last time

arrays versus pointers

left shift — arithmetic and logical

left/right shift versus multiply/divide by power of two

bitwise and/or/xor

topics today

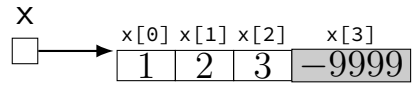
some other C details

interlude: using the command line

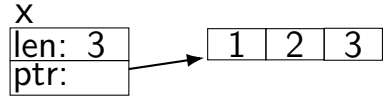
then, doing interesting things with bitwise operators

some lists

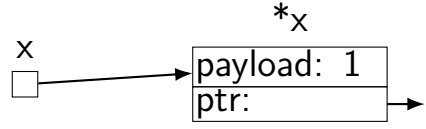
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```




```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```




```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```

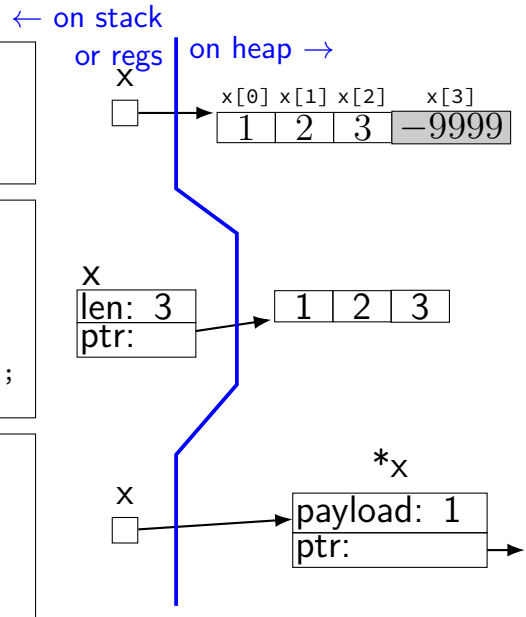


some lists

```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```

```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```



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);
```

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)

```c
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)

```c
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)

```
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;
```

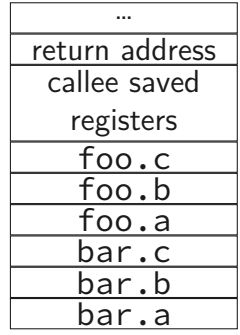
typedef struct (2)

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

structs aren't references

```
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
```

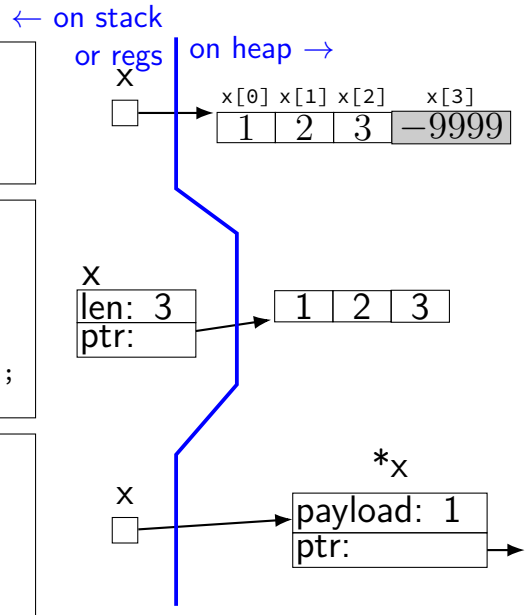


some lists

```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```

```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```



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

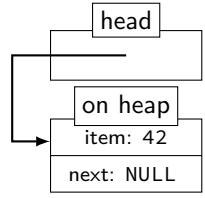
```
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...
```

```
  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 */
```

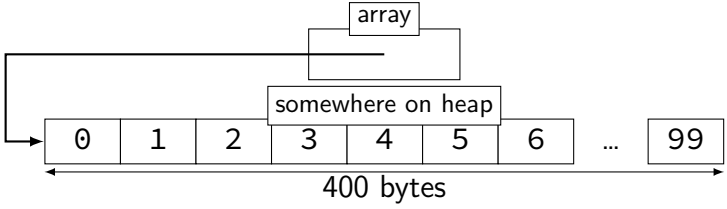


dynamic arrays

```
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

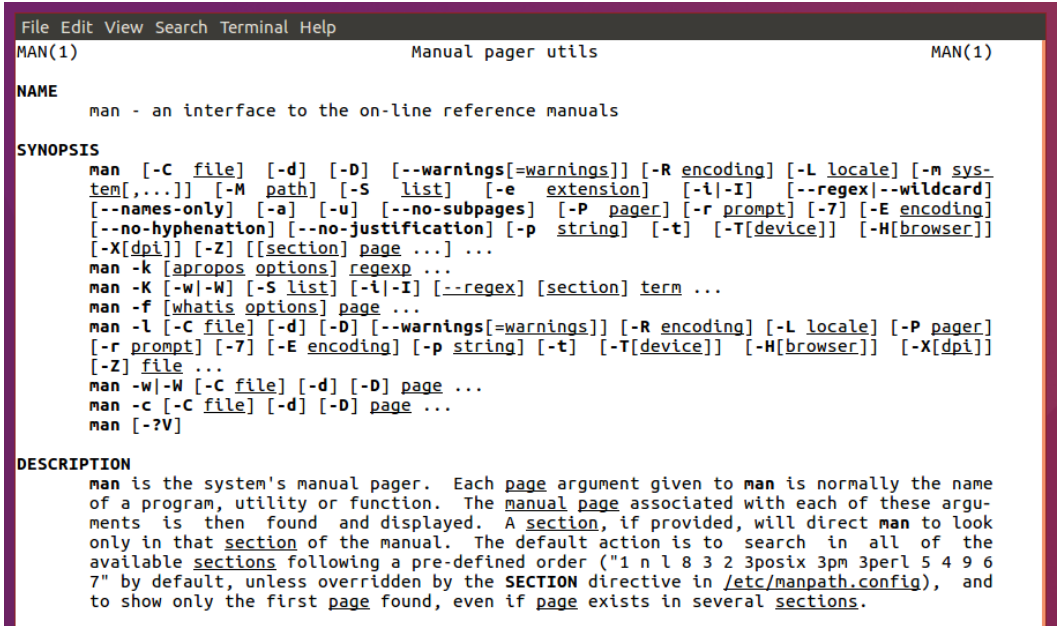
```
int *array = malloc(sizeof(int)*100);
  // C++: new int[100]
for (i = 0; i < 100; ++i) {
    array[i] = i;
}
// ...
free(array); // C++: delete[] array
```



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** system[,...]] [**-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 arguments 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.

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 processor 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 regular 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.

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**. Multiple 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**),

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

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

```
1  int custNo = 1000;
2  const char *name = "Jane Smith"
3      printf("Customer #%d: %s\n " ,
4          custNo, name);
5  // "Customer #1000: Jane Smith"
6  // same as:
7  cout << "Customer #" << custNo
8       << ": " << name << endl;
```

printf

```
1  int custNo = 1000;
2  const char *name = "Jane Smith"
3      printf("Customer #%d: %s\n " ,
4          custNo, name);
5  // "Customer #1000: Jane Smith"
6  // same as:
7  cout << "Customer #" << custNo
8       << ": " << 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		
%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)	%

detailed docs: `man 3 printf`

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)

```c
#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)

```c
#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)

```
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

and/or/xor

AND	**0**	**1**
0	0	0
1	0	1

&

conditionally clear bit
conditionally keep bit

OR	**0**	**1**
0	0	1
1	1	1

|

conditionally set bit

XOR	**0**	**1**
0	0	1
1	1	0

^

conditionally flip bit

extract 0x3 from 0x1234

```
unsigned get_second_nibble1_bitwise(unsigned value)
    return (value >> 4) & 0xF; // 0xF: 00001111
    // like (value / 16) % 16
}

unsigned get_second_nibble2_bitwise(unsigned value)
    return (value & 0xF0) >> 4; // 0xF0: 11110000
    // like (value % 256) / 16;
}
```

extract 0x3 from 0x1234

```
get_second_nibble1_bitwise:
    movl %edi, %eax
    shrl $4, %eax
    andl $0xF, %eax
    ret

get_second_nibble2_bitwise:
    movl %edi, %eax
    andl $0xF0, %eax
    shrl $4, %eax
    ret
```

bit-puzzles

future assignment

bit manipulation puzzles

solve some problem with bitwise ops

- maybe that you could do with normal arithmetic, comparisons, etc.

why?

- good for thinking about HW design
- good for understanding bitwise ops
- unreasonably common interview question type

note: ternary operator

```
w = (x ? y : z)
```

```
if (x) { w = y; } else { w = z; }
```

one-bit ternary

`(x ? y : z)`

constraint: *x, y, and z are 0 or 1*

now: reimplement in C without if/else/||/etc.

(assembly: no jumps probably)

one-bit ternary

```
(x ? y : z)
```

constraint: *x, y, and z are 0 or 1*

now: reimplement in C without if/else/||/etc.

(assembly: no jumps probably)

divide-and-conquer:

```
(x ? y : 0)
(x ? 0 : z)
```

one-bit ternary parts (1)

constraint: *x, y, and z are 0 or 1*

```
(x ? y : 0)
```

one-bit ternary parts (1)

constraint: *x, y, and z are 0 or 1*

```
(x ? y : 0)
```

	y=0	**y=1**
x=0	0	0
x=1	0	1

→ `(x & y)`

one-bit ternary parts (2)

`(x ? y : 0)` = `(x & y)`

one-bit ternary parts (2)

`(x ? y : 0)` = `(x & y)`

`(x ? 0 : z)`

opposite `x`: `~x`

`((~x) & z)`

one-bit ternary

constraint: *x, y, and z are 0 or 1*

```
(x ? y : z)
```

```
(x ? y : 0) | (x ? 0 : z)
```

```
(x & y) | ((~x) & z)
```

multibit ternary

constraint: x *is 0 or 1*

old solution `((x & y) | (~x) & 1)` only gets least sig. bit

`(x ? y : z)`

multibit ternary

constraint: x *is 0 or 1*

old solution `((x & y) | (~x) & 1)` only gets least sig. bit

```
(x ? y : z)
(x ? y : 0) | (x ? 0 : z)
```

constructing masks

constraint: x *is 0 or 1*

`(x ? y : 0)`

if x = 1: want `1111111111...1` (keep y)

if x = 0: want `0000000000...0` (want `0`)

constructing masks

constraint: x *is 0 or 1*

`(x ? y : 0)`

if x = 1: want `1111111111…1` (keep y)

if x = 0: want `0000000000…0` (want `0`)

a trick: −x (`-1` is `1111…1`)

constructing masks

constraint: x *is 0 or 1*

`(x ? y : 0)`

if x = 1: want `1111111111...1` (keep y)

if x = 0: want `0000000000...0` (want `0`)

a trick: −x (`-1` is `1111...1`)

```
((-x) & y)
```

constructing other masks

constraint: x *is 0 or 1*

`(x ? 0 : z)`

if x = ~~1~~ 0: want `1111111111…1`

if x = ~~0~~ 1: want `0000000000…0`

mask: ~~x~~

constructing other masks

constraint: x *is 0 or 1*

`(x ? 0 : z)`

if x = 1: 0: want `1111111111…1`

if x = 0: 1: want `0000000000…0`

mask: ~~>x~~ `-(x^1)`

multibit ternary

constraint: x *is 0 or 1*

old solution `((x & y) | (~x) & 1)` only gets least sig. bit

```
(x ? y : z)
(x ? y : 0) | (x ? 0 : z)
((-x) & y) | ((-(x ^ 1)) & z)
```

fully multibit

~~constraint: x is 0 or 1~~

```
(x ? y : z)
```

fully multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

easy C way: `!x` = 0 or 1, `!!x` = 0 or 1

- x86 assembly: `testq %rax, %rax` then `sete`/`setne` (copy from ZF)

fully multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

easy C way: `!x` = 0 or 1, `!!x` = 0 or 1

x86 assembly: `testq %rax, %rax` then `sete`/`setne` (copy from ZF)

`(x ? y : 0) | (x ? 0 : z)`

`((-!!x) & y) | ((-!x) & z)`

simple operation performance

typical modern desktop processor:

bitwise and/or/xor, shift, add, subtract, compare — $\sim$ 1 cycle
integer multiply — $\sim$ 1-3 cycles
integer divide — $\sim$ 10-150 cycles

(smaller/simpler/lower-power processors are different)

simple operation performance

typical modern desktop processor:

bitwise and/or/xor, shift, add, subtract, compare — $\sim$ 1 cycle
integer multiply — $\sim$ 1-3 cycles
integer divide — $\sim$ 10-150 cycles

(smaller/simpler/lower-power processors are different)

add/subtract/compare are more complicated in hardware!

but *much* more important for typical applications

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

how do we solve is x is two bits? four bits?

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

how do we solve is x is two bits? four bits?

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

wasted work (1)

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

in general: `(x & 1) | (y & 1) == (x | y) & 1`

wasted work (1)

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

in general: `(x & 1) | (y & 1) == (x | y) & 1`

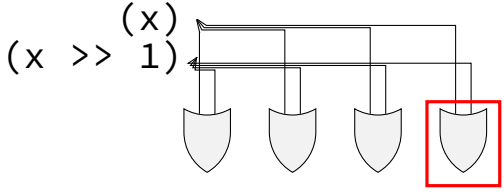
```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

wasted work (2)

4-bit any set: `(x | (x >> 1)| (x >> 2) | (x >> 3)) & 1`

performing 3 bitwise ors

...each bitwise or does 4 OR operations



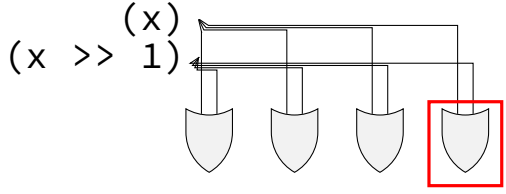
wasted work (2)

4-bit any set: `(x | (x >> 1)| (x >> 2) | (x >> 3)) & 1`

performing 3 bitwise ors

...each bitwise or does 4 OR operations

but only result of one of the 4!



any-bit: divide and conquer

four-bit input $x = x_1x_2x_3x_4$

`x | (x >> 1)` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1y_2y_3y_4$

any-bit: divide and conquer

four-bit input $x = x_1x_2x_3x_4$

`x | (x >> 1)` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1y_2y_3y_4$

`y | (y >> 2)` $= (y_1|0)(y_2|0)(y_3|y_1)(y_4|y_2) = z_1z_2z_3z_4$

$z_4 = (y_4|y_2) = ((x_2|x_1)|(x_4|x_3)) = x_4|x_3|x_2|x_1$ "is any bit set?"

any-bit: divide and conquer

four-bit input $x = x_1x_2x_3x_4$

`x | (x >> 1)` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1y_2y_3y_4$

`y | (y >> 2)` $= (y_1|0)(y_2|0)(y_3|y_1)(y_4|y_2) = z_1z_2z_3z_4$

$z_4 = (y_4|y_2) = ((x_2|x_1)|(x_4|x_3)) = x_4|x_3|x_2|x_1$ "is any bit set?"

```
unsigned int any_of_four(unsigned int x) {
    int part_bits = (x >> 1) | x;
    return ((part_bits >> 2) | part_bits) & 1;
}
```

any-bit-set: 32 bits

```
unsigned int any(unsigned int x) {
    x = (x >> 1) | x;
    x = (x >> 2) | x;
    x = (x >> 4) | x;
    x = (x >> 8) | x;
    x = (x >> 16) | x;
    return x & 1;
}
```

bitwise strategies

use paper, find subproblems, etc.

mask and shift

```
(x & 0xF0) >> 4
```

factor/distribute

```
(x & 1) | (y & 1) == (x | y) & 1
```

divide and conquer

common subexpression elimination

```
return ((-!!x) & y) | ((-!x) & z)
```

becomes

```
d = !x; return ((-!d) & y) | ((-d) & z)
```

exercise

Which of these will swap last and second-to-last bit of an `unsigned int` x? ($abcdef$ becomes $abcdfe$)

```
/* version A */
    return ((x >> 1) & 1) | (x & (~1));

/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));

/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);

/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
```

version A

```
/* version A */
    return ((x >> 1) & 1) | (x & (~1));
    //      ^^^^^^^^^^^^^^
    //       abcdef --> 0abcde -> 00000e

    //                         ^^^^^^^^^^
    //       abcdef --> abcde0

    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //       00000e | abcde0 = abcdee
```

version B

```
/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));
    //      ^^^^^^^^^^^^^^
    //       abcdef --> 0abcde --> 00000e

    //                          ^^^^^^^^^^^^^^^
    //       abcdef --> bcdef0 --> bcde00

    //                                              ^^^^^^^^^
    //       abcdef -->             abcd00
```

version C

```
/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);
    //      ^^^^^^^^^^
    //       abcdef -->             abcd00

    //                    ^^^^^^^^^^^^^^
    //       abcdef --> 00000f --> 0000f0

    //                                    ^^^^^^^^^^^^^
    //       abcdef --> 0abcde --> 00000e
```

version D

```
/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
    //      ^^^^^^^^^^^^^^^
    //       abcdef --> 00000f --> 0000f0

    //                          ^^^^^^^^^^^^^^
    //       abcdef --> 0000ef --> 00000e

    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //      0000fe ^ abcdef --> abcd(f XOR e)(e XOR f)
```

expanded code

```
int lastBit = x & 1;
int secondToLastBit = x & 2;
int rest = x & ~3;
int lastBitInPlace = lastBit << 1;
int secondToLastBitInPlace = secondToLastBit >> 1;
return rest | lastBitInPlace | secondToLastBitInPlace;
```

ISAs being manufactured today

x86 — dominant in desktops, servers

ARM — dominant in mobile devices

POWER — Wii U, IBM supercomputers and some servers

MIPS — common in consumer wifi access points

SPARC — some Oracle servers, Fujitsu supercomputers

z/Architecture — IBM mainframes

Z80 — TI calculators

SHARC — some digital signal processors

RISC V — some embedded

...

microarchitecture v. instruction set

microarchitecture — design of the hardware

- "generations" of Intel's x86 chips
- different microarchitectures for very low-power versus laptop/desktop
- changes in performance/efficiency

instruction set — interface visible by software

- what matters for software compatibility
- many ways to implement (but some might be easier)

ISA variation

instruction set	instr. length	# normal registers	*approx.* # instrs.
x86-64	1–15 byte	16	1500
Y86-64	1–10 byte	15	18
ARMv7	4 byte*	16	400
POWER8	4 byte	32	1400
MIPS32	4 byte	31	200
Itanium	41 bits*	128	300
Z80	1–4 byte	7	40
VAX	1–14 byte	8	150
z/Architecture	2–6 byte	16	1000
RISC V	4 byte*	31	500*

other choices: condition codes?

instead of:

```
cmpq %r11, %r12
je somewhere
```

could do:

```
/* _B_ranch if _EQ_ual */
beq  %r11, %r12, somewhere
```

other choices: addressing modes

ways of specifying operands. examples:

x86-64: `10(%r11,%r12,4)`

ARM: `%r11 << 3` (shift register value by constant)

VAX: `((%r11))` (register value is pointer to pointer)

other choices: number of operands

`add src1, src2, dest`

ARM, POWER, MIPS, SPARC, ...

`add src2, src1=dest`

x86, AVR, Z80, ...

VAX: both

other choices: instruction complexity

instructions that write multiple values?

- x86-64: `push`, `pop`, `movsb`, ...

more?

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC — Complex Instruction Set Computer

some VAX instructions

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

`POLY` *x*, *coefficientsLen*, *coefficientsPtr*
Evaluate the polynomial whose coefficients are pointed to by *coefficientPtr* at the value *x*.

`EDITPC` *sourceLen*, *sourcePtr*, *patternLen*, *patternPtr*
Edit the string pointed to by *sourcePtr* using the pattern string specified by *patternPtr*.

microcode

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

loop in hardware???

typically: lookup sequence of microinstructions (“microcode”)

secret simpler instruction set

Why RISC?

complex instructions were usually not faster

complex instructions were harder to implement

compilers, not hand-written assembly

Why RISC?

complex instructions were usually not faster

complex instructions were harder to implement

compilers, not hand-written assembly

assumption: okay to require compiler modifications

typical RISC ISA properties

fewer, simpler instructions

seperate instructions to access memory

fixed-length instructions

more registers

no “loops” within single instructions

no instructions with two memory operands

few addressing modes

ISAs: who does the work?

CISC-like (harder to implement, easier to use assembly)

- choose instructions with particular assembly language in mind?
- more options for hardware to optimize?
- ...but more resources spent on making hardware correct?
- easier to specialize for particular applications
- less work for compilers

RISC-like (easier to implement, harder to use assembly)

- choose instructions with particular HW implementation in mind?
- less options for hardware to optimize?
- simpler to build/test hardware
- ...so more resources spent on making hardware fast?
- more work for compilers

Is CISC the winner?

well, can't get rid of x86 features

- backwards compatibility matters

more application-specific instructions

but...compilers tend to use more RISC-like subset of instructions

common x86 implementations convert to RISC-like "microinstructions"

- relatively cheap because lots of instruction preprocessing needed in 'fast' CPU designs (even for RISC ISAs)

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64 instruction set

based on x86

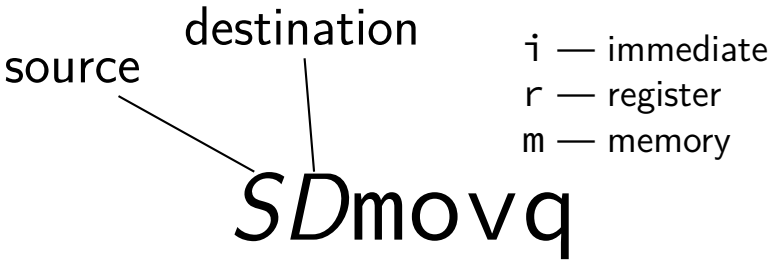
omits most of the 1000+ instructions

leaves

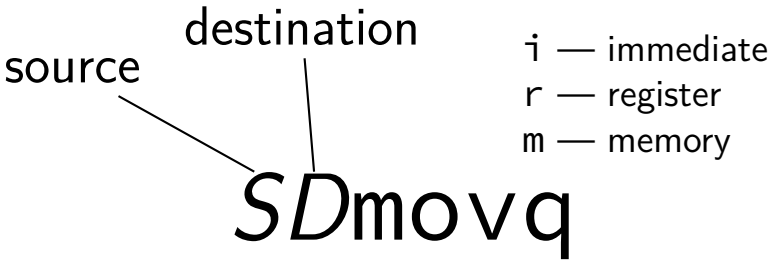
`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64: movq



Y86-64: movq

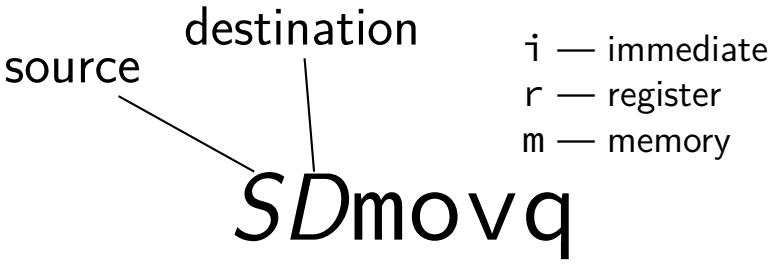


irmovq ~~immovq~~ ~~iimovq~~

rrmovq rmmovq ~~rimovq~~

mrmovq ~~mmmovq~~ ~~mimovq~~

Y86-64: movq



irmovq ~~immovq~~

rrmovq rmmovq

mrmovq ~~mmmovq~~