C to assembly / C

last time

AT&T syntax

- destination last
- `d(base,index,scale)` = `memory[d+base+index×scale]`
- `jmp *`

lea

condition codes — ZF, SF, CF, OF

- set by last arithmetic instruction
- ZF = result was zero
- SF = result was negative (sign bit set)
- CF = overflow if treating arithmetic as unsigned
- OF = overflow if treating arithmetic as unsigned

jle, jg, jne, je, ja, jb, etc. use condition codes

- named based on how `cmp` sets condition codes (subtraction)

mistake on quiz question

if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

```
            if (b < 42) goto after_then;
            a += 10;
            goto after_else;
after_then: a *= b;
after_else:
```

if-to-assembly (2)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

```
// a is in %rax, b is in %rbx
    cmpq $42, %rbx    // computes rbx - 42 to 0
                      // i.e compare rbx to 42
    jl after_then     // jump if rbx - 42 < 0
                      // AKA rbx < 42
    addq $10, %rax    // a += 1
    jmp after_else
after_then:
    imulq %rbx, %rax // rax = rax * rbx
after_else:
```

while-to-assembly (1)

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

while-to-assembly (1)

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

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

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
```

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:
```

while exercise: translating?

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

while exercise: translating?

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

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

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:
    ...
    ...
    ...
    ...
```

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:
    ...
    ...
    ...
```

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:
```

compiling switches (1)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    default: ...
}
```

```
    // same as if statement?
    cmpq $1, %rax
    je code_for_1
    cmpq $2, %rax
    je code_for_2
    cmpq $3, %rax
    je code_for_3
    ...
    jmp code_for_default
```

compiling switches (2)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
    // binary search
    cmpq $50, %rax
    jl code_for_less_than_50
    cmpq $75, %rax
    jl code_for_50_to_75
    ...
code_for_less_than_50:
    cmpq $25, %rax
    jl less_than_25_cases
    ...
```

compiling switches (3)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
// jump table
cmpq $100, %rax
jg code_for_default
cmpq $1, %rax
jl code_for_default
jmp *table(,%rax,8)
```

```
table:
  // not instructions
  // .quad = 64-bit (4 x 16) constant
  .quad code_for_1
  .quad code_for_2
  .quad code_for_3
  .quad code_for_4
     ...
```

computed jumps

```
    cmpq $100, %rax
    jg code_for_default
    cmpq $1, %rax
    jl code_for_default
    // jump to memory[table + rax * 8]
      // table of pointers to instructions
    jmp *table(,%rax,8)
    // intel: jmp QWORD PTR[rax*8 + table]
    ...
table:
    .quad code_for_1
    .quad code_for_2
    .quad code_for_3
    ...
```

C Data Types

Varies between machines(!). For this course:

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

C Data Types

Varies between machines(!). For this course:

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

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
anything `*`	8

truth

~~`bool`~~

truth

~~`bool`~~

`x == 4` is an **`int`**

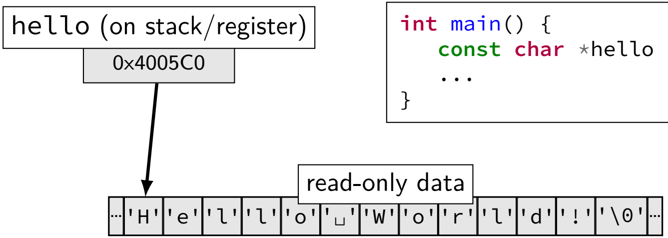
`1` if true; `0` if false

false values in C

`0`

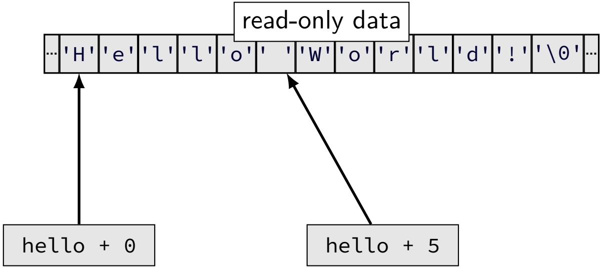
including null pointers — `0` cast to a pointer

strings in C

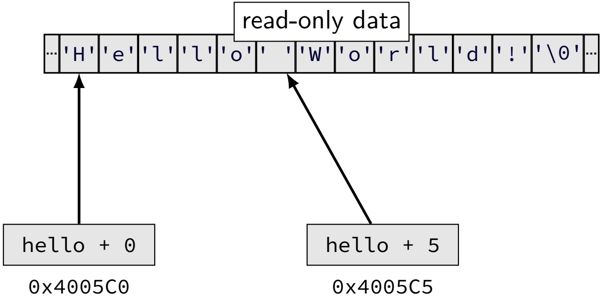



```c
int main() {
    const char *hello = "Hello World!";
    ...
}
```

pointer arithmetic



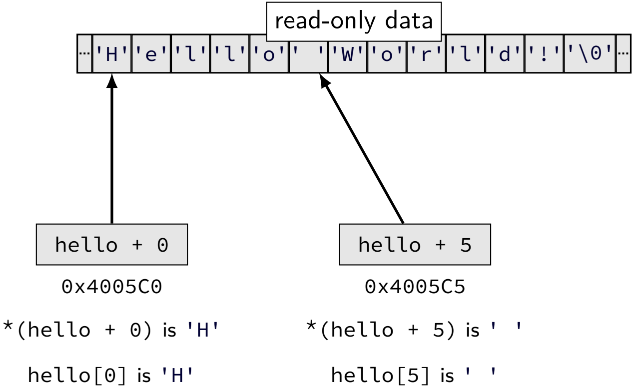
pointer arithmetic



`*(hello + 0)` is `'H'`

`*(hello + 5)` is `' '`

pointer arithmetic



arrays and pointers

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

arrays ‘decay’ into pointers

arrays of non-bytes

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

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

arrays of non-bytes

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

```
int numbers[4] = {10, 11, 12, 13};
int *pointer;
pointer = numbers;
*pointer = 20;  // numbers[0] = 20;
pointer = pointer + 2;
/* adds 8 (2 ints) to address */
*pointe… s[2] = 30;
// numbers is 20, 11, 30, 13
```

assembly: `addq $8, …`

exercise

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

Final value of `foo`?

A. "fao"

B. "zao"

C. "baz"

D. "bao"

E. something else/crash

exercise

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

Final value of `foo`?

A. "fao"

B. "zao"

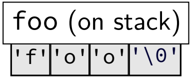
C. "baz"

D. "bao"

E. something else/crash

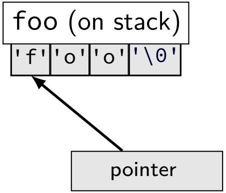
exercise explanation

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



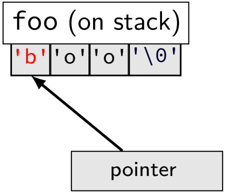
exercise explanation

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



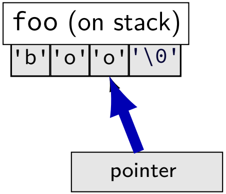
exercise explanation

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



exercise explanation

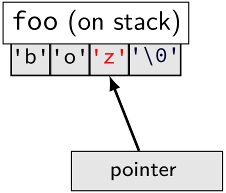
```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



exercise explanation

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

better style: `*pointer = 'z';`

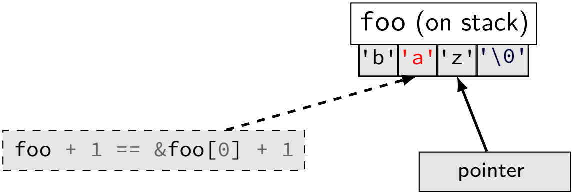


exercise explanation

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

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`



arrays: not quite pointers (1)

```
int array[100];
int *pointer;
```

Legal: `pointer = array;`

same as `pointer = &(array[0]);`

arrays: not quite pointers (1)

```
int array[100];
int *pointer;
```

Legal: `pointer = array;`

same as `pointer = &(array[0]);`

Illegal: ~~`array = pointer;`~~

arrays: not quite pointers (2)

```
int array[100];
int *pointer = array;
```

```
sizeof(array) == 400
```

size of all elements

arrays: not quite pointers (2)

```
int array[100];
int *pointer = array;
```

```
sizeof(array) == 400
```

size of all elements

```
sizeof(pointer) == 8
```

size of address

arrays: not quite pointers (2)

```
int array[100];
int *pointer = array;
```

```
sizeof(array) == 400
```

size of all elements

```
sizeof(pointer) == 8
```

size of address

```
sizeof(&array[0]) == ???
```

(`&array[0]` same as `&(array[0])`)

interlude: command line tips

```
cr4bd@reiss-lenovo:~$ man man
```

man man

```
File  Edit  View  Search  Terminal  Help
MAN(1)                        Manual pager utils                        MAN(1)

NAME
       man - an interface to the on-line reference manuals

SYNOPSIS
       man  [-C  file]  [-d]  [-D]  [--warnings[=warnings]] [-R encoding] [-L locale] [-m sys-
       tem[,...]]  [-M  path]  [-S   list]   [-e   extension]   [-i|-I]   [--regex|--wildcard]
       [--names-only]  [-a]  [-u]  [--no-subpages]  [-P  pager] [-r prompt] [-7] [-E encoding]
       [--no-hyphenation] [--no-justification] [-p  string]  [-t]  [-T[device]]  [-H[browser]]
       [-X[dpi]] [-Z] [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [-w|-W] [-S list] [-i|-I] [--regex] [section] term ...
       man -f [whatis options] page ...
       man -l [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L locale] [-P pager]
       [-r prompt] [-7] [-E encoding] [-p string] [-t]  [-T[device]]  [-H[browser]]  [-X[dpi]]
       [-Z] file ...
       man -w|-W [-C file] [-d] [-D] page ...
       man -c [-C file] [-d] [-D] page ...
       man [-?V]

DESCRIPTION
       man is the system's manual pager.  Each page argument given to man is normally the name
       of a program, utility or function.  The manual page associated with each of these argu-
       ments  is  then  found  and displayed.  A section, if provided, will direct man to look
       only in that section of the manual.  The default action is to  search  in  all  of  the
       available sections following a pre-defined order ("1 n l 8 3 2 3posix 3pm 3perl 5 4 9 6
       7" by default, unless overridden by the SECTION directive in /etc/manpath.config),  and
       to show only the first page found, even if page exists in several sections.

 Manual page man(1) line 1 (press h for help or q to quit)
```

man man

```
File  Edit  View  Search  Terminal  Help
EXAMPLES
       man ls
           Display the manual page for the item (program) ls.

       man -a intro
           Display,  in  succession,  all of the available intro manual pages contained within
           the manual.  It is possible to quit between successive  displays  or  skip  any  of
           them.

       man -t alias | lpr -Pps
           Format the manual page referenced by `alias', usually a shell manual page, into the
           default troff or groff format and pipe it to the printer  named  ps.   The  default
           output  for groff is usually PostScript.  man --help should advise as to which pro-
           cessor is bound to the -t option.

       man -l -Tdvi ./foo.1x.gz > ./foo.1x.dvi
           This command will decompress and format the nroff source  manual  page  ./foo.1x.gz
           into  a device independent (dvi) file.  The redirection is necessary as the -T flag
           causes output to be directed to stdout with no pager.  The output could  be  viewed
           with  a  program  such as xdvi or further processed into PostScript using a program
           such as dvips.

       man -k printf
           Search the short descriptions and manual page names for the keyword printf as regu-
           lar expression.  Print out any matches.  Equivalent to apropos printf.

       man -f smail
           Lookup the manual pages referenced by smail and print out the short descriptions of
           any found.  Equivalent to whatis smail.

 Manual page man(1) line 68 (press h for help or q to quit)
```

tar

the standard Linux/Unix file archive utility

Table of contents: `tar tf filename.tar`

eXtract: `tar xvf filename.tar`

Create: `tar cvf filename.tar directory`

(v: verbose; f: file — default is tape)

tab completion and history

stdio.h

C does not have `<iostream>`

instead `<stdio.h>`

stdio

```
cr4bd@power1
: /if22/cr4bd ; man stdio

…
STDIO(3)                Linux Programmer's Manual               STDIO(3)

NAME
       stdio - standard input/output library functions

SYNOPSIS
       #include <stdio.h>

       FILE *stdin;
       FILE *stdout;
       FILE *stderr;

DESCRIPTION
       The  standard  I/O  library  provides  a  simple  and  efficient
       buffered stream I/O interface.  Input and output is mapped  into
       logical  data  streams  and the physical I/O characteristics are
       concealed.  The functions and  macros  are  listed  below;  more
       information is available from the individual man pages.
```

stdio

```
STDIO(3)                Linux Programmer's Manual               STDIO(3)

NAME
       stdio - standard input/output library functions

…

   List of functions
       Function      Description
       ---------------------------------------------------------------
       clearerr      check and reset stream status
       fclose        close a stream

…

       printf        formatted output conversion

…
```

printf

```
int custNo = 1000;
const char *name = "Jane Smith"
    printf("Customer #%d: %s\n " ,
        custNo, name);
// "Customer #1000: Jane Smith"
// same as:
cout << "Customer #" << custNo
     << ": " << name << endl;
```

printf

```
int custNo = 1000;
const char *name = "Jane Smith"
    printf("Customer #%d: %s\n " ,
        custNo, name);
// "Customer #1000: Jane Smith"
// same as:
cout << "Customer #" << custNo
     << ": " << name << endl;
```

printf

```
int custNo = 1000;
const char *name = "Jane Smith"
    printf("Customer #%d: %s\n " ,
        custNo, name);
// "Customer #1000: Jane Smith"
// same as:
cout << "Customer #" << custNo
      << ": " << name << endl;
```

format string must match types of argument

printf formats quick reference

Specifier	Argument Type	Example(s)
%s	char *	Hello, World!
%p	any pointer	0x4005d4
%d	int/short/char	42
%u	unsigned int/short/char	42
%x	unsigned int/short/char	2a
%ld	long	42
%f	double/float	42.000000
		0.000000
%e	double/float	4.200000e+01
		4.200000e-19
%g	double/float	42, 4.2e-19
%%	(no argument)	%

printf formats quick reference

Specifier	Argument Type	Example(s)
`%s`	char *	`Hello, World!`
`%p`	any pointer	`0x4005d4`
`%d`	int/short/char	`42`
`%u`	unsigned int/short/char	`42`
`%x`	unsigned int/short/char	`2a`
`%ld`	[illegible]	[illegible]
`%f`	double/float	`42.000000` `0.000000`
`%e`	double/float	`4.200000e+01` `4.200000e-19`
`%g`	double/float	`42`, `4.2e-19`
`%%`	(no argument)	`%`

detailed docs: `man 3 printf`

struct

```c
struct rational {
    int numerator;
    int denominator;
};
// ...
struct rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
struct rational *pointer = &two_and_a_half;
printf("%d/%d\n",
       pointer->numerator,
       pointer->denominator);
```

struct

```
struct rational {
    int numerator;
    int denominator;
};
// ...
struct rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
struct rational *pointer = &two_and_a_half;
printf("%d/%d\n",
       pointer->numerator,
       pointer->denominator);
```

typedef

instead of writing:

```
...
unsigned int a;
unsigned int b;
unsigned int c;
```

can write:

```
typedef unsigned int uint;
...
uint a;
uint b;
uint c;
```

typedef struct (1)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// ...
rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
rational *pointer = &two_and_a_half;
printf("%d/%d\n",
       pointer->numerator,
       pointer->denominator);
```

typedef struct (1)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// ...
rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
rational *pointer = &two_and_a_half;
printf("%d/%d\n",
       pointer->numerator,
       pointer->denominator);
```

typedef struct (2)

```c
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// same as:
typedef struct other_name_for_rational {
    int numerator;
    int denominator;
} rational;
```

typedef struct (2)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// same as:
typedef struct other_name_for_rational {
    int numerator;
    int denominator;
} rational;
```

typedef struct (2)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// same as:
typedef struct other_name_for_rational {
    int numerator;
    int denominator;
} rational;
// almost the same as:
typedef struct {
    int numerator;
    int denominator;
} rational;
```

typedef struct (3)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
```

valid ways to declare an instance:

```
struct other_name_for_rational some_variable;
rational some_variable;
```

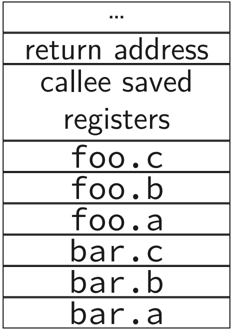
INVALID ways:

```
/* INVALID: */ struct rational some_variable;
/* INVALID: */ other_name_for_rational some_variable;
```

structs aren't references

```c
typedef struct {
    long a; long b; long c;
} triple;
...

triple foo;
foo.a = foo.b = foo.c = 3;
triple bar = foo;
bar.a = 4;
// foo is 3, 3, 3
// bar is 4, 3, 3
```



unsigned and signed types

type	min	max
`signed int` = `signed` = `int`	-2^{31}	$2^{31}-1$
`unsigned int` = `unsigned`	0	$2^{32}-1$
`signed long` = `long`	-2^{63}	$2^{63}-1$
`unsigned long`	0	$2^{64}-1$

⋮

unsigned/signed comparison trap (1)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

unsigned/signed comparison trap (1)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

result is `0`

unsigned/signed comparison trap (1)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

result is `0`

short solution: don't compare signed to unsigned:

```
    (long) x < (long) y
```

unsigned/sign comparison trap (2)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

compiler converts both to same type first

`int` if all possible values fit

otherwise: first operand (x, y) type from this list:

```
unsigned long
long
unsigned int
int
```

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — “K&R C”

very different from modern C

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"

very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

compiler option: `-ansi`, `-std=c90`

looks mostly like modern C

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — “K&R C”

- very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

- compiler option: `-ansi`, `-std=c90`
- looks mostly like modern C

1999: ISO (and ANSI) update C standard — C99

- compiler option: `-std=c99`
- adds: declare variables in middle of block
- adds: `//` comments

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"

very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

compiler option: `-ansi`, `-std=c90`

looks mostly like modern C

1999: ISO (and ANSI) update C standard — C99

compiler option: `-std=c99`

adds: declare variables in middle of block

adds: `//` comments

2011: Second ISO update — C11

undefined behavior example (1)

```c
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX));
}
```

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX));
}
```

without optimizations: `0`

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX));
}
```

without optimizations: `0`

with optimizations: `1`

undefined behavior example (2)

```c
int test(int number) {
    return (number + 1) > number;
}
```

Optimized:

```
test:
    movl    $1, %eax        # eax ← 1
    ret
```

Less optimized:

```
test:
    leal    1(%rdi), %eax # eax ← rdi + 1
    cmpl    %eax, %edi
    setl    %al           # al ← eax < edi
    movzbl  %al, %eax     # eax ← al (pad with zeros)
    ret
```

undefined behavior

compilers can do whatever they want

- what you expect
- crash your program
- ...

common types:

- *signed* integer overflow/underflow
- out-of-bounds pointers
- integer divide-by-zero
- writing read-only data
- out-of-bounds shift

undefined behavior

why undefined behavior?

different architectures work differently
- allow compilers to expose whatever processor does "naturally"
- don't encode any particular machine in the standard

flexibility for optimizations

extracting hexadecimal nibble (1)

problem: given 0xAB
extract 0xA

(hexadecimal digits
called "nibbles")

```
typedef unsigned char byte;
int get_top_nibble(byte value) {
    return ???;
}
```

extracing hexadecimal nibbles (2)

```
typedef unsigned char byte;
int get_top_nibble(byte value) {
    return value / 16;
}
```

aside: division

division is really slow

Intel “Skylake” microarchitecture:

- about six cycles per division
- ...and much worse for eight-byte division
- versus: four additions per cycle

aside: division

division is really slow

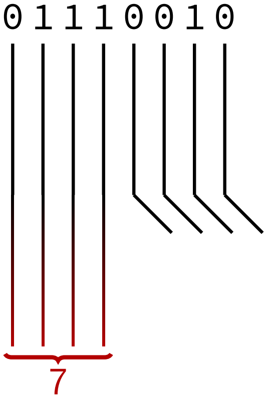
Intel “Skylake” microarchitecture:

- about six cycles per division
- ...and much worse for eight-byte division
- versus: four additions per cycle

but this case: it’s just extracting ‘top wires’ — simpler?

extracting bits in hardware

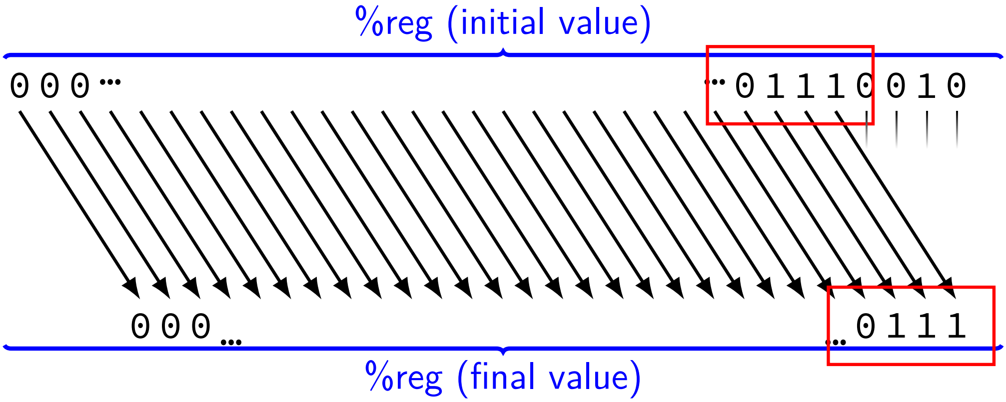
`0111 0010` = 0x72



exposing wire selection

x86 instruction: `shr` — shift right

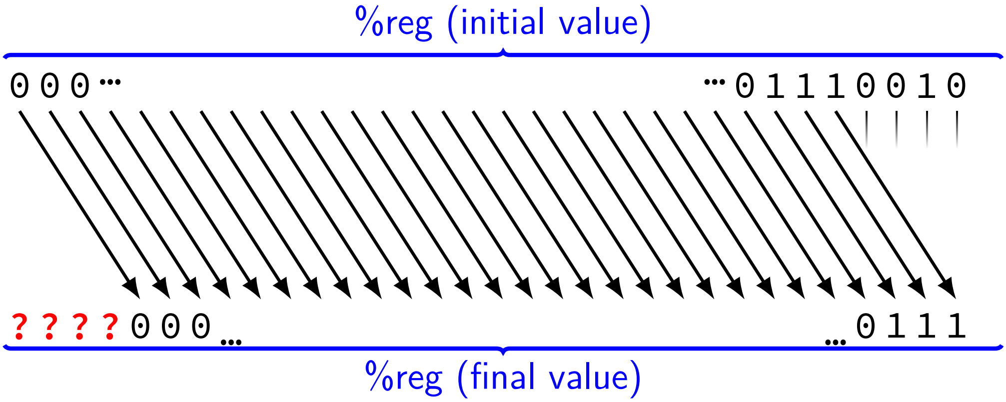
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

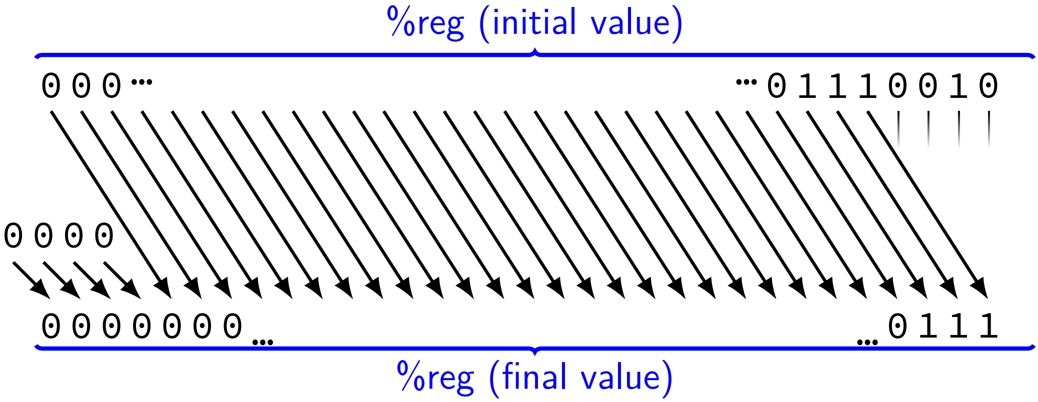
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



exposing wire selection

x86 instruction: `shr` — shift right

`shr $amount, %reg` (or variable: `shr %cl, %reg`)



shift right

x86 instruction: `shr` — shift right

`shr $amount, %reg`

(or variable: `shr %cl, %reg`)

```
get_top_nibble:
 // eax ← dil (low byte of rdi) w/ zero padding
 movzbl %dil, %eax
 shrl $4, %eax
 ret
```

shift right

x86 instruction: `shr` — shift right

`shr $amount, %reg`

(or variable: `shr %cl, %reg`)

```
get_top_nibble:
 // eax ← dil (low byte of rdi) w/ zero padding
 movzbl %dil, %eax
 shrl $4, %eax
 ret
```

shift right

x86 instruction: `shr` — shift right

`shr $amount, %reg`

(or variable: `shr %cl, %reg`)

```
get_top_nibble:
 // eax ← dil (low byte of rdi) w/ zero padding
 movzbl %dil, %eax
 shrl $4, %eax
 ret
```

right shift in C

```
get_top_nibble:
   // eax ← dil (low byte of rdi) w/ zero padding
   movzbl %dil, %eax
   shrl $4, %eax
   ret
```

```c
typedef unsigned char byte;
int get_top_nibble(byte value) {
    return value >> 4;
}
```

right shift in C

```
typedef unsigned char byte;
int get_top_nibble1(byte value) { return value >> 4; }
int get_top_nibble2(byte value) { return value / 16; }
```

right shift in C

```
typedef unsigned char byte;
int get_top_nibble1(byte value) { return value >> 4; }
int get_top_nibble2(byte value) { return value / 16; }
```

example output from optimizing compiler:

```
get_top_nibble1:
    shrb $4, %dil
    movzbl %dil, %eax
    ret

get_top_nibble2:
    shrb $4, %dil
    movzbl %dil, %eax
    ret
```

right shift in math

```
1 >> 0 == 1        0000 0001
1 >> 1 == 0        0000 0000
1 >> 2 == 0        0000 0000

10 >> 0 == 10      0000 1010
10 >> 1 == 5       0000 0101
10 >> 2 == 2       0000 0010
```

$$x \texttt{ >> } y = \lfloor x \times 2^{-y} \rfloor$$

exercise

```
int foo(int)
```

```
foo:
        movl %edi, %eax
        shrl $1, %eax
        ret
```

what is the value of `foo(-2)`?

A. -4 B. -2 C. -1 D. 0

E. a small positive number F. a large positive number

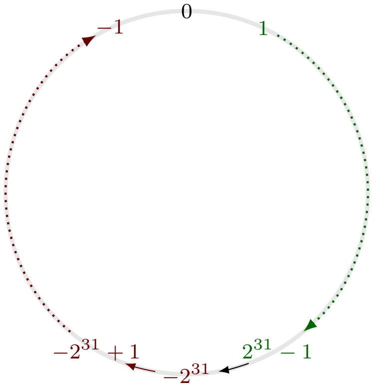
G. a large negative number H. something else

two's complement refresher

	-2^{31}	$+2^{30}$	$+2^{29}$		$+2^{2}$	$+2^{1}$	$+2^{0}$
$-1 =$	1	1	1	...	1	1	1

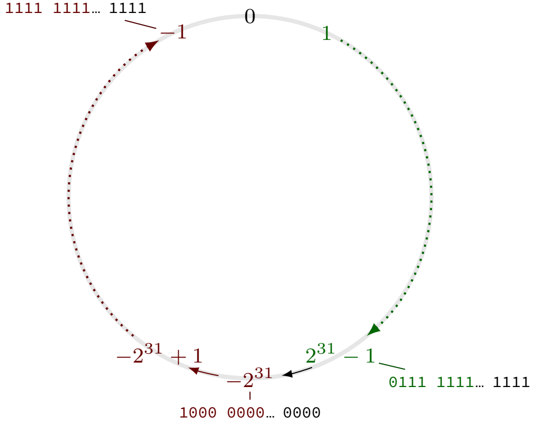
two's complement refresher

	-2^{31}	$+2^{30}$	$+2^{29}$		$+2^{2}$	$+2^{1}$	$+2^{0}$
$-1 =$	1	1	1	…	1	1	1



two's complement refresher

$$-1 = \overset{-2^{31}}{1}\ \overset{+2^{30}}{1}\ \overset{+2^{29}}{1}\ \ldots\ \overset{+2^{2}}{1}\ \overset{+2^{1}}{1}\ \overset{+2^{0}}{1}$$



dividing negative by two

start with $-x$

flip all bits and add one to get x

right shift by one to get $x/2$

flip all bits and add one to get $-x/2$

dividing negative by two

start with $-x$

flip all bits and add one to get x

right shift by one to get $x/2$

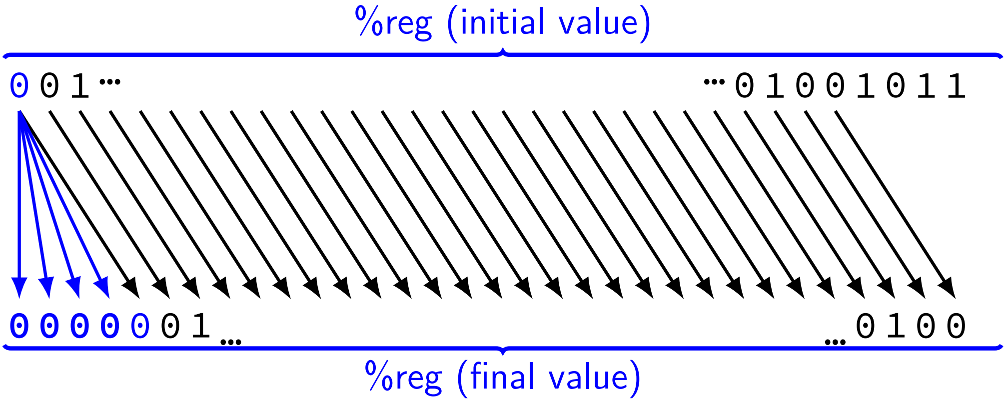
flip all bits and add one to get $-x/2$

same as right shift by one, adding `1`s instead of `0`s
(except for rounding)

arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

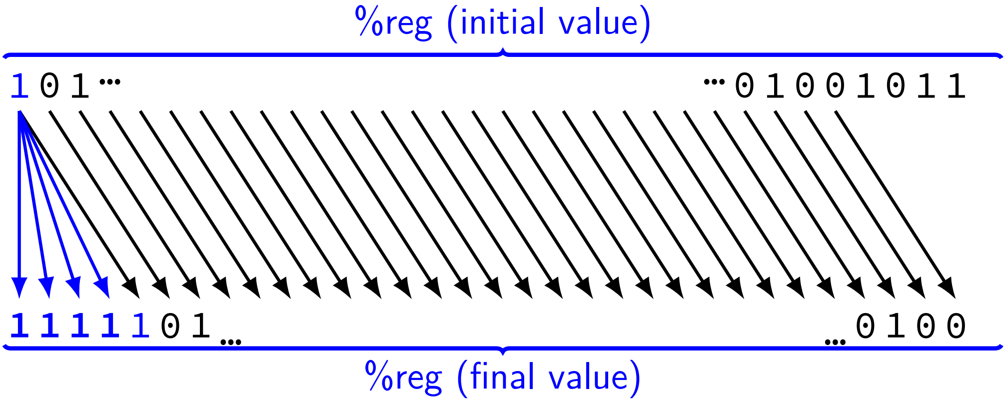
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

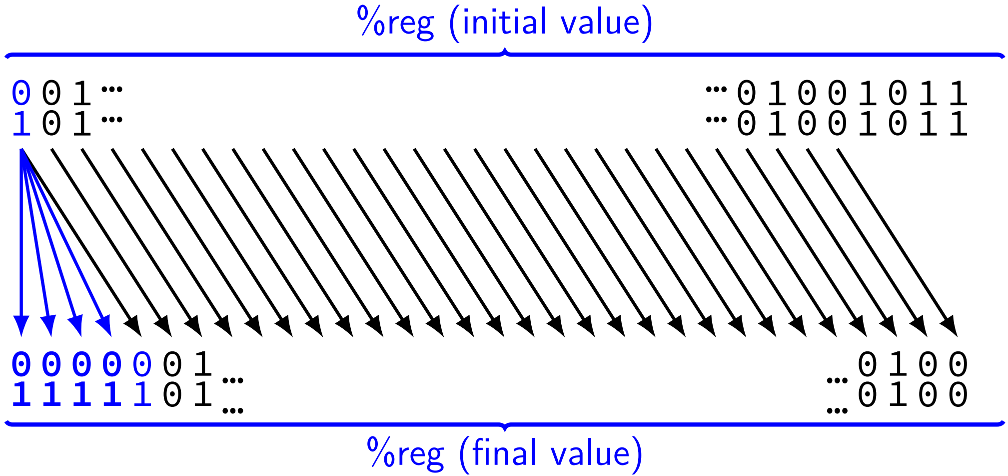
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

`sar $amount, %reg` (or variable: `sar %cl, %reg`)



right shift in C

```
int shift_signed(int x) {
    return x >> 5;
}
unsigned shift_unsigned(unsigned x) {
    return x >> 5;
}
```

```
shift_signed:
    movl %edi, %eax
    sarl $5, %eax
    ret
```

```
shift_unsigned:
    movl %edi, %eax
    shrl $5, eax
    ret
```

standards and shifts in C

signed right shift is implementation-defined

- standard lets compilers choose which type of shift to do
- all x86 compilers I know of — arithmetic

shift amount $\geq$ width of type: undefined

- x86 assembly: only uses lower bits of shift amount

exercise

```
int shiftTwo(int x) {
    return x >> 2;
}
```

```
shiftTwo(-6) = ???
```

A. -4 B. -3 C. -2 D. -1 E. 0

E. some positive number F. something else

dividing negative by two

start with $-x$

flip all bits and add one to get x

right shift by one to get $x/2$

flip all bits and add one to get $-x/2$

same as right shift by one, adding `1`s instead of `0`s
(except for rounding)

divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: “bias” adjustments — described in textbook

divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: "bias" adjustments — described in textbook

```
divideBy8: // GCC generated code
    leal   7(%rdi), %eax // eax ← edi + 7
    testl  %edi, %edi    // set cond. codes based on %edi
    cmovns %edi, %eax    // if (SF = 0) eax ← edi
    sarl   $3, %eax      // arithmetic shift
```

backup slides

do-while-to-assembly (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

do-while-to-assembly (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    int x = 99;
start_loop:
    foo()
    x--;
    if (x >= 0) goto start_loop;
```

do-while-to-assembly (2)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // computes r12 - 0 = r12
    jge start_loop // jump if r12 - 0 >= 0
```

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.

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

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
```

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
```

30 — SF = 0 (not negative), ZF = 0 (not zero)

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

condition codes example (2)

```
movq $−10, %rax  // rax <- (-10)
movq $20, %rbx   // rbx <- 20
cmpq %rax, %rbx  // set cond codes w/ rbx - rax
jle foo // not taken; %rbx - %rax > 0
```

do-while-to-assembly (2)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // computes r12 - 0 = r12
    jge start_loop // jump if r12 - 0 >= 0
```

omitting the cmp

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

```
    movq $99, %r12              // x (r12) ← 99
start_loop:
    call foo                    // foo()
    subq $1, %r12               // x (r12) ← x - 1
    jge  start_loop             // new r12 >= 0?
```

condition codes example: no cmp (3)

```
movq $−10, %rax  // rax ← (-10)
movq $20, %rbx   // rbx ← 20
subq %rax, %rbx  // rbx ← rbx - rax = 30
jle  foo // not taken, %rbx - %rax > 0

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

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.

condition codes examples (4)

```
movq $20, %rbx
addq $−20, %rbx // result is 0
movq $1, %rax   // irrelevant to cond. codes
je   foo        // taken, result is 0
```

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

condition codes: exercise (1)

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

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

$\mathrm{ZF} = ?$

$\mathrm{SF} = ?$

condition codes: exercise (1)

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

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

$\mathtt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathtt{SF} = 0$ (false)	not negative	rax $<=$ rbx

condition codes example: no cmp (3)

```
movq $−10, %rax   // rax ← (-10)
movq $20, %rbx    // rbx ← 20
subq %rax, %rbx   // rbx ← rbx - rax = 30
jle  foo // not taken, %rbx - %rax > 0
```

SF = 0, ZF = 0 (not negative, not zero)

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

SF = 0, ZF = 1 (not negative, is zero)

condition codes examples (4)

```
movq $20, %rbx
addq $−20, %rbx // result is 0
movq $1, %rax   // irrelevant to cond. codes
je   foo        // taken, result is 0
```

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

condition codes: exercise (2)

```
  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)

$\mathrm{ZF} = ?$

$\mathrm{SF} = ?$

condition codes: exercise (2)

```
  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)

$\mathrm{ZF} = 0$ (false)	not zero	result not zero
$\mathrm{SF} = 1$ (true)	negative	result is negative

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)
- `OF` ("overflow flag") — did computation overflow (as signed)?

- `CF` ("carry flag") — did computation overflow (as unsigned)?

- (and one more)

GDB: part of "eflags" register

set by cmp, test, arithmetic

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)
- OF ("overflow flag") — did computation overflow (as signed)?
  - signed conditional jumps: JL, JLE, JG, JGE, ...
  - e.g. JL (jump if less) checks SF + OF
- CF ("carry flag") — did computation overflow (as unsigned)?
  - unsigned conditional jumps: JA, JAE, JB, JBE, ...
  - e.g. JB (jump if below) checks CF

GDB: part of "eflags" register

set by cmp, test, arithmetic

condition codes: exercise (1)

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

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

$\text{ZF} = ?$

$\text{SF} = ?$

condition codes: exercise (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!))

$\mathtt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathtt{SF} = 0$ (false)	not negative	rax <= rbx
$\mathtt{OF} = ?$		
$\mathtt{OF} = ?$		

condition codes: exercise (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!))

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

condition codes: exercise (2)

```
  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)

$\texttt{ZF} = 0$ (false) not zero result not zero

$\texttt{SF} = 1$ (true) negative result is negative

$\texttt{OF} = ?$

$\texttt{OF} = ?$

condition codes: exercise (2)

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

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

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

condition codes: exercise (3)

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

$\mathtt{ZF} = \ ?$

$\mathtt{SF} = \ ?$

$\mathtt{OF} = \ ?$

$\mathtt{CF} = \ ?$

condition codes: exercise (3 solution)

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

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

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

$\texttt{ZF} = 0$ (false) not zero rax and rbx not equal

condition codes: exercise (3 solution)

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

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

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

$\texttt{ZF} = 0$ (false) not zero rax and rbx not equal

condition codes: exercise (3 solution)

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

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

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

$\mathrm{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathrm{SF} = 0$ (false)	not negative	rax <= rbx (if correct)

condition codes: exercise (3 solution)

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

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

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

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

condition codes: exercise (3 solution)

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

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

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

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

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;
```

linked lists / dynamic allocation

```
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...
```

linked lists / dynamic allocation

```
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...
```

linked lists / dynamic allocation

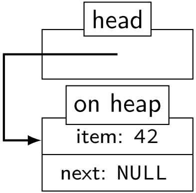
```
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...
```

```
  list* head = malloc(sizeof(list));
    /* C++: new list; */
  head->item = 42;
  head->next = NULL;
  // ...
  free(head);
    /* C++: delete list */
```

linked lists / dynamic allocation

```
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...

  list* head = malloc(sizeof(list));
    /* C++: new list; */
  head->item = 42;
  head->next = NULL;
  // ...
  free(head);
    /* C++: delete list */
```

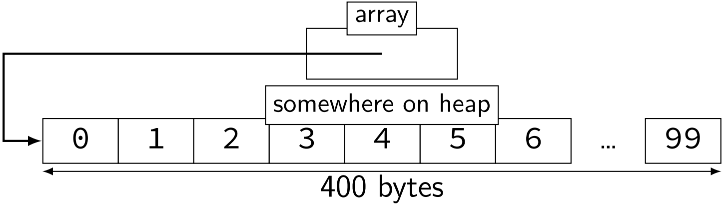


dynamic arrays

```
int *array = malloc(sizeof(int)*100);
  // C++: new int[100]
for (i = 0; i < 100; ++i) {
    array[i] = i;
}
// ...
free(array); // C++: delete[] array
```

dynamic arrays

```
int *array = malloc(sizeof(int)*100);
  // C++: new int[100]
for (i = 0; i < 100; ++i) {
    array[i] = i;
}
// ...
free(array); // C++: delete[] array
```



man chmod

```
File  Edit  View  Search  Terminal  Help
CHMOD(1)                         User Commands                        CHMOD(1)

NAME
       chmod - change file mode bits

SYNOPSIS
       chmod [OPTION]... MODE[,MODE]... FILE...
       chmod [OPTION]... OCTAL-MODE FILE...
       chmod [OPTION]... --reference=RFILE FILE...

DESCRIPTION
       This  manual page documents the GNU version of chmod.  chmod changes the file mode bits
       of each given file according to mode, which can be either a symbolic representation  of
       changes to make, or an octal number representing the bit pattern for the new mode bits.

       The  format  of a symbolic mode is [ugoa...][[-+=][perms...]...], where perms is either
       zero or more letters from the set rwxXst, or a single letter from the set ugo.   Multi-
       ple symbolic modes can be given, separated by commas.

       A  combination  of  the  letters  ugoa controls which users' access to the file will be
       changed: the user who owns it (u), other users in the file's group (g), other users not
       in  the  file's group (o), or all users (a).  If none of these are given, the effect is
       as if (a) were given, but bits that are set in the umask are not affected.

       The operator + causes the selected file mode bits to be added to the existing file mode
       bits  of  each  file;  -  causes  them to be removed; and = causes them to be added and
       causes unmentioned bits to be removed except that a directory's  unmentioned  set  user
       and group ID bits are not affected.

       The  letters  rwxXst select file mode bits for the affected users: read (r), write (w),
 Manual page chmod(1) line 1/125 27% (press h for help or q to quit)
```

chmod

```
chmod --recursive og-r /home/USER
```

chmod

```
chmod --recursive og-r /home/USER
```

others and group (student)

− remove

read

chmod

```
chmod --recursive og-r /home/USER
```

user (yourself) / group / others

− remove / + add

read / write / execute or search

a note on precedence

`&foo[42]` is the same as `&(foo[42])` (*not* `(&foo)[42]`)

`*foo[42]` is the same as `*(foo[42])` (*not* `(*foo)[42]`)

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