

Assembly part 2 / C part 1

1

last time

linking extras:

different kinds of relocations — addresses versus offset to addresses
dynamic linking (briefly)

AT&T syntax

destination last

$O(B, I, S) \text{ — } B + I \times S + O$

jmp *

lea — mov, but don't do memory access

if/do-while to assembly

condition codes — last arithmetic result

2

reminder: quiz

post-quiz — after this lecture

pre-quiz — before next lecture

demo

3

condition codes

x86 has **condition codes**

set by (almost) all arithmetic instructions
addq, subq, imulq, etc.

store info about **last arithmetic result**
was it zero? was it negative? etc.

4

condition codes and jumps

`jg`, `jle`, etc. read condition codes

named based on interpreting **result of subtraction**

0: equal; negative: less than; positive: greater than

5

condition codes example (1)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 30
// result > 0: %rbx was > %rax
jle foo // not taken; 30 > 0
```

6

condition codes and `cmpq`

“**last arithmetic result**”???

then what is `cmp`, etc.?

`cmp` does **subtraction** (but doesn't store result)

similar `test` does bitwise-and

`testq %rax, %rax` — result is `%rax`

7

condition codes example (2)

```
movq $-10, %rax
movq $20, %rbx
cmpq %rax, %rbx
jle foo // not taken; %rbx - %rax > 0
```

8

omitting the cmp

```
movq $99, %r12 // register for x
start_loop:
call foo
subq $1, %r12
cmpq $0, %r12
// compute r12 - 0 + sets cond. codes
jge start_loop // r12 >= 0?
// or result >= 0?
```

```
movq $99, %r12 // register for x
start_loop:
call foo
subq $1, %r12
// new r12 = old r12 - 1 + sets cond. codes
jge start_loop // old r12 >= 1?
// or result >= 0?
```

9

condition codes example (3)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx
jle foo // not taken, %rbx - %rax > 0 -> %rbx
```

```
movq $20, %rbx
addq $-20, %rbx
je foo // taken, result is 0
// x - y = 0 -> x = y
```

10

what sets condition codes

most instructions that compute something **set condition codes**

some instructions **only** set condition codes:

```
cmp ~ sub
test ~ and (bitwise and — later)
testq %rax, %rax — result is %rax
```

some instructions don't change condition codes:

```
lea, mov
control flow: jmp, call, ret, jle, etc.
```

11

condition codes examples (4)

```
movq $20, %rbx
addq $-20, %rbx // result is 0
movq $1, %rax // irrelevant
je foo // taken, result is 0
```

12

while-to-assembly (1)

```
while (x >= 0) {  
    foo()  
    x--;  
}
```

13

while-to-assembly (1)

```
while (x >= 0) {  
    foo()  
    x--;  
}
```

```
start_loop:  
    if (x < 0) goto end_loop;  
    foo()  
    x--;  
    goto start_loop;  
end_loop:
```

13

while-to-assembly (2)

```
start_loop:  
    if (x < 0) goto end_loop;  
    foo()  
    x--;  
    goto start_loop;  
end_loop:
```

```
start_loop:  
    cmpq $0, %r12  
    jl end_loop // jump if r12 - 0 >= 0  
    call foo  
    subq $1, %r12  
    jmp start_loop
```

14

while exercise

```
while (b < 10) { foo(); b += 1; }
```

Assume b is in **callee-saved** register %rbx. Which are correct assembly translations?

```
// version A  
start_loop:  
    call foo  
    addq $1, %rbx  
    cmpq $10, %rbx  
    jl start_loop
```

```
// version B  
start_loop:  
    cmpq $10, %rbx  
    jge end_loop  
    call foo  
    addq $1, %rbx  
    jmp start_loop  
end_loop:
```

```
// version C  
start_loop:  
    movq $10, %rax  
    subq %rbx, %rax  
    jge end_loop  
    call foo  
    addq $1, %rbx  
    jmp start_loop  
end_loop:
```

15

while to assembly (1)

```
while (b < 10) {  
    foo();  
    b += 1;  
}
```

16

while to assembly (1)

```
while (b < 10) {  
    foo();  
    b += 1;  
}
```

```
start_loop: if (b < 10) goto end_loop;  
            foo();  
            b += 1;  
            goto start_loop;  
end_loop:
```

16

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

start_loop: cmpq \$10, %rbx jge end_loop call foo addq \$1, %rbx jmp start_loop end_loop:		
---	--	--

17

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

start_loop: cmpq \$10, %rbx jge end_loop call foo addq \$1, %rbx jmp start_loop end_loop:		cmpq \$10, %rbx jge end_loop start_loop: call foo addq \$1, %rbx cmpq \$10, %rbx jne start_loop end_loop:
---	--	--

17

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

start_loop: cmpq \$10, %rbx jge end_loop call foo addq \$1, %rbx jmp start_loop end_loop:	cmpq \$10, %rbx jge end_loop start_loop: call foo addq \$1, %rbx cmpq \$10, %rbx jne start_loop end_loop:	cmpq \$10, %rbx jge end_loop movq \$10, %rax subq %rbx, %rax movq %rax, %rbx start_loop: call foo decq %rbx jne start_loop movq \$10, %rbx end_loop:
---	---	--

17

condition codes: closer look

x86 condition codes:

ZF (“zero flag”) — was result zero? (sub/cmp: equal)
SF (“sign flag”) — was result negative? (sub/cmp: less)

(and some more, e.g. to handle overflow)

GDB: part of “eflags” register

set by cmp, test, arithmetic

18

condition codes example (2)

```
movq $-10, %rax  
movq $20, %rbx  
cmpq %rax, %rbx  
jle foo // not taken; %rbx - %rax > 0
```

$\%rbx - \%rax = 30$ — SF = 0 (not negative), ZF = 0 (not zero)

19

condition codes examples (4)

```
movq $20, %rbx  
addq $-20, %rbx // result is 0  
movq $1, %rax // irrelevant  
je foo // taken, result is 0
```

$20 + -20 = 0$ — SF = 0 (not negative), ZF = 1 (zero)

20

condition codes: closer look

x86 condition codes:

- ZF (“zero flag”) — was result zero? (sub/cmp: equal)
- SF (“sign flag”) — was result negative? (sub/cmp: less)
- CF (“carry flag”) — did computation overflow (as unsigned)?
- OF (“overflow flag”) — did computation overflow (as signed)? (and one more)

GDB: part of “eflags” register

set by cmp, test, arithmetic

21

closer look: condition codes (1)

```
movq $-10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30} 30$ (overflow!)

ZF = 0 (false) not zero rax and rbx not equal

22

closer look: condition codes (1)

```
movq $-10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30} 30$ (overflow!)

ZF = 0 (false) not zero rax and rbx not equal

22

closer look: condition codes (1)

```
movq $-10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30} 30$ (overflow!)

ZF = 0 (false) not zero rax and rbx not equal

SF = 0 (false) not negative rax <= rbx

22

closer look: condition codes (1)

```
movq $-10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30} 30$ (overflow!)

ZF = 0 (false)	not zero	rax and rbx not equal
SF = 0 (false)	not negative	rax <= rbx
OF = 0 (false)	no overflow as signed	correct for signed

22

closer look: condition codes (1)

```
movq $-10, %rax
movq $20, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30} 30$ (overflow!)

ZF = 0 (false)	not zero	rax and rbx not equal
SF = 0 (false)	not negative	rax <= rbx
OF = 0 (false)	no overflow as signed	correct for signed
CF = 1 (true)	overflow as unsigned	incorrect for unsigned

22

exercise: condition codes (2)

```
// 2^63 - 1
movq $0x7FFFFFFFFFFFFFFF, %rax
// 2^63 (unsigned); -2^63 (signed)
movq $0x8000000000000000, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax
```

ZF = ?
SF = ?
OF = ?
CF = ?

23

closer look: condition codes (2)

```
// 2**63 - 1
movq $0x7FFFFFFFFFFFFFFF, %rax
// 2**63 (unsigned); -2**63 (signed)
movq $0x8000000000000000, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - (2^{63} - 1) = \cancel{-2^{64} - 1} 1$ (overflow)

as unsigned: $2^{63} - (2^{63} - 1) = 1$

ZF = 0 (false) not zero rax and rbx not equal

24

closer look: condition codes (2)

```
// 2**63 - 1
movq $0x7FFFFFFFFFFFFFFF, %rax
// 2**63 (unsigned); -2**63 (signed)
movq $0x8000000000000000, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - (2^{63} - 1) = \cancel{-2^{64} + 1} 1$ (overflow)

as unsigned: $2^{63} - (2^{63} - 1) = 1$

ZF = 0 (false) not zero rax and rbx not equal

24

closer look: condition codes (2)

```
// 2**63 - 1
movq $0x7FFFFFFFFFFFFFFF, %rax
// 2**63 (unsigned); -2**63 (signed)
movq $0x8000000000000000, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - (2^{63} - 1) = \cancel{-2^{64} + 1} 1$ (overflow)

as unsigned: $2^{63} - (2^{63} - 1) = 1$

ZF = 0 (false) not zero rax and rbx not equal
SF = 0 (false) not negative rax <= rbx (if correct)

24

closer look: condition codes (2)

```
// 2**63 - 1
movq $0x7FFFFFFFFFFFFFFF, %rax
// 2**63 (unsigned); -2**63 (signed)
movq $0x8000000000000000, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - (2^{63} - 1) = \cancel{-2^{64} + 1} 1$ (overflow)

as unsigned: $2^{63} - (2^{63} - 1) = 1$

ZF = 0 (false) not zero rax and rbx not equal
SF = 0 (false) not negative rax <= rbx (if correct)
OF = 1 (true) overflow as signed incorrect for signed

24

closer look: condition codes (2)

```
// 2**63 - 1
movq $0x7FFFFFFFFFFFFFFF, %rax
// 2**63 (unsigned); -2**63 (signed)
movq $0x8000000000000000, %rbx
cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - (2^{63} - 1) = \cancel{-2^{64} + 1} 1$ (overflow)

as unsigned: $2^{63} - (2^{63} - 1) = 1$

ZF = 0 (false) not zero rax and rbx not equal
SF = 0 (false) not negative rax <= rbx (if correct)
OF = 1 (true) overflow as signed incorrect for signed
CF = 0 (false) no overflow as unsigned correct for unsigned

24

closer look: condition codes (3)

```
movq  $-1, %rax
addq  $-2, %rax
// result = -3
```

as signed: $-1 + (-2) = -3$

as unsigned: $(2^{64} - 1) + (2^{64} - 2) = \cancel{2^{65} - 3} 2^{64} - 3$ (overflow)

ZF = 0 (false) not zero result not zero

25

closer look: condition codes (3)

```
movq  $-1, %rax
addq  $-2, %rax
// result = -3
```

as signed: $-1 + (-2) = -3$

as unsigned: $(2^{64} - 1) + (2^{64} - 2) = \cancel{2^{65} - 3} 2^{64} - 3$ (overflow)

ZF = 0 (false)	not zero	result not zero
SF = 1 (true)	negative	result is negative
OF = 0 (false)	no overflow as signed	correct for signed
CF = 1 (true)	overflow as unsigned	incorrect for unsigned

25

C Data Types

Varies between machines(!). For **this course**:

type	size (bytes)
char	1
short	2
int	4
long	8

26

C Data Types

Varies between machines(!). For **this course**:

type	size (bytes)
char	1
short	2
int	4
long	8
float	4
double	8

26

C Data Types

Varies between machines(!). For **this course**:

type	size (bytes)
char	1
short	2
int	4
long	8
float	4
double	8
void *	8
<i>anything</i> *	8

26

truth

`bool`

27

truth

`bool`

`x == 4` is an `int`
`1` if true; `0` if false

27

false values in C

`0`

including null pointers — `0` cast to a pointer

28

short-circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() || one());
6     printf(">_%d\n", one() || zero());
7     return 0;
8 }
```

zero()

one()

> 1

one()

> 1

	OR	false	true
false	false	false	true
true	true	true	true

29

short-circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() || one());
6     printf(">_%d\n", one() || zero());
7     return 0;
8 }
```

zero()

one()

> 1

one()

> 1

	OR	false	true
false	false	false	true
true	true	true	true

29

short-circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() || one());
6     printf(">_%d\n", one() || zero());
7     return 0;
8 }
```

zero()

one()

> 1

one()

> 1

	OR	false	true
false	false	false	true
true	true	true	true

29

short-circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() || one());
6     printf(">_%d\n", one() || zero());
7     return 0;
8 }
```

zero()

one()

> 1

one()

> 1

	OR	false	true
false	false	false	true
true	true	true	true

29

short-circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() || one());
6     printf(">_%d\n", one() || zero());
7     return 0;
8 }
```

```
zero()
one()
> 1
one()
> 1
```

	OR	false	true
false	false	true	true
true	true	true	true

29

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() && one());
6     printf(">_%d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

	AND	false	true
false	false	false	false
true	false	false	true

30

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() && one());
6     printf(">_%d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

	AND	false	true
false	false	false	false
true	false	false	true

30

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() && one());
6     printf(">_%d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

	AND	false	true
false	false	false	false
true	false	false	true

30

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() && one());
6     printf(">_%d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	false	true
false	false	false
true	false	true

30

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">_%d\n", zero() && one());
6     printf(">_%d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	false	true
false	false	false
true	false	true

30

strings in C

hello (on stack/register)

0x4005C0

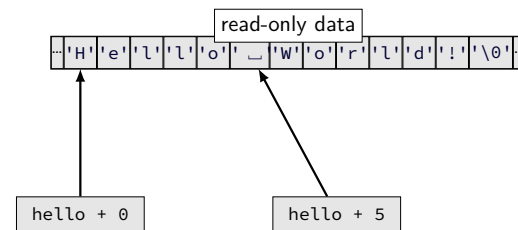
```
int main() {
    const char *hello = "Hello World!";
    ...
}
```

read-only data

... 'H' 'e' 'l' 'l' 'o' ' ' 'W' 'o' 'r' 'l' 'd' '!' '\0' ...

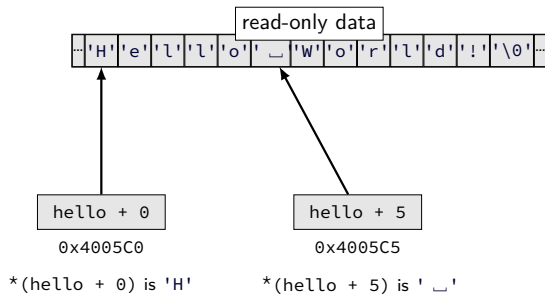
31

pointer arithmetic



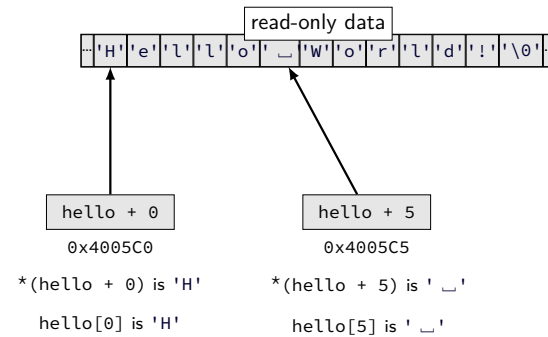
32

pointer arithmetic



32

pointer arithmetic



32

arrays and pointers

`*(foo + bar)` exactly the same as `foo[bar]`

arrays 'decay' into pointers

33

arrays of non-bytes

`array[2]` and `*(array + 2)` still the same

```
1 int numbers[4] = {10, 11, 12, 13};
2 int *pointer;
3 pointer = numbers;
4 *pointer = 20; // numbers[0] = 20;
5 pointer = pointer + 2;
6 /* adds 8 (2 ints) to address */
7 *pointer = 30; // numbers[2] = 30;
8 // numbers is 20, 11, 30, 13
```

34

arrays of non-bytes

array[2] and *(array + 2) still the same

```
1 int numbers[4] = {10, 11, 12, 13};
2 int *pointer;
3 pointer = numbers;
4 *pointer = 20; // numbers[0] = 20;
5 pointer = pointer + 2;
6 /* adds 8 (2 ints) to address */
7 *pointer = 30; // numbers[2] = 30;
8 // numbers is 20, 11, 30, 13
```

34

a note on precedence

&foo[1] is the same as &(foo[1]) (not (&foo)[1])

*foo[0] is the same as *(foo[0]) (not (*foo)[0])

*foo++ is the same as *(foo++) (not (*foo)++)

35

exercise

```
1 char foo[4] = "foo";
2 // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';
8 *(foo + 1) = 'a';
```

Final value of foo?

- A. "fao"
- B. "zao"
- C. "baz"
- D. "bao"
- E. something else/crash

36

exercise

```
1 char foo[4] = "foo";
2 // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';
8 *(foo + 1) = 'a';
```

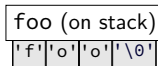
Final value of foo?

- A. "fao"
- B. "zao"
- C. "baz"
- D. "bao"
- E. something else/crash

36

exercise explanation

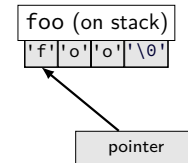
```
1 char foo[4] = "foo";
2   // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';
8 *(foo + 1) = 'a';
```



37

exercise explanation

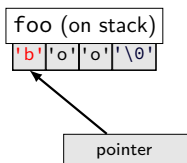
```
1 char foo[4] = "foo";
2   // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';
8 *(foo + 1) = 'a';
```



37

exercise explanation

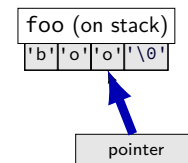
```
1 char foo[4] = "foo";
2   // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';
8 *(foo + 1) = 'a';
```



37

exercise explanation

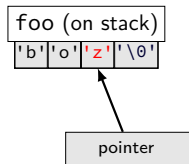
```
1 char foo[4] = "foo";
2   // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';
8 *(foo + 1) = 'a';
```



37

exercise explanation

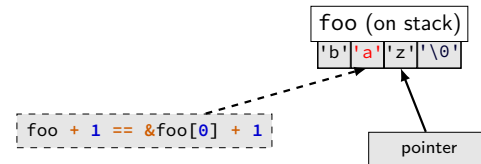
```
1 char foo[4] = "foo";
2   // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';    better style: *pointer = 'z';
8 *(foo + 1) = 'a';
```



37

exercise explanation

```
1 char foo[4] = "foo";
2   // {'f', 'o', 'o', '\0'}
3 char *pointer;
4 pointer = foo;
5 *pointer = 'b';
6 pointer = pointer + 2;
7 pointer[0] = 'z';    better style: *pointer = 'z';
8 *(foo + 1) = 'a';    better style: foo[1] = 'a';
```



37

backup slides

38

example: C that is not C++

valid C and invalid C++:

```
char *str = malloc(100);
```

valid C and valid C++:

```
char *str = (char *) malloc(100);
```

valid C and invalid C++:

```
int class = 1;
```

39