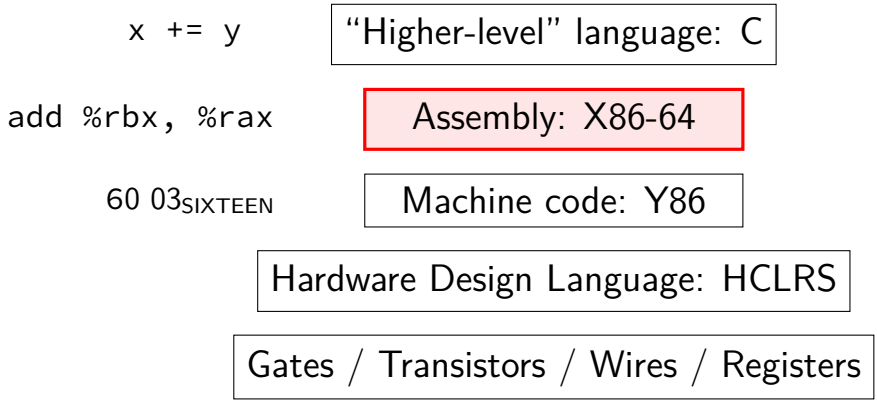
layers of abstraction



last time

model: processor $\leftrightarrow$ memory via bus

- processor can compute + has 'registers'
- processor sends messages to read/write from memory
- both machine code and data
- I/O also connected ("bridge" determines whether I/O or memory)
- (model is simplified; example: caches mean not all reads/writes go to memory)

little endianness: least significant part has lowest address

AT&T syntax:

- destination last
- \$ before constants, % before registers
- displacement(base, index, scale) =
  - memory at displacement+ base + index $\times$ scale

quiz demo

AT&T versus Intel syntax by example

```
movq $42, (%rbx)
                mov QWORD PTR [rbx], 42
subq %rax, %r8
                sub r8, rax
movq $42, 100(%rbx,%rcx,4)
                mov QWORD PTR [rbx+rcx*4+100], 42
jmp *%rax
                 jmp rax
jmp *1000(%rax,%rbx,8)
                jmp QWORD PTR [RAX+RBX*8+1000]
```

AT&T versus Intel syntax (1)

AT&T syntax:

```
movq $42, (%rbx)
```

Intel syntax:

```
mov QWORD PTR [rbx], 42
```

effect (pseudo-C):

```
memory[rbx] <- 42
```

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q ('quad') indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T syntax: addressing

`100(%rbx)`: `memory[rbx + 100]`

`100(%rbx,8)`: `memory[rbx * 8 + 100]`

`100(,%rbx,8)`: `memory[rbx * 8 + 100]`

`100(%rcx,%rbx,8)`:

```
memory[rcx + rbx * 8 + 100]
```

`100`:

```
memory[100]
```

`100(%rbx,%rcx)`:

```
memory[rbx+rcx+100]
```

AT&T versus Intel syntax (3)

`r8` ← `r8 - rax`

AT&T syntax: **`subq`** `%rax, %r8`
Intel syntax: **`sub`** `r8, rax`

same for **`cmp`**

- after **`cmpq`** `%rax, %r8`,
  `jg` jumps if `%r8` is greater

AT&T syntax: addresses

```
addq 0x1000, %rax
// Intel syntax: add rax, QWORD PTR [0x1000]
// rax ← rax + memory[0x1000]
addq $0x1000, %rax
// Intel syntax: add rax, 0x1000
// rax ← rax + 0x1000
```

no $ — probably memory address

AT&T syntax in one slide

destination last

`()` means value in memory

`disp(base, index, scale)` same as
`memory[disp + base + index * scale]`

- omit disp (defaults to `0`)
- and/or omit base (defaults to `0`)
- and/or scale (defualts to `1`)

`$` means constant

plain number/label means value in memory

extra detail: computed jumps

```
jmpq *%rax
// Intel syntax: jmp RAX
    // goto RAX
jmpq *1000(%rax,%rbx,8)
// Intel syntax: jmp QWORD PTR[RAX+RBX*8+1000]
    // read address from memory at RAX + RBX * 8 + 1
    // go to that address
```

AT&T versus Intel syntax by example

```
movq $42, (%rbx)
                mov QWORD PTR [rbx], 42
subq %rax, %r8
                sub r8, rax
movq $42, 100(%rbx,%rcx,4)
                mov QWORD PTR [rbx+rcx*4+100], 42
jmp *%rax
                 jmp rax
jmp *1000(%rax,%rbx,8)
                jmp QWORD PTR [RAX+RBX*8+1000]
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

swap (Intel syntax)

```
swap:
  mov RAX, QWORD PTR [RDI]
  mov RDX, QWORD PTR [RSI]
  mov QWORD PTR [RDI], RDX
  mov QWORD PTR [RSI], RAX
  ret
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

as pseudocode

```
swap:
    RAX ← memory[RDI (arg 1)]
    RDX ← memory[RSI (arg 2)]
    memory[RDI (arg 1)] ← RDX
    memory[RSI (arg 2)] ← RAX
    return
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

registers

%rax	???
%rdx	???
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFFF8
…	…

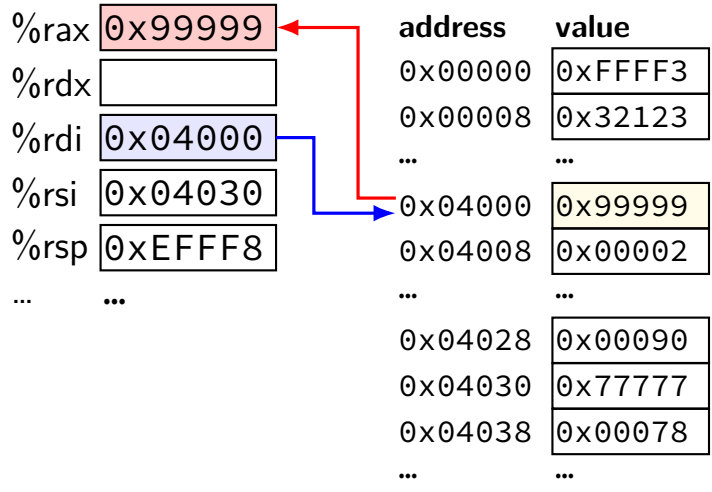
memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
…	…
0x04000	0x99999
0x04008	0x00002
…	…
0x04028	0x00090
0x04030	0x77777
0x04038	0x00078
…	…

swap

swap (AT&T syntax)

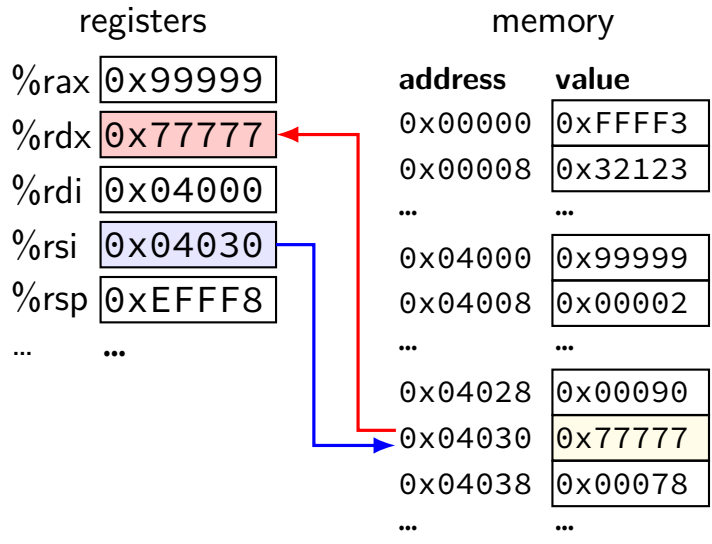
```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

registers

%rax	0x99999
%rdx	0x77777
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFFF8
…	…

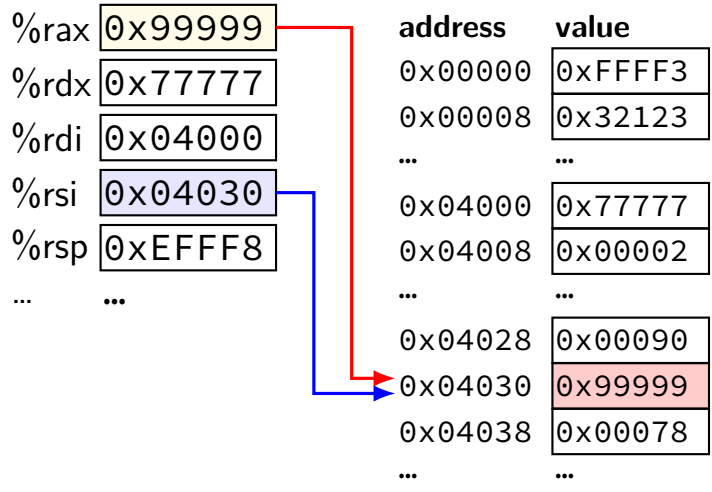
memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
…	…
0x04000	0x77777
0x04008	0x00002
…	…
0x04028	0x00090
0x04030	0x77777
0x04038	0x00078
…	…

swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```



swap

swap (AT&T syntax)

```
// swap(long *rdi,
//      long *rsi)
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

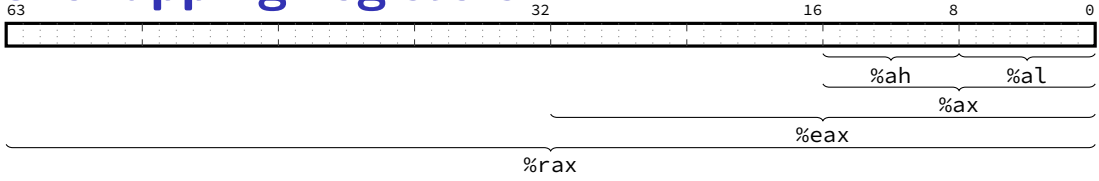
registers

%rax	0x99999
%rdx	0x77777
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFFF8
…	…

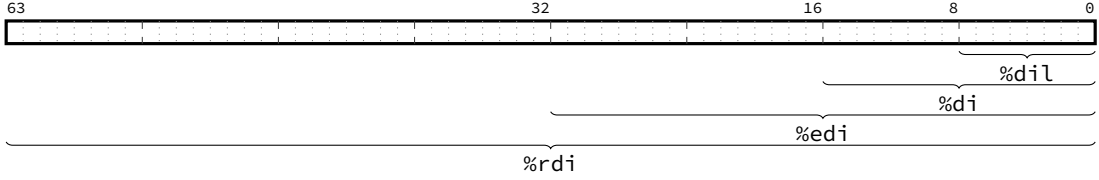
memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
…	…
0x04000	0x77777
0x04008	0x00002
…	…
0x04028	0x00090
0x04030	0x99999
0x04038	0x00078
…	…

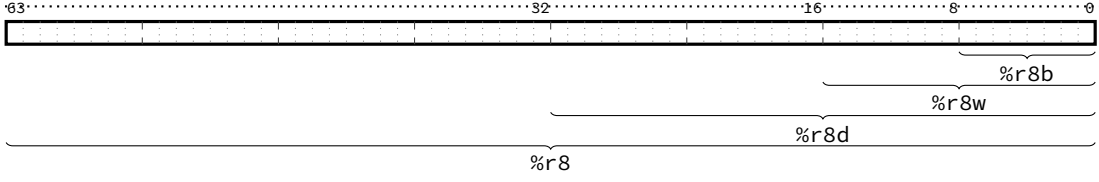
overlapping registers



+ similar for `%rcx`, `%rbx`, `%rdx`



+ similar for `%rbp`, `%rsi`, `%rsp`



+ similar for `%r9` through `%r15`

overlapping registers (1)

setting 32-bit registers — clears corresponding 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax
movl $0x1, %eax
```

%rax is `0x1` (not `0xFFFFFFFF00000001`)

overlapping registers (2)

setting 8/16-bit registers: don't clear 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax
movb $0x1, %al
```

%rax is `0xFFFFFFFFFFFFFF01`

labels (1)

labels represent addresses

labels (2)

```
    addq string, %rax
    // intel syntax: add rax, QWORD PTR [label]
    // rax ← rax + memory[address of "a string"]
    addq $string, %rax
    // intel syntax: add rax, OFFSET label
    // rax ← rax + address of "a string"
string:  .ascii "a string"
```

`addq label`: read value at the address

`addq $label`: use address as an integer constant

on %rip

%rip (**I**nstruction **P**ointer) = address of next instruction

```
movq 500(%rip), %rax
```

rax ← memory[next instruction address + 500]

on %rip

%rip (**I**nstruction **P**ointer) = address of next instruction

`movq 500(%rip), %rax`

rax ← memory[next instruction address + 500]

very special case: `label(%rip)` ≈ `label`

different ways of writing address of label in machine code
(with %rip — relative to next instruction)

exericse

```
hello:
    .string "Hello, World!" ; nul-terminated string
example:
    movb hello+1, %bl
    subb $1, %bl
    movb %bl, hello
    movq $hello, %rdi
    ; int puts(const char *s [%rdi])
    callq puts
    ret
```

What is the the argument to puts, %rdi?

A. a pointer to 'Hello, World!' B. a pointer to 'dello, World!'
C. a pointer to 'Hdllo, World!' D. a pointer to 'fello, World!'
E. a pointer to 'Jello, World!' F. a pointer to a different string
G. an integer constructed from the ASCII for 'Hello, W' (puts probably crashes)
H. an integer constructed from the ASCII for 'Jello, W' (puts probably crashes)
I. an integer constructed from the ASCII for a different string (puts probably crashes)

exercise (explanation)

```
hello:
  .string "Hello, World!" ; nul-terminated string
example:
  movb hello+1, %bl
    // hello = address of 'H' in string, hello+1 = addr of 'e', ...
    // %bl becomes 'e'
  subb $1, %bl
    // %bl becomes 'd'
  movb %bl, hello
    // move 'd' to where 'H' is stored; string now "dello, World!"
  movq $hello, %rdi
    // move address of (first char in) the string "dello, World"
  ; int puts(const char *s [%rdi])
  callq puts
  ret
```

exercise (explanation)

```
hello:
  .string "Hello, World!" ; nul-terminated string
example:
  movb hello+1, %bl
    // hello = address of 'H' in string, hello+1 = addr of 'e', ...
    // %bl becomes 'e'
  subb $1, %bl
    // %bl becomes 'd'
  movb %bl, hello
    // move 'd' to where 'H' is stored; string now "dello, World!"
  movq $hello, %rdi
    // move address of (first char in) the string "dello, World"
  ; int puts(const char *s [%rdi])
  callq puts
  ret
```

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address

(sort of like & operator in C?)

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address

(sort of like & operator in C?)

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

"address of memory[rax + 4]" = rax + 4

LEA tricks

```
leaq (%rax,%rax,4), %rax
```

rax $\leftarrow$ rax $\times$ 5

rax $\leftarrow$ `address-of(memory[rax + rax * 4])`

```
leaq (%rbx,%rcx), %rdx
```

rdx $\leftarrow$ rbx $+$ rcx

rdx $\leftarrow$ `address-of(memory[rbx + rcx])`

exercise: what is this function?

```
mystery:
    leal 0(,%rdi,8), %eax
    subl %edi, %eax
    ret
```

```
int mystery(int arg) { return ...; }
```

A. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-arg * 7`
E. none of these
F. it has a different prototype

exercise: what is this function?

```
mystery:
    leal 0(,%rdi,8), %eax
    subl %edi, %eax
    ret
```

```
int mystery(int arg) { return ...; }
```

A. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-arg * 7`
E. none of these
F. it has a different prototype

backup slides

authoriative source (1)

Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

authoriative source (2)

System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.99.7

Edited by
Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitche

November 17, 2014

question

```
pushq $0x1
pushq $0x2
addq $0x3, 8(%rsp)
popq %rax
popq %rbx
```

What is value of `%rax` and `%rbx` after this?

a. `%rax` = `0x2`, `%rbx` = `0x4`
b. `%rax` = `0x5`, `%rbx` = `0x1`
c. `%rax` = `0x2`, `%rbx` = `0x1`
d. the snippet has invalid syntax or will crash
e. more information is needed
f. something else?