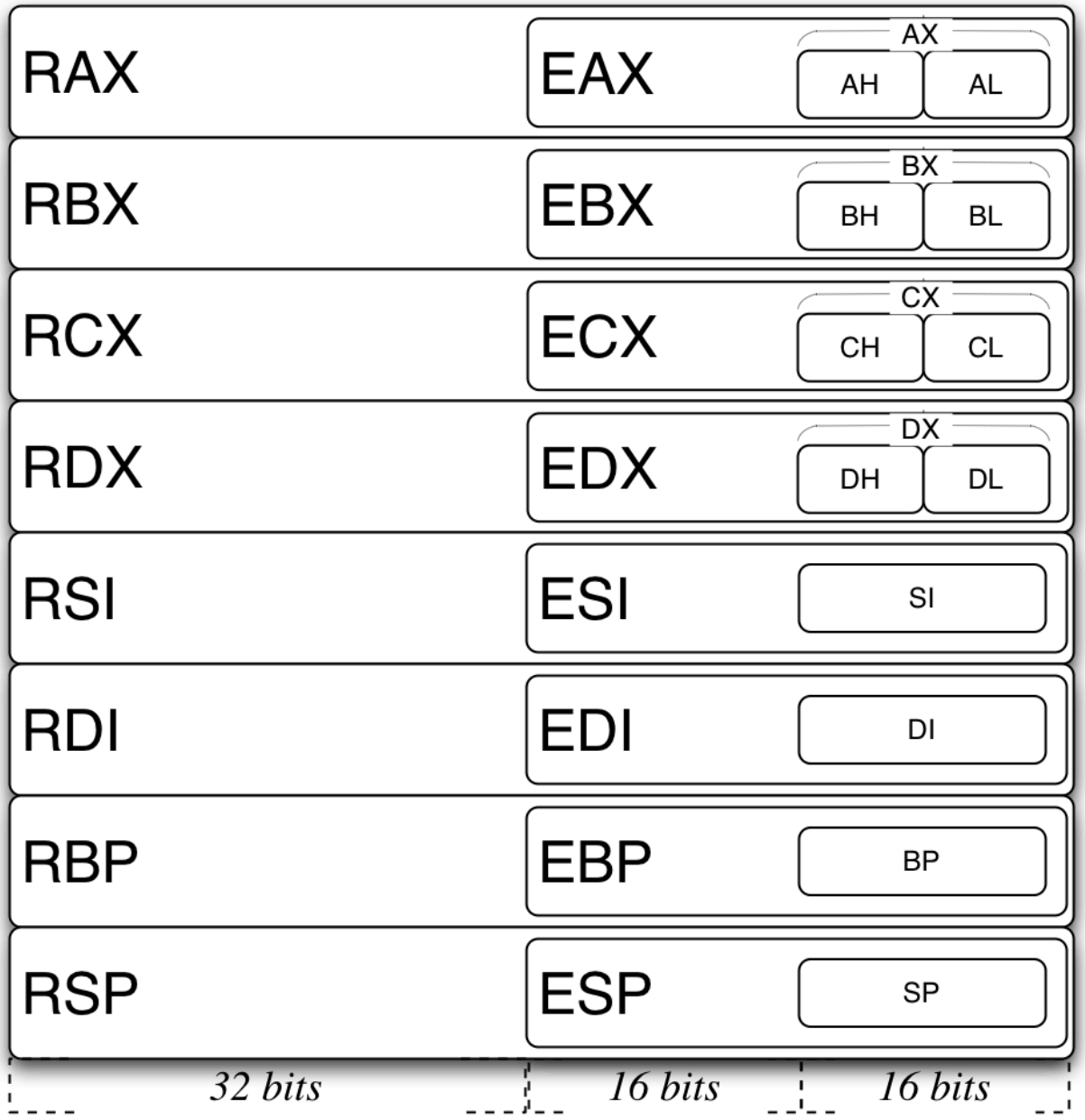
32-bit "Register File"



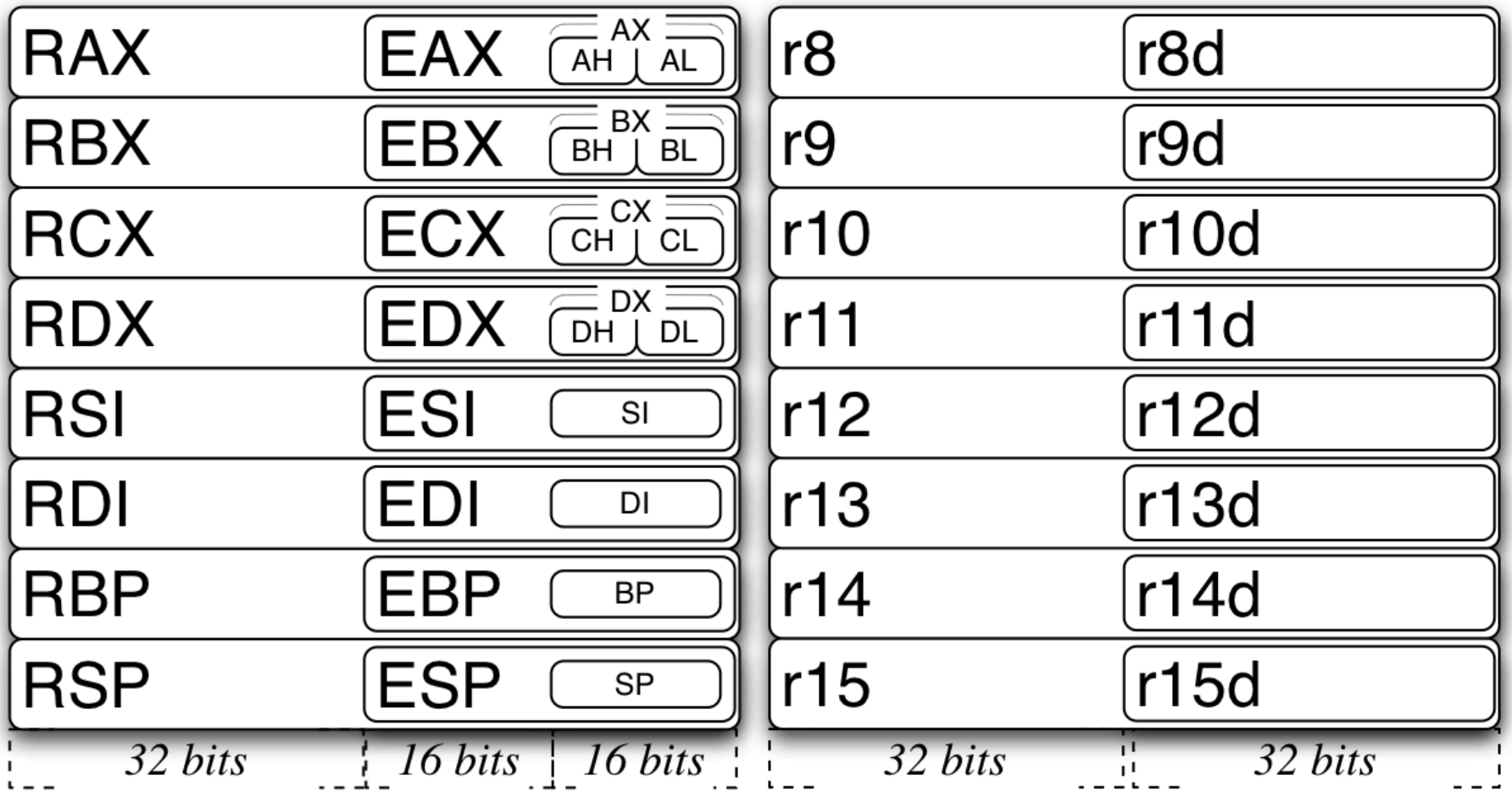
“general purpose” registers

register	**common use**
EAX	accumulator, return
EBX	pointer to items in data segment
ECX	loop control
EDX	I/O pointer
ESI	src ptr for string ops
EDI	dst ptr for string ops
ESP	stack pointer
EBP	base pointer

64-bit Register File



but wait, there's more in x86-64



data types in Intel-speak

name	size
byte	1 byte
word	2 bytes
doubleword	4 bytes
quadword	8 bytes
double quadword	16 bytes

- so a word isn't a word?

simplest function ever

```
1  int sum()
2  {
3     int x=30;
4     int y=57;
5     int z=39;
6
7     return x+y+z;
8  }
```

gcc output of sum function

```
1           .text
2   .globl _sum
3   _sum:
4           pushl       %ebp                    #,
5           movl        %esp, %ebp              #,
6           subl        $24, %esp               #,
7           movl        $30, -20(%ebp)          #, x
8           movl        $57, -16(%ebp)          #, y
9           movl        $39, -12(%ebp)          #, z
10          movl        -16(%ebp), %eax         # y, y
11          addl        -20(%ebp), %eax         # x, D.1509
12          addl        -12(%ebp), %eax         # z, D.1508
13          leave
14          ret
```

simplest function ever

```
1  int sum()
2  {
3    int x=30;
4    int y=57;
5    int z=39;
6
7    return x+y+z;
8  }
```

```
1           .text
2  .globl _sum
3  _sum:
4           pushl    %ebp
5           movl     %esp, %ebp
6           subl     $24, %esp
7           movl     $30, -20(%ebp)
8           movl     $57, -16(%ebp)
9           movl     $39, -12(%ebp)
10          movl     -16(%ebp), %eax
11          addl     -20(%ebp), %eax
12          addl     -12(%ebp), %eax
13          leave
14          ret
```

let’s try a full program

```
1  /* file summain.c */
2  int main(void)
3  {
4    int x=30;
5    int y=57;
6    int z=39;
7
8    return x+y+z;
9  }
```

its assembly output

```
1  /* file summain.s */
2                  .text
3                  .globl _main
4  _main:
5                  pushl %ebp
6                  movl  %esp, %ebp
7                  subl  $24, %esp
8                  movl  $30, -20(%ebp)
9                  movl  $57, -16(%ebp)
10                 movl  $39, -12(%ebp)
11                 movl  -16(%ebp), %eax
12                 addl  -20(%ebp), %eax
13                 addl  -12(%ebp), %eax
14                 leave
15                 ret
16                 .subsections_via_symbols
```

same code on SPARC

```
1   .section        ".text"
2           .align 4
3           .global main
4           .type    main,#function
5           .proc   04
6   main:
7           !#PROLOGUE# 0
8           save    %sp, -128, %sp
9           !#PROLOGUE# 1
10          mov     30, %o0
11          st      %o0, [%fp-20]
12          mov     57, %o0
13          st      %o0, [%fp-24]
14          mov     39, %o0
15          st      %o0, [%fp-28]
16          ld      [%fp-20], %o0
17          ld      [%fp-24], %o1
18          add     %o0, %o1, %o0
19          ld      [%fp-28], %o1
20          add     %o0, %o1, %o0
21          mov     %o0, %i0
22          b       .LL2
23           nop
24  .LL2:
25          ret
26          restore
27  .LLfe1:
28          .size    main,.LLfe1-main
29          .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

most of the same on ARM

```
1   sum:
2           @ args = 0, pretend = 0, frame = 16
3           @ frame_needed = 1, uses_anonymous_args = 0
4           @ link register save eliminated.
5           push        {r7}
6           sub         sp, sp, #20
7           add         r7, sp, #0
8           mov         r3, #30
9           str         r3, [r7, #12]
10          mov         r3, #57
11          str         r3, [r7, #8]
12          mov         r3, #39
13          str         r3, [r7, #4]
14          ldr         r2, [r7, #12]
15          ldr         r3, [r7, #8]
16          adds        r2, r2, r3
17          ldr         r3, [r7, #4]
18          adds        r3, r2, r3
19          mov         r0, r3
20          add         r7, r7, #20
21          mov         sp, r7
22          pop         {r7}
23          bx          lr
```

sum.c

```
int sum(int a, int b, int c)
{
  return a+b+c;
}
```

compiler

sum.s

```
    .text
.globl _sum
_sum:
    pushl %ebp
    movl  %esp, %ebp
    subl  $8, %esp
    movl  12(%ebp), %eax
    addl  8(%ebp), %eax
    addl  16(%ebp), %eax
    leave
    ret
```

assembler

sum.o

```
01010101100010011110010110000011111011000000100010001011010001010000110000000011010001010000100000000011010001010001000011001001110000110000000000000000000000000000000000000100000000000000000000000000000000000011110000000010000000000000000000000000000000000000000000000000000000000000000000000000000000010111110111001101110101011011010000000000000000000000000
```

libc

printf
scanf
fopen
...
strcpy
...

linker

executable file:

sum

```
01010101100010011110010110000011111011000000100010001011010001010000110000000011010001010000100000000011010001010001000011001001110000110000000000000000000000000000000000000100000000000000000000000000000000000011110000000010000000000000000000000000000000000000000000000000000000000000000000000000000000010111110111001101110101011011010000000000000000000000000
```

sum.c

```
int sum(int a, int b, int c)
{
  return a+b+c;
}
```

compiler

sum.s

```
    .text
.globl _sum
_sum:
    pushl %ebp
    movl  %esp, %ebp
    subl  $8, %esp
    movl  12(%ebp), %eax
    addl  8(%ebp), %eax
    addl  16(%ebp), %eax
    leave
    ret
```

understand what's in each

assembler

sum.o

```
01010101100010011110010110000011111011000000100010001011010001010000110000000011010001010000100000000011010001010001000011001001110000110000000000000000000000000000000000000100000000000000000000000000000000000011110000000010000000000000000000000000000000000000000000000000000000000000000000000000000010111110111001101110101011011010000000000000000000000000
```

libc

printf
scanf
fopen
...
strcpy
...

linker

executable file:

sum

```
01010101100010011110010110000011111011000000100010001011010001010000110000000011010001010000100000000011010001010001000011001001110000110000000000000000000000000000000000000100000000000000000000000000000000000011110000000010000000000000000000000000000000000000000000000000000000000000000000000000000010111110111001101110101011011010000000000000000000000000
```

sum.c

```
int sum(int a, int b, int c)
{
  return a+b+c;
}
```

decompiler?

opposite direction?

sum.s

```
   .text
.globl _sum
_sum:
   pushl %ebp
   movl  %esp, %ebp
   subl  $8, %esp
   movl  12(%ebp), %eax
   addl  8(%ebp), %eax
   addl  16(%ebp), %eax
   leave
   ret
```

disassembler

sum.o

```
01010101100010011110010110000011111011000000100010001011010001010000110000000011010001010001000001101000101000100001100100111000011000000000000000000000000000000000100000000000000000000000000000000111100000001000000000000000000000000000000000000000000000000000000000000000000000101111110111001101110101011011010000000000000000000000000
```

what can we disassemble?

- any executable
- disassembler interprets bytes as assembly src
- no source code required

binary compatibility

- Linux runs on my Intel desktop
- Windows runs on my Intel desktop
- Why can't I take my Windows binaries and run them on Linux and vice versa?