

Changelog

changes since first lecture:

25 Aug 2022: adjust AT&T syntax exercise to ask why it does not assemble

25 Aug 2022: move labels slides earlier

last time

things besides building processors

future topics: parallelism + caches

processor \longleftrightarrow memory

memory just stores bytes

processor asks for values or to set values

memory stores both instructions *and* many values instruction uses

processor stores register file, program counter

endianness: combining multiple pieces into one value

matters when we can access the pieces separately

typical piece: bytes of memory

little endian: lowest address becomes least significant part (1's place)

assembly: adding before and/or after accessing memory

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)

$r8 \leftarrow r8 - rax$

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` (defaults 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 + 1000
```

```
// 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]`

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

exercise

recall:

destination **last**

() means value **in memory**

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

(disp, base, scale) defaults to (0, 0, 1)

\$ means constant

plain number/label means value **in memory**

what is `addq foo, (%rax, %rcx, 8)` most likely trying to do? [Bonus Q: why doesn't it assemble?]

- A. add a value from an array at address `%rax` into a variable `foo`
- B. add a variable `foo` to an element of an array at `%rax`
- C. add the constant `foo` to an element of an array at `%rax`
- D. make `foo` point to the `%rcx`'th element of an array starting at `%rax`
- E. something else

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)

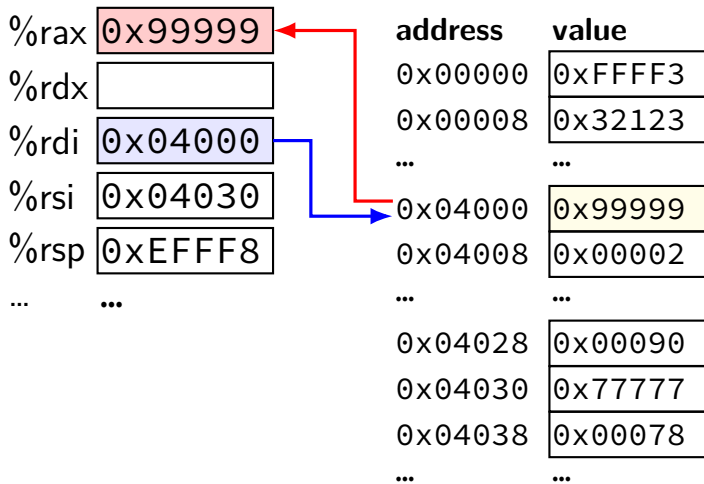
```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

registers

%rax	0x99999
%rdx	
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFF8
...	...

memory

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



swap

swap (AT&T syntax)

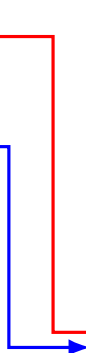
```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

registers

%rax	0x99999
%rdx	0x77777
%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
```

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)

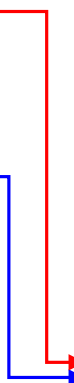
```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

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
...	...



swap

swap (AT&T syntax)

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

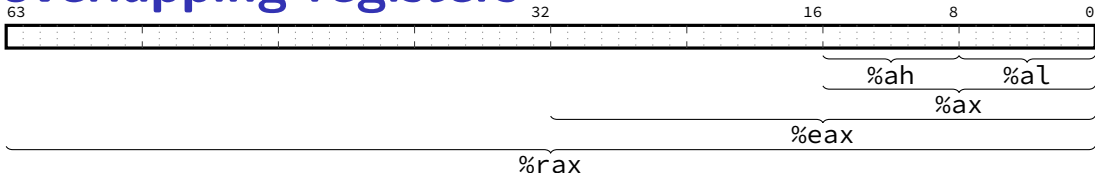
registers

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

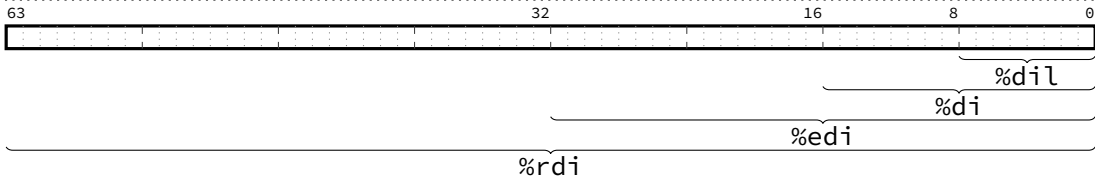
memory

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

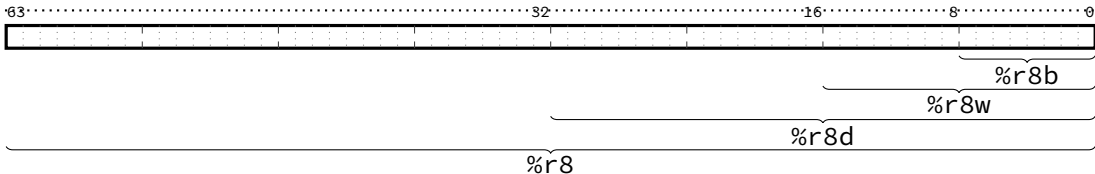
overlapping registers



+ similar for %rcx, %rbx, %rdx

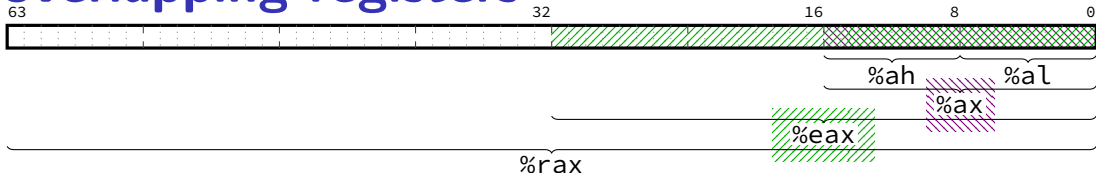


+ similar for %rbp, %rsi, %rsp

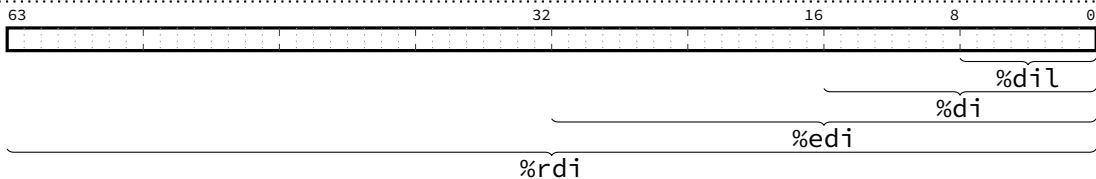


+ similar for %r9 through %r15

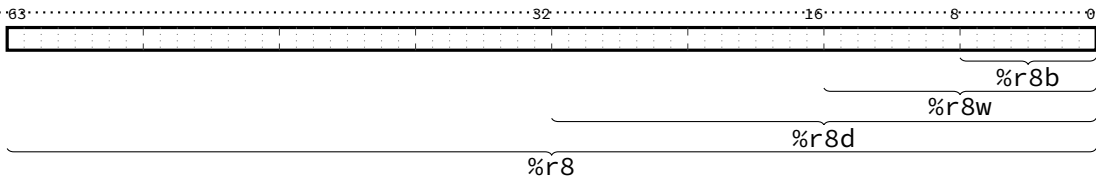
overlapping registers



+ 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)

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

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

%rax is 0xFFFFFFFFFFFFFFFF01

(why? my guess: clearing whole register is faster with CPU designs that were common when 64-bit registers were added)

Linux x86-64 calling convention

registers for first 6 arguments:

`%rdi` (or `%edi` or `%di`, etc.), then

`%rsi` (or `%esi` or `%si`, etc.), then

`%rdx` (or `%edx` or `%dx`, etc.), then

`%rcx` (or `%ecx` or `%cx`, etc.), then

`%r8` (or `%r8d` or `%r8w`, etc.), then

`%r9` (or `%r9d` or `%r9w`, etc.)

rest on stack (pushed before return address)

return value in `%rax`

don't memorize: Figure 3.28 in book

x86-64 calling convention example

```
int foo(int x, int y, int z) { return 42; }
```

```
...
```

```
    foo(1, 2, 3);
```

```
...
```

```
...
```

```
    // foo(1, 2, 3)
```

```
    movl $1, %edi
```

```
    movl $2, %esi
```

```
    movl $3, %edx
```

```
    call foo // call pushes address of next instruction  
            // then jumps to foo
```

```
...
```

```
foo:
```

```
    movl $42, %eax
```

```
    ret
```

call/ret

call:

push address just past the end of the call instruction on the stack
“return address” – where we expect to go when function finishes

ret:

pop address from stack; jump

callee-saved registers

functions **must preserve** these

`%rsp` (stack pointer), `%rbx`, `%rbp` (frame pointer, maybe)

`%r12–%r15`

caller/callee-saved

foo:

```
    pushq %r12 // r12 is callee-saved
    ... use r12 ...
    popq %r12
    ret
```

...

other_function:

```
    ...
    pushq %r11 // r11 is caller-saved
    callq foo
    popq %r11
```

Hello, World!

hello:

```
.string "Hello, World!" ; nul-terminated string
```

example:

```
movq $hello, %rdi  
; int puts(const char *s [%rdi])  
callq puts  
ret
```

Hello, World!

hello:

```
.string "Hello, World!" ; nul-terminated string
```

example:

```
movq $hello, %rdi  
; int puts(const char *s [%rdi])  
callq puts  
ret
```

`.string`: put string in memory (instead of machine code)

`$hello`: address that hello points to

hello: bytes at that address (interpreted as an integer)

`puts`: takes pointer to (= address of) string + outputs it

Linux calling convention: `%rdi` = first argument

exercice

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)

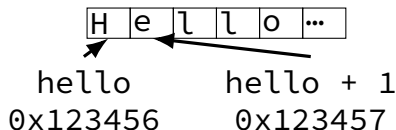
exercise (explanation)

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
```



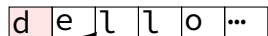
exercise (explanation)

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
```



hello hello + 1
0x123456 0x123457

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` \approx `addq $4, %rax`

“address of memory[`rax + 4`]” = `rax + 4`

LEA tricks

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

$\text{rax} \leftarrow \text{rax} \times 5$

```
rax ← address-of(memory[rax + rax * 4])
```

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

$\text{rdx} \leftarrow \text{rbx} + \text{rcx}$

```
rdx ← address-of(memory[rbx + rcx])
```