

# Exam Review

# layers of abstraction

`x += y`

“Higher-level” language: C

`add %rbx, %rax`

Assembly: X86-64

`60 03SIXTEEN`

Machine code: Y86

Hardware Design Language: HCLRS

Gates / Transistors / Wires / Registers

## exercise: nop/add CPU

Let's say we wanted to make **nop+add CPU**. Where would need MUXes?

- A. before one or both of the register file 'register number to read' inputs
- B. before the PC register's input
- C. before one of the register file 'register number to write' inputs
- D. before one of the register file 'register value to write' inputs
- E. before the instruction memory's address input

# interlude: powers of two

|          |       |
|----------|-------|
|          | ...   |
| $2^0$    | 1     |
| $2^1$    | 2     |
| $2^2$    | 4     |
| $2^3$    | 8     |
| $2^4$    | 16    |
| $2^5$    | 32    |
| $2^6$    | 64    |
| $2^7$    | 128   |
| $2^8$    | 256   |
| $2^9$    | 512   |
| $2^{10}$ | 1 024 |

**K** (or Ki)

|          |        |
|----------|--------|
|          | ...    |
| $2^{11}$ | 2 048  |
| $2^{12}$ | 4 096  |
| $2^{13}$ | 8 192  |
| $2^{14}$ | 16 384 |
| $2^{15}$ | 32 768 |
| $2^{16}$ | 65 536 |

$2^{20}$  1 048 576 **M** (or Mi)

$2^{30}$  1 073 741 824 **G** (or Gi)

|          |               |
|----------|---------------|
| $2^{31}$ | 2 147 483 648 |
| $2^{32}$ | 4 294 967 296 |

...

# powers of two: forward

$$2^{35}$$

$$2^{21}$$

$$2^9$$

$$2^{14}$$

# powers of two: forward

$$2^{35} = 2^5 \cdot 2^{30} = 32G \quad (30 = G)$$

$$2^{21}$$

$$2^9$$

$$2^{14}$$

# powers of two: forward

$$2^{35} = 2^5 \cdot 2^{30} = 32G \quad (30 = G)$$

$$2^{21}$$

$$2^9$$

$$2^{14}$$

## powers of two: forward

$$2^{35} = 2^5 \cdot 2^{30} = 32G \quad (30 = G)$$

$$2^{21} = 2^1 \cdot 2^{20} = 2M \quad (20 = M)$$

$$2^9$$

$$2^{14}$$



## powers of two: forward

$$2^{35} = 2^5 \cdot 2^{30} = 32G \quad (30 = G)$$

$$2^{21} = 2^1 \cdot 2^{20} = 2M \quad (20 = M)$$

$$2^9 = 512$$

$$2^{14}$$

## powers of two: forward

$$2^{35} = 2^5 \cdot 2^{30} = 32G \quad (30 = G)$$

$$2^{21} = 2^1 \cdot 2^{20} = 2M \quad (20 = M)$$

$$2^9 = 512$$

$$2^{14} = 2^4 \cdot 2^{10} = 16K$$

# powers of two: backward

16G

128K

4M

256T

## powers of two: backward

$$16\text{G} = 16 \cdot 2^{30} = 2^{30+4} = 2^{34}$$

128K

4M

256T

## powers of two: backward

$$16\text{G} = 16 \cdot 2^{30} = 2^{30+4} = 2^{34}$$

$$128\text{K} = 128 \cdot 2^{10} = 2^{10+7} = 2^{17}$$

4M

256T

## powers of two: backward

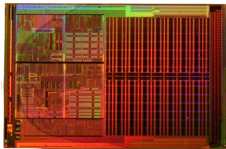
$$16\text{G} = 16 \cdot 2^{30} = 2^{30+4} = 2^{34}$$

$$128\text{K} = 128 \cdot 2^{10} = 2^{10+7} = 2^{17}$$

$$4\text{M} = 4 \cdot 2^{20} = 2^{20+2} = 2^{22}$$

$$256\text{T} = 256 \cdot 2^{40} = 2^{40+8} = 2^{48}$$

# processors and memory



processor

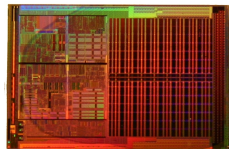


memory

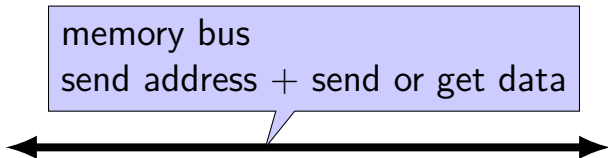
Images:

Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons  
SDRAM by Arnaud 25, via Wikimedia Commons

# processors and memory



processor



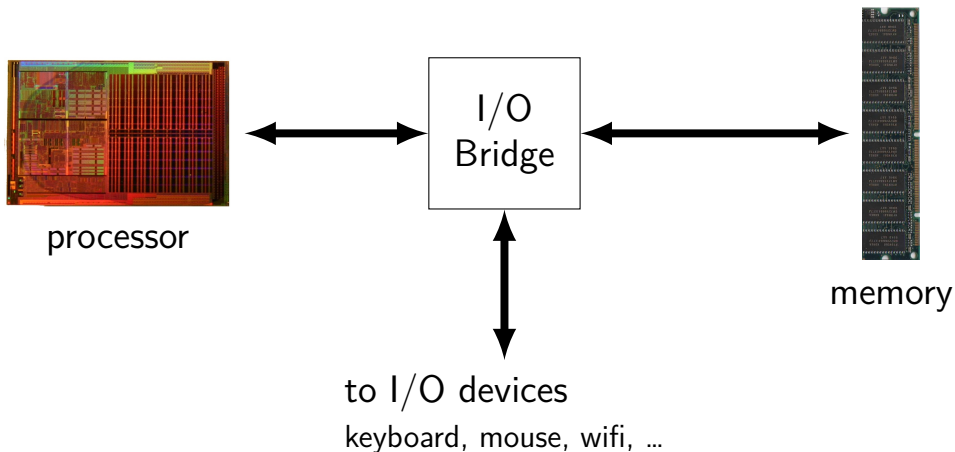
memory

Images:

Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons  
SDRAM by Arnaud 25, via Wikimedia Commons



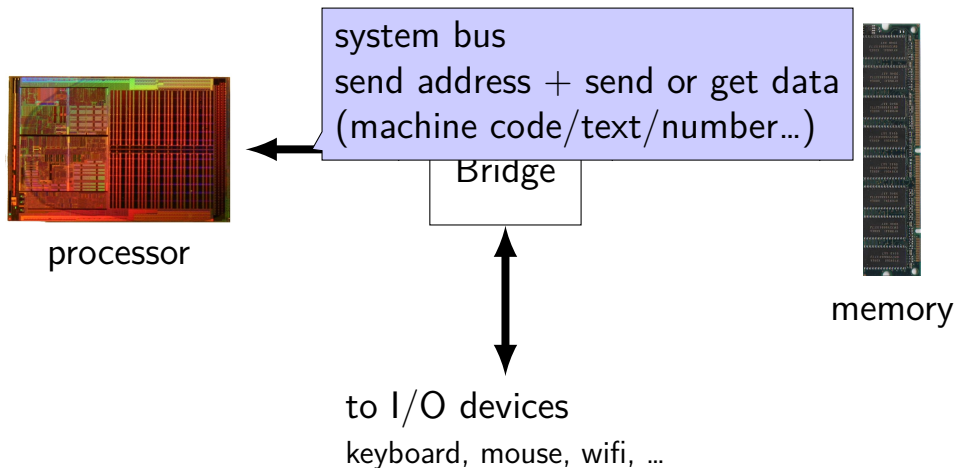
# processors and memory



Images:

Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons  
SDRAM by Arnaud 25, via Wikimedia Commons

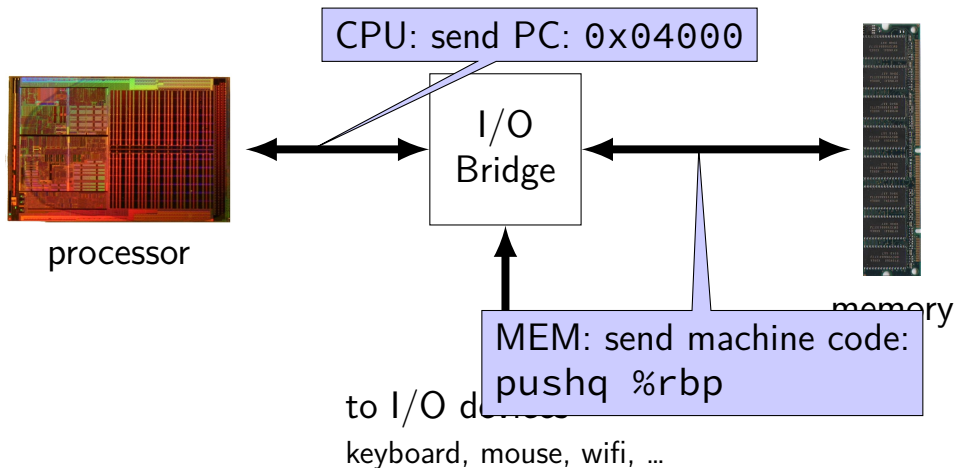
# processors and memory



Images:

Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons  
SDRAM by Arnaud 25, via Wikimedia Commons

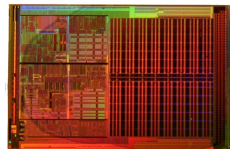
# processors and memory



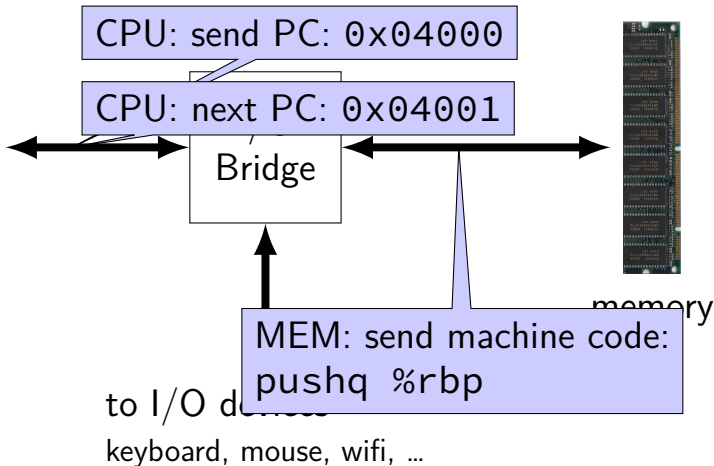
Images:

Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons  
SDRAM by Arnaud 25, via Wikimedia Commons

# processors and memory



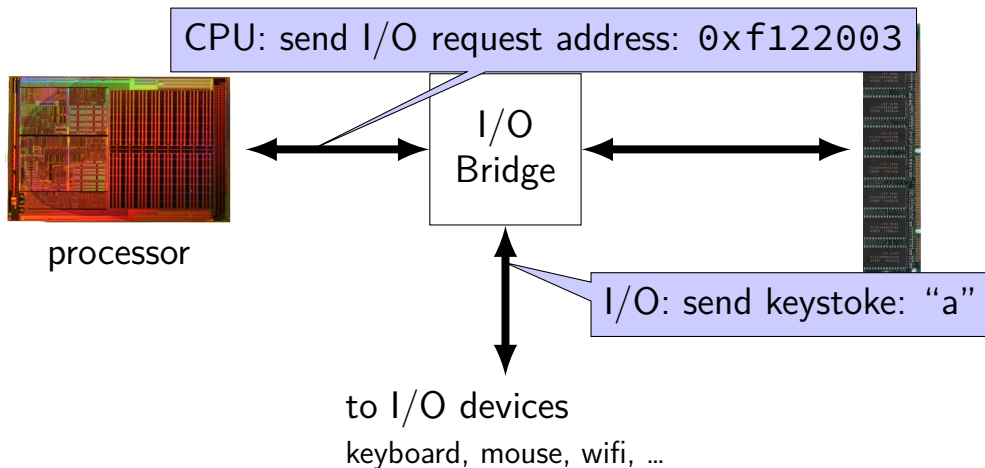
processor



Images:

Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons  
SDRAM by Arnaud 25, via Wikimedia Commons

# processors and memory



Images:

Single core Opteron 8xx die: Dg2fer at the German language Wikipedia, via Wikimedia Commons  
SDRAM by Arnaud 25, via Wikimedia Commons

# memory

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF4 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |
| 0x00000000 | 0xA0  |

# memory

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF2 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |
| 0x00000000 | 0xA0  |

array of bytes (byte = 8 bits)

CPU interprets based on how accessed

# memory

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF2 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |
| 0x00000000 | 0xA0  |

| address    | value |
|------------|-------|
| 0x00000000 | 0xA0  |
| 0x00000001 | 0xE0  |
| 0x00000002 | 0xFE  |
| ...        | ...   |
| 0x00041FFE | 0x60  |
| 0x00041FFF | 0x03  |
| 0x00042000 | 0x00  |
| 0x00042001 | 0x01  |
| 0x00042002 | 0x02  |
| 0x00042003 | 0x03  |
| 0x00042004 | 0x04  |
| 0x00042005 | 0x05  |
| 0x00042006 | 0x06  |
| ...        | ...   |
| 0xFFFFFFF2 | 0xDE  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFFF | 0x14  |



# endianness

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF2 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |

```
int *x = (int*)0x42000;  
cout << *x << endl;
```

# endianness

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF4 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |

```
int *x = (int*)0x42000;  
cout << *x << endl;
```

# endianness

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF2 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |

```
int *x = (int*)0x42000;  
cout << *x << endl;
```

0x03020100 = 50462976

0x00010203 = 66051

# endianness

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF2 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |

```
int *x = (int*)0x42000;  
cout << *x << endl;
```

0x03020100 = 50462976

little endian

(least significant byte has lowest address)

0x00010203 = 66051

big endian

(most significant byte has lowest address)

# endianness

| address    | value |
|------------|-------|
| 0xFFFFFFFF | 0x14  |
| 0xFFFFFFF0 | 0x45  |
| 0xFFFFFFF2 | 0xDE  |
| ...        | ...   |
| 0x00042006 | 0x06  |
| 0x00042005 | 0x05  |
| 0x00042004 | 0x04  |
| 0x00042003 | 0x03  |
| 0x00042002 | 0x02  |
| 0x00042001 | 0x01  |
| 0x00042000 | 0x00  |
| 0x00041FFF | 0x03  |
| 0x00041FFE | 0x60  |
| ...        | ...   |
| 0x00000002 | 0xFE  |
| 0x00000001 | 0xE0  |

```
int *x = (int*)0x42000;  
cout << *x << endl;
```

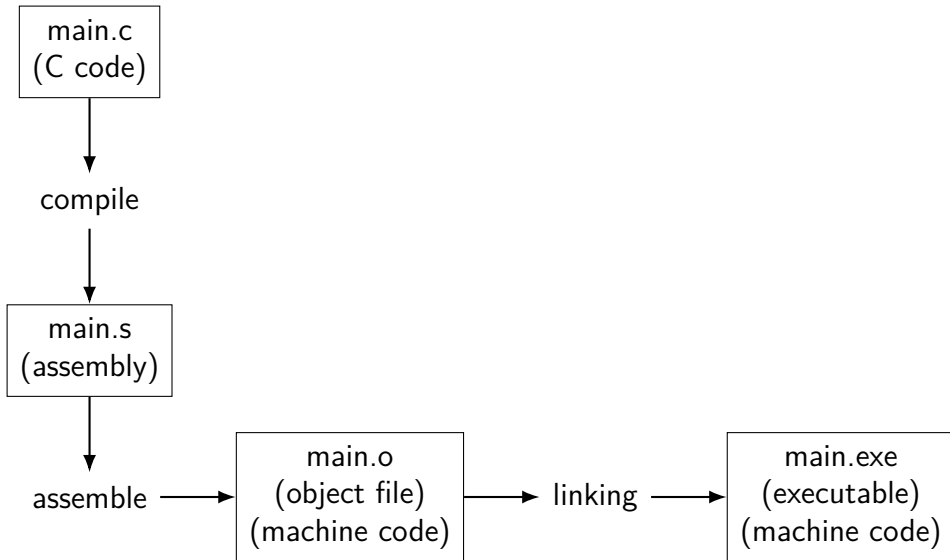
0x03020100 = 50462976

little endian  
(least significant byte has lowest address)

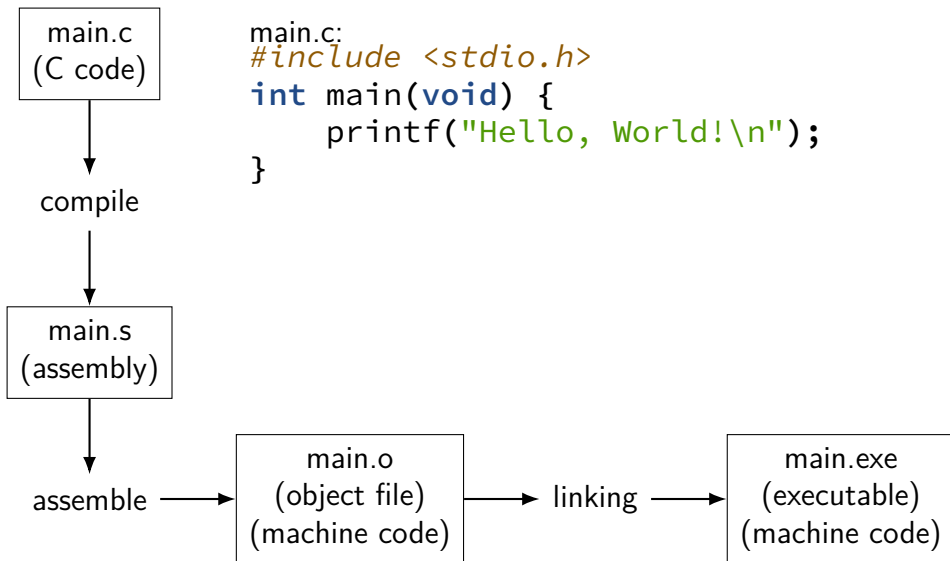
0x00010203 = 66051

big endian  
(most significant byte has lowest address)

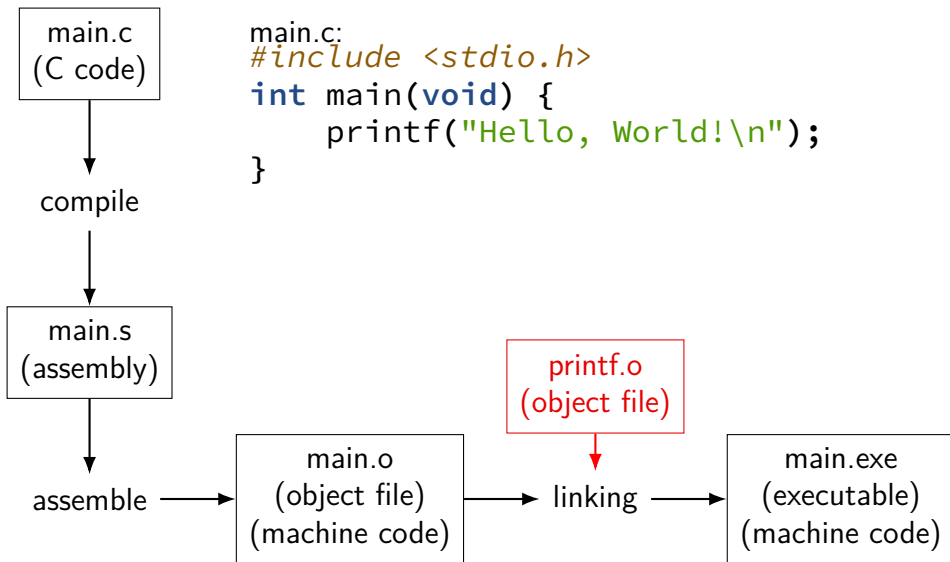
# compilation pipeline



# compilation pipeline



# compilation pipeline





# compilation commands

**compile:** `gcc -S file.c`  $\Rightarrow$  `file.s` (assembly)  
**assemble:** `gcc -c file.s`  $\Rightarrow$  `file.o` (object file)  
**link:** `gcc -o file file.o`  $\Rightarrow$  `file` (executable)

`c+a:` `gcc -c file.c`  $\Rightarrow$  `file.o`  
`c+a+l:` `gcc -o file file.c`  $\Rightarrow$  `file`  
...

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call  puts
    xor    %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call  puts
    xor    %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

```
text (code) segment:
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call  puts
    xor    %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

```
text (code) segment:
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
data segment:
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

```
text (code) segment:
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3

data segment:
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

```
text (code) segment:
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3

data segment:
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00

relocations:
    take 0s at           and replace with
    text, byte 6 (|)    data segment, byte 0
    text, byte 10 (|)  address of puts
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

```
text (code) segment:
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3

data segment:
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00

relocations:
    take 0s at           and replace with
    text, byte 6 (|)    data segment, byte 0
    text, byte 10 (|)  address of puts

symbol table:
main  text byte 0
```



# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:  
48 83 EC 08 BF 00 00 00 00 E8 00 00  
00 00 31 C0 48 83 C4 08 C3

data segment:  
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00

relocations:  
take 0s at                      and replace with  
text, byte 6 (|)                data segment, byte 0  
text, byte 10 (|)               address of puts

symbol table:  
main   text byte 0

+ stdio.o

hello.exe

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

|                   |                      |
|-------------------|----------------------|
| take 0s at        | and replace with     |
| text, byte 6 ( )  | data segment, byte 0 |
| text, byte 10 ( ) | address of puts      |

symbol table:

```
main text byte 0
```

+ stdio.o

hello.exe

(actually binary, but shown as hexadecimal) ...

```
48 83 EC 08 BF A7 02 04 00
E8 08 4A 04 00 31 C0 48
83 C4 08 C3 ...
...(code from stdio.o) ...
48 65 6C 6C 6F 2C 20 57 6F
72 6C 00 ...
...(data from stdio.o) ...
```

# hello.s

```
.LC0:      .section          .rodata.str1.1,"aMS",@progbt
           .string "Hello, World!"
           .text
           .globl  main

main:
           subq    $8, %rsp
           movl   $.LC0, %edi
           call   puts
           movl   $0, %eax
           addq   $8, %rsp
           ret
```

# exercise (1)

main.c:

```
1  #include <stdio.h>
2  void sayHello(void) {
3      puts("Hello, World!");
4  }
5  int main(void) {
6      sayHello();
7  }
```

Which files contain the **memory address** of sayHello?

- A. main.s (assembly)
- B. main.o (object)
- C. main.exe (executable)
- D. B and C
- E. A, B and C
- F. something else

## exercise (2)

main.c:

```
1 #include <stdio.h>
2 void sayHello(void) {
3     puts("Hello, World!");
4 }
5 int main(void) {
6     sayHello();
7 }
```

Which files contain the **literal ASCII string** of Hello, World!?

- A. main.s (assembly)
- B. main.o (object)
- C. main.exe (executable)
- D. B and C
- E. A, B and C
- F. something else

# relocation types

machine code doesn't always use addresses as is

“call function 4303 bytes later”

linker needs to compute “4303”

extra field on relocation list

# dynamic linking (very briefly)

*dynamic linking* — done **when application is loaded**

idea: don't have  $N$  copies of `printf`

other type of linking: *static* (`gcc -static`)

often extra indirection:

```
call functionTable[number_for_printf]
```

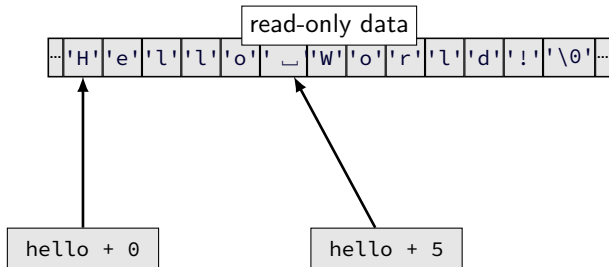
linker fills in `functionTable` instead of changing `calls`

# ldd /bin/ls

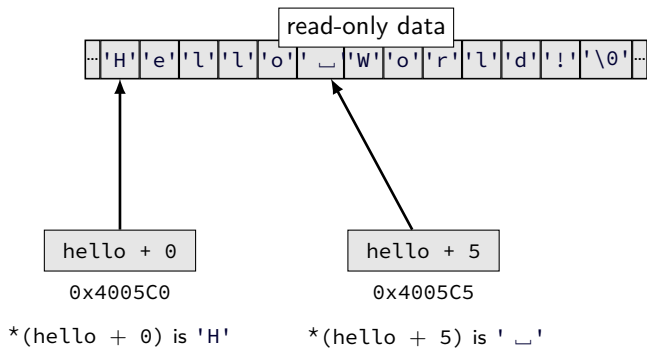
```
$ ldd /bin/ls
linux-vdso.so.1 => (0x00007ffcca9d8000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1
(0x00007f851756f000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f85171a5000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
(0x00007f8516f35000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
(0x00007f8516d31000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8517791000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007f8516b14000)
```



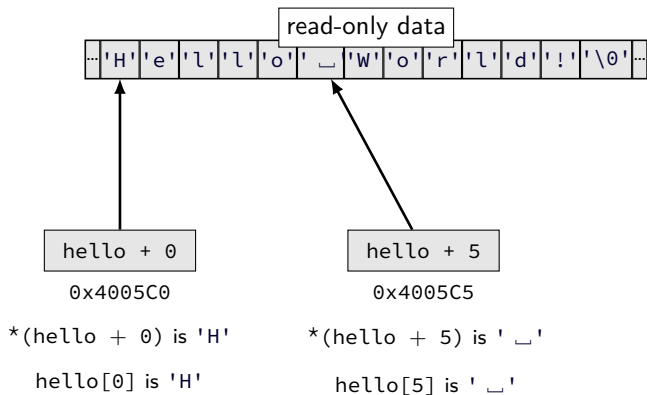
# pointer arithmetic



# pointer arithmetic

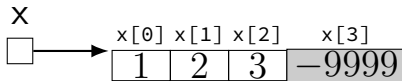


# pointer arithmetic

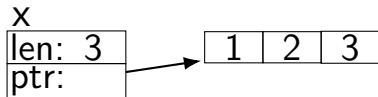


# some lists

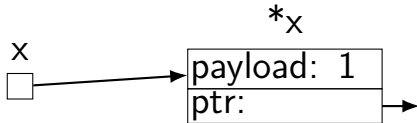
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```



```
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;
...
```

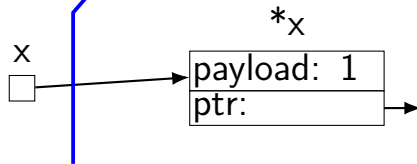
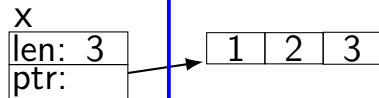
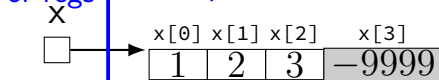
```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

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

← on stack

or regs

on heap →



# AT&T versus Intel syntax (1)

AT&T syntax:

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

Intel syntax:

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

effect (pseudo-C):

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

# AT&T syntax example (1)

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

destination last

()s represent value in memory

constants start with \$

registers start with %

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

l: 4; w: 2; b: 1

sometimes can be omitted

# AT&T syntax example (1)

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

destination last

( )s represent value **in memory**

constants start with \$

registers start with %

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

l: 4; w: 2; b: 1

sometimes can be omitted



# AT&T syntax example (1)

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

destination last

( )s represent value in memory

**constants** start with \$

registers start with %

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

l: 4; w: 2; b: 1

sometimes can be omitted

# AT&T syntax example (1)

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

destination last

( )s represent value in memory

constants start with \$

**registers** start with %

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

l: 4; w: 2; b: 1

sometimes can be omitted

# AT&T syntax example (1)

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

destination last

( )s represent value in memory

constants start with \$

registers start with %

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

l: 4; w: 2; b: 1

sometimes can be omitted

## AT&T versus Intel syntax (2)

AT&T syntax:

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

Intel syntax:

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

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] ← 42
```

## AT&T versus Intel syntax (2)

AT&T syntax:

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

Intel syntax:

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

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] ← 42
```

## AT&T versus Intel syntax (2)

AT&T syntax:

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

Intel syntax:

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

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] ← 42
```

## AT&T versus Intel syntax (2)

AT&T syntax:

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

Intel syntax:

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

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] ← 42
```

## AT&T syntax: addressing

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

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

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

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

`100:  
memory[100]`

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



## AT&T versus Intel syntax (3)

$r8 \leftarrow r8 - rax$

Intel syntax: **sub** r8, rax

AT&T syntax: **subq** %rax, %r8

same for **cmpq**

# AT&T syntax: addresses

```
addq 0x1000, %rax
```

```
// Intel syntax: add rax, QWORD PTR [0x1000]
```

```
// rax ← rax + memory[0x1000]
```

```
addq $0x1000, %rax
```

```
// Intel syntax: add rax, 0x1000
```

```
// rax ← rax + 0x1000
```

no \$ — probably memory address

# AT&T syntax in one slide

destination **last**

() means value **in memory**

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

omit `disp` (defaults to 0)

and/or omit `base` (defaults to 0)

and/or `scale` (defaults to 1)

\$ means constant

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

## extra detail: computed jumps

```
jmpq *%rax
```

```
// Intel syntax: jmp RAX
```

```
// goto RAX
```

```
jmpq *1000(%rax,%rbx,8)
```

```
// Intel syntax: jmp QWORD PTR[RAX+RBX*8+1000]
```

```
// read address from memory at RAX + RBX * 8 + 1000
```

```
// go to that address
```

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

## 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?
```

## 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
```

# 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  
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 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)

## 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)



# 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

## 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

## 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

## 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

## 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    |

## 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 |

## 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 = ?

## 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



## 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

## 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) |

## 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:  $-2^{63} - (2^{63} - 1) = \cancel{-2^{64} + 1} \quad 1$  (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    |

## 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    |

## 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

## 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 |

# 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 (2)

```
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
```

# push/pop

**pushq %rbx**

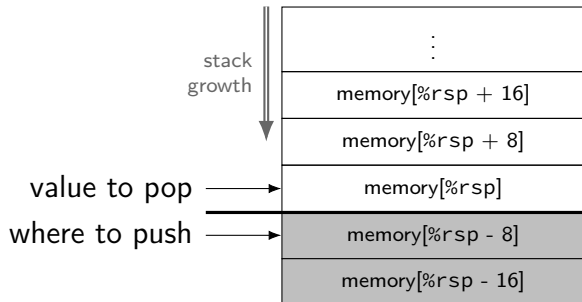
$\%rsp \leftarrow \%rsp - 8$

$\text{memory}[\%rsp] \leftarrow \%rbx$

**popq %rbx**

$\%rbx \leftarrow \text{memory}[\%rsp]$

$\%rsp \leftarrow \%rsp + 8$



# Y86-64 instruction formats

| byte:                | 0 | 1  | 2    | 3  | 4 | 5 | 6 | 7 | 8 | 9 |
|----------------------|---|----|------|----|---|---|---|---|---|---|
| halt                 | 0 | 0  |      |    |   |   |   |   |   |   |
| nop                  | 1 | 0  |      |    |   |   |   |   |   |   |
| rrmovq/cmovCC rA, rB | 2 | cc | rA   | rB |   |   |   |   |   |   |
| irmovq V, rB         | 3 | 0  | F    | rB | V |   |   |   |   |   |
| rmmovq rA, D(rB)     | 4 | 0  | rA   | rB | D |   |   |   |   |   |
| mrmmovq D(rB), rA    | 5 | 0  | rA   | rB | D |   |   |   |   |   |
| OPq rA, rB           | 6 | fn | rA   | rB |   |   |   |   |   |   |
| jCC Dest             | 7 | cc | Dest |    |   |   |   |   |   |   |
| call Dest            | 8 | 0  | Dest |    |   |   |   |   |   |   |
| ret                  | 9 | 0  |      |    |   |   |   |   |   |   |
| pushq rA             | A | 0  | rA   | F  |   |   |   |   |   |   |
| popq rA              | B | 0  | rA   | F  |   |   |   |   |   |   |

# Secondary opcodes: *OPq*

| byte:                       | 0 | 1  | 2    | 3  | 4 | 5   | 6 | 7 | 8 | 9 |
|-----------------------------|---|----|------|----|---|-----|---|---|---|---|
| halt                        | 0 | 0  |      |    |   |     |   |   |   |   |
| nop                         | 1 | 0  |      |    |   |     |   |   |   |   |
| <i>rrmovq/cmovCC rA, rB</i> | 2 | cc | rA   | rB |   |     |   |   |   |   |
| <i>irmovq V, rB</i>         | 3 | 0  | F    | rB | V |     |   |   |   |   |
| <i>rmmovq rA, D(rB)</i>     | 4 | 0  | rA   | rB | D |     |   |   |   |   |
| <i>mrmmovq D(rB), rA</i>    | 5 | 0  | rA   | rB | D |     |   |   |   |   |
| <i>OPq rA, rB</i>           | 6 | fn | rA   | rB | 0 | add |   |   |   |   |
|                             |   |    |      |    | 1 | sub |   |   |   |   |
|                             |   |    |      |    | 2 | and |   |   |   |   |
|                             |   |    |      |    | 3 | xor |   |   |   |   |
| <i>jCC Dest</i>             | 7 | cc | Dest |    |   |     |   |   |   |   |
| <i>call Dest</i>            | 8 | 0  | Dest |    |   |     |   |   |   |   |
| ret                         | 9 | 0  |      |    |   |     |   |   |   |   |
| <i>pushq rA</i>             | A | 0  | rA   | F  |   |     |   |   |   |   |
| <i>popq rA</i>              | B | 0  | rA   | F  |   |     |   |   |   |   |

# Registers: $rA$ , $rB$

| byte:                   | 0 | 1  | 2        |
|-------------------------|---|----|----------|
| halt                    | 0 | 0  |          |
| nop                     | 1 | 0  |          |
| $rrmovq/cmovCC\ rA, rB$ | 2 | cc | $rA\ rB$ |
| $irmovq\ V, rB$         | 3 | 0  | F $rB$   |
| $rmmovq\ rA, D(rB)$     | 4 | 0  | $rA\ rB$ |
| $mrmmovq\ D(rB), rA$    | 5 | 0  | $rA\ rB$ |
| $OPq\ rA, rB$           | 6 | ff | $rA\ rB$ |
| $jCC\ Dest$             | 7 | cc |          |
| $call\ Dest$            | 8 | 0  |          |
| ret                     | 9 | 0  |          |
| $pushq\ rA$             | A | 0  | $rA\ F$  |
| $popq\ rA$              | B | 0  | $rA\ F$  |

|   |      |   |      |
|---|------|---|------|
| 0 | %rax | 8 | %r8  |
| 1 | %rcx | 9 | %r9  |
| 2 | %rdx | A | %r10 |
| 3 | %rbx | B | %r11 |
| 4 | %rsp | C | %r12 |
| 5 | %rbp | D | %r13 |
| 6 | %rsi | E | %r14 |
| 7 | %rdi | F | none |

# Immediates: $V$ , $D$ , $Dest$

| byte:                     | 0 | 1  | 2      | 3    | 4   | 5 | 6 | 7 | 8 | 9 |
|---------------------------|---|----|--------|------|-----|---|---|---|---|---|
| halt                      | 0 | 0  |        |      |     |   |   |   |   |   |
| nop                       | 1 | 0  |        |      |     |   |   |   |   |   |
| rrmovq/cmovCC $rA$ , $rB$ | 2 | cc | $rA$   | $rB$ |     |   |   |   |   |   |
| irmovq $V$ , $rB$         | 3 | 0  | F      | $rB$ | $V$ |   |   |   |   |   |
| rmmovq $rA$ , $D(rB)$     | 4 | 0  | $rA$   | $rB$ | $D$ |   |   |   |   |   |
| mrmmovq $D(rB)$ , $rA$    | 5 | 0  | $rA$   | $rB$ | $D$ |   |   |   |   |   |
| $OPq$ $rA$ , $rB$         | 6 | fn | $rA$   | $rB$ |     |   |   |   |   |   |
| $jCC$ $Dest$              | 7 | cc | $Dest$ |      |     |   |   |   |   |   |
| call $Dest$               | 8 | 0  | $Dest$ |      |     |   |   |   |   |   |
| ret                       | 9 | 0  |        |      |     |   |   |   |   |   |
| pushq $rA$                | A | 0  | $rA$   | F    |     |   |   |   |   |   |
| popq $rA$                 | B | 0  | $rA$   | F    |     |   |   |   |   |   |

# Immediates: $V$ , $D$ , $Dest$

| byte:                     | 0 | 1  | 2      | 3    | 4   | 5 | 6 | 7 | 8 | 9 |
|---------------------------|---|----|--------|------|-----|---|---|---|---|---|
| halt                      | 0 | 0  |        |      |     |   |   |   |   |   |
| nop                       | 1 | 0  |        |      |     |   |   |   |   |   |
| rrmovq/cmovCC $rA$ , $rB$ | 2 | cc | $rA$   | $rB$ |     |   |   |   |   |   |
| irmovq $V$ , $rB$         | 3 | 0  | F      | $rB$ | $V$ |   |   |   |   |   |
| rmmovq $rA$ , $D(rB)$     | 4 | 0  | $rA$   | $rB$ | $D$ |   |   |   |   |   |
| mrmmovq $D(rB)$ , $rA$    | 5 | 0  | $rA$   | $rB$ | $D$ |   |   |   |   |   |
| $OPq$ $rA$ , $rB$         | 6 | fn | $rA$   | $rB$ |     |   |   |   |   |   |
| $jCC$ $Dest$              | 7 | cc | $Dest$ |      |     |   |   |   |   |   |
| call $Dest$               | 8 | 0  | $Dest$ |      |     |   |   |   |   |   |
| ret                       | 9 | 0  |        |      |     |   |   |   |   |   |
| pushq $rA$                | A | 0  | $rA$   | F    |     |   |   |   |   |   |
| popq $rA$                 | B | 0  | $rA$   | F    |     |   |   |   |   |   |



# 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)
```

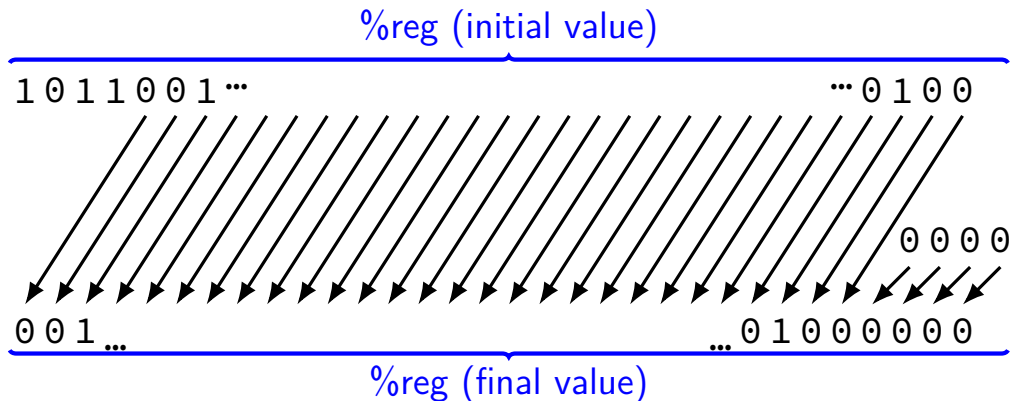
becomes

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

# shift left

x86 instruction: `shl` — shift left

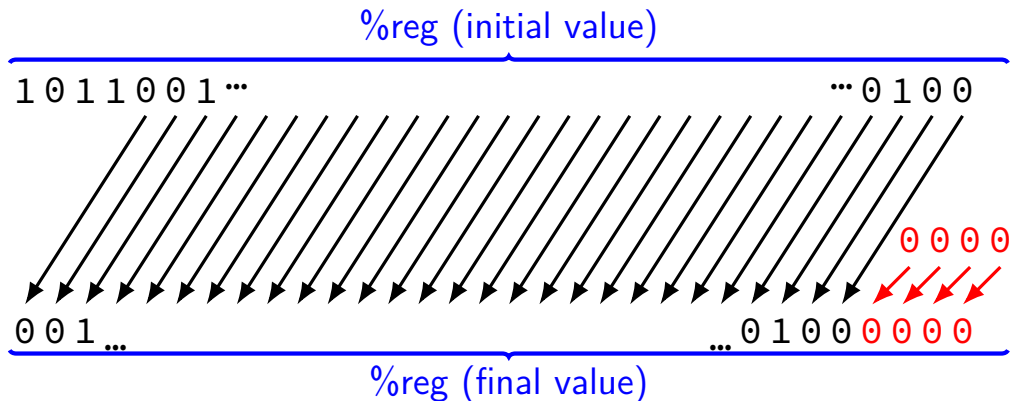
`shl $amount, %reg` (or variable: `shr %cl, %reg`)



# shift left

x86 instruction: `shl` — shift left

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



# left shift in math

1 << 0 == 1

1 << 1 == 2

1 << 2 == 4

0000 0001

0000 0010

0000 0100

10 << 0 == 10

10 << 1 == 20

10 << 2 == 40

0000 1010

0001 0100

0010 1000

# left shift in math

1 << 0 == 1

0000 0001

1 << 1 == 2

0000 0010

1 << 2 == 4

0000 0100

10 << 0 == 10

0000 1010

10 << 1 == 20

0001 0100

10 << 2 == 40

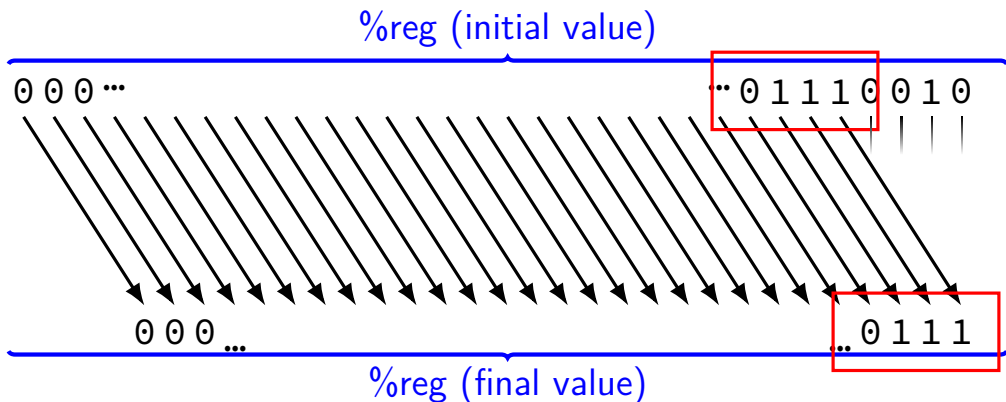
0010 1000

$$x \ll y = x \times 2^y$$

# logical right shift

x86 instruction: `shr` — logical shift right

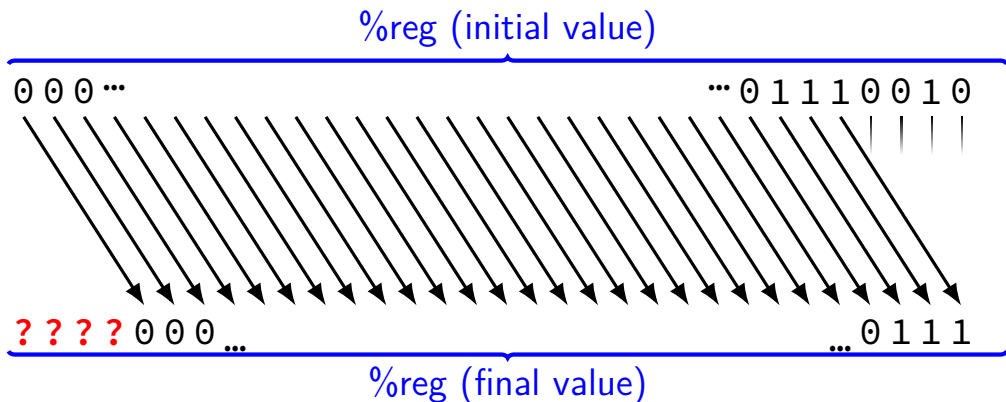
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



# logical right shift

x86 instruction: `shr` — logical shift right

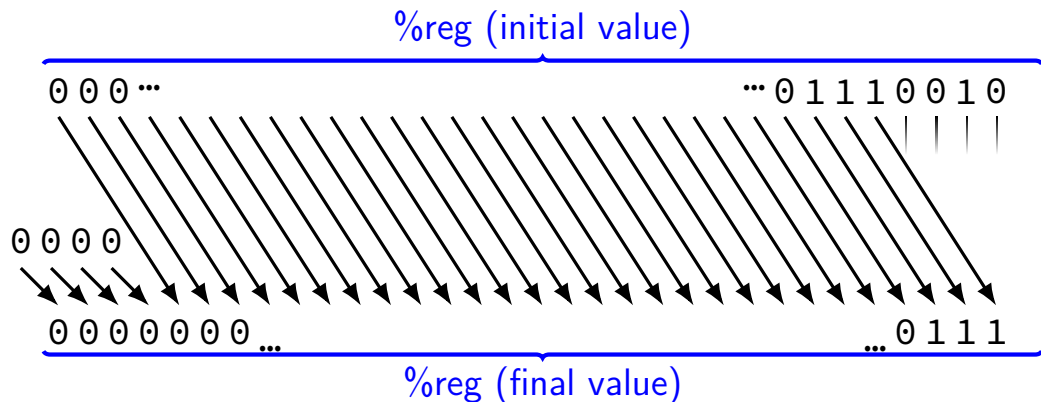
`shr $amount, %reg` (or variable: `shr %cl, %reg`)



# logical right shift

x86 instruction: `shr` — logical shift right

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

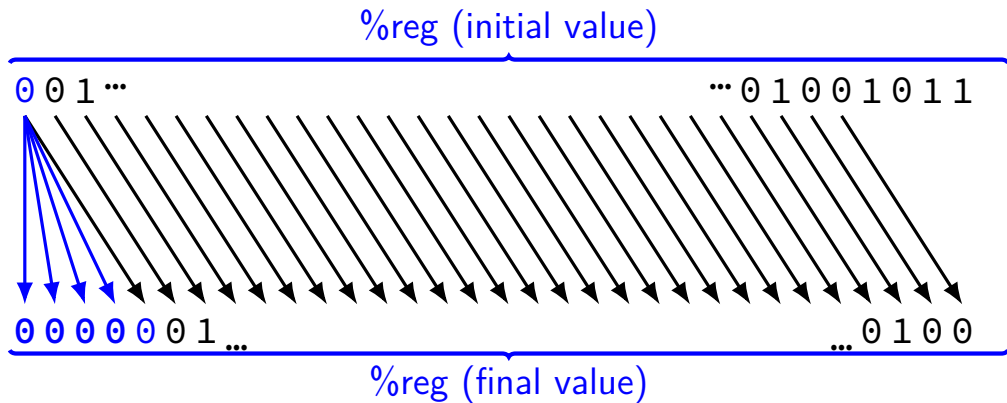




# arithmetic right shift

x86 instruction: `sar` — arithmetic shift right

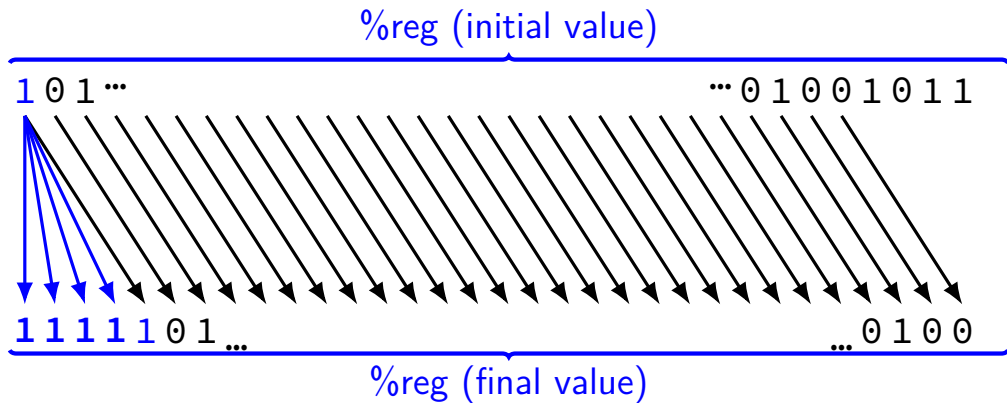
`sar $amount, %reg` (or variable: `sar %cl, %reg`)



# 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; // arithmetic; fill w/ copies of  
}  
unsigned shift_unsigned(unsigned x) {  
    return x >> 5; // logical; fill with zeroes  
}
```

---

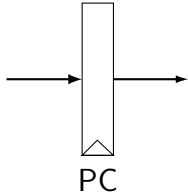
shift\_signed:

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

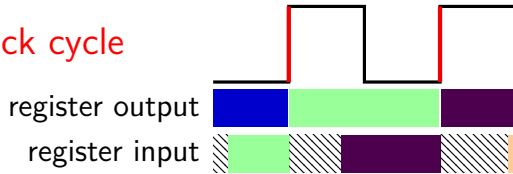
shift\_unsigned:

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

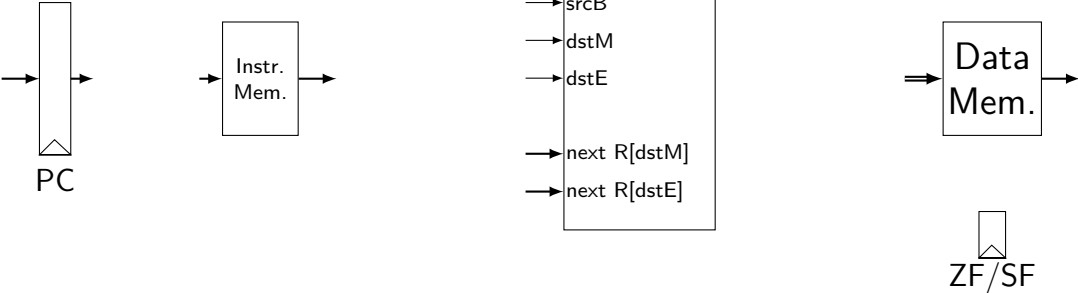
# registers



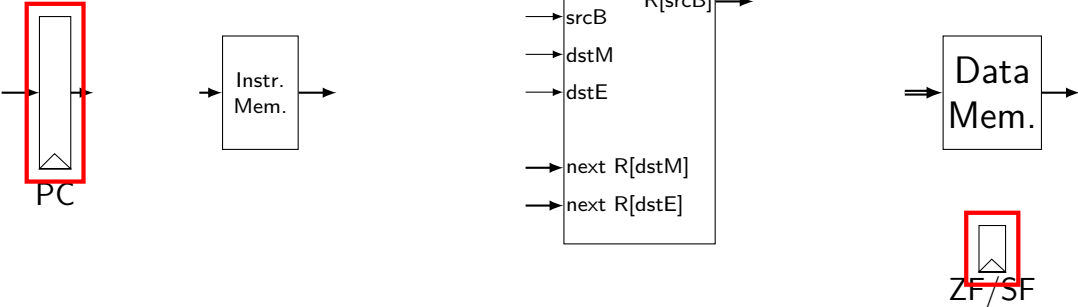
updates every **clock cycle**



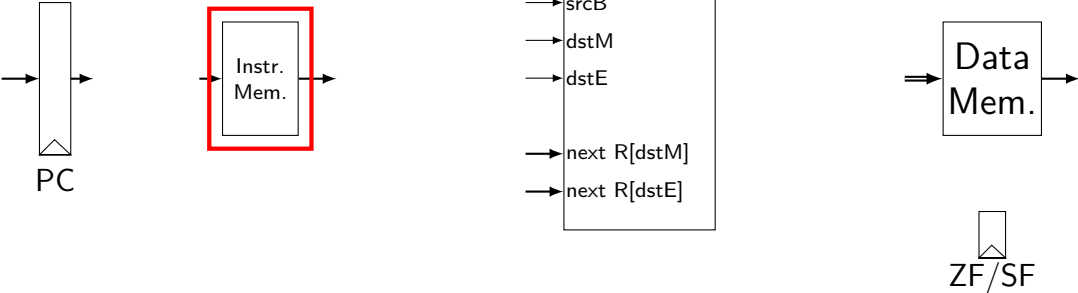
# state in Y86-64



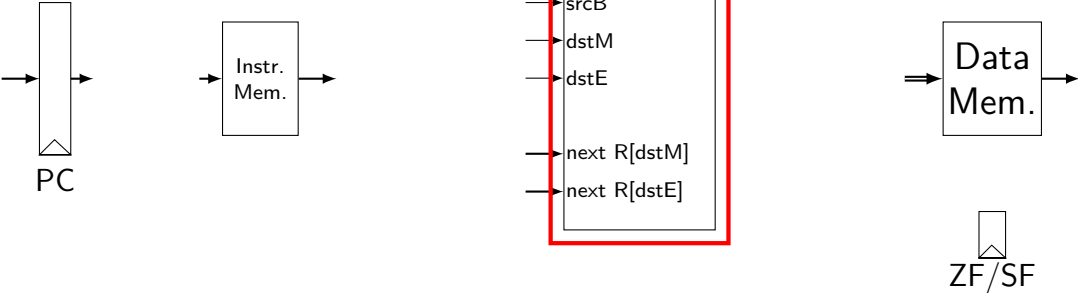
# state in Y86-64



# state in Y86-64

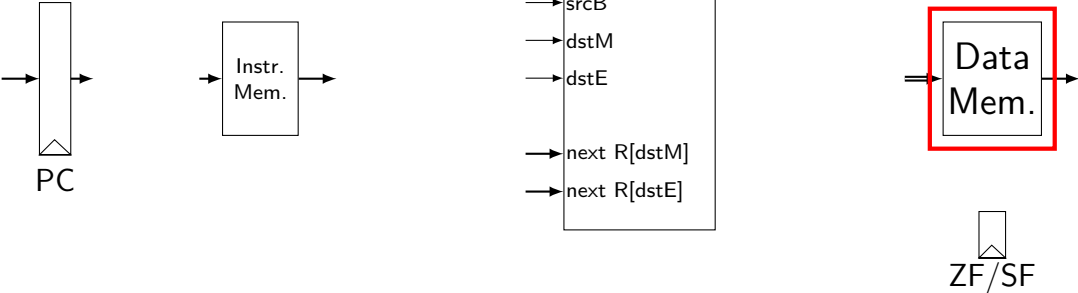


# state in Y86-64

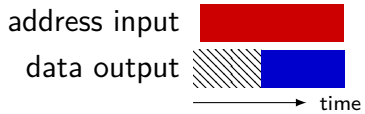
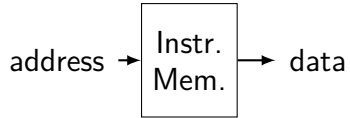




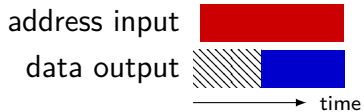
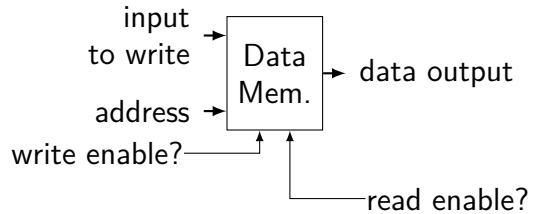
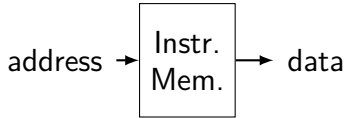
# state in Y86-64



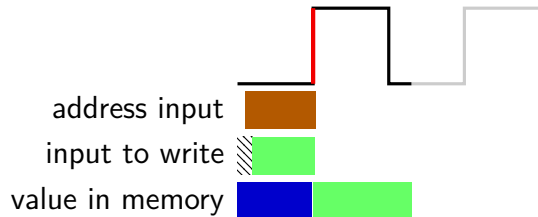
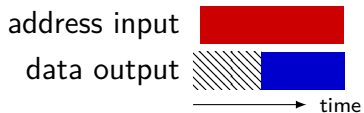
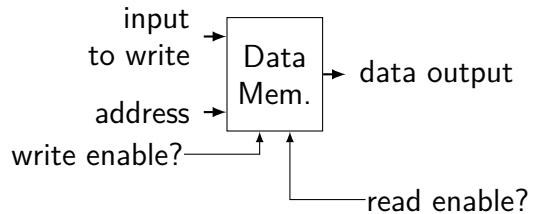
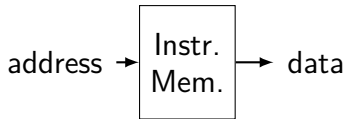
# memories



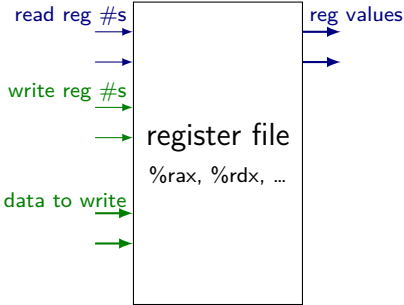
# memories



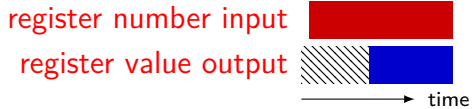
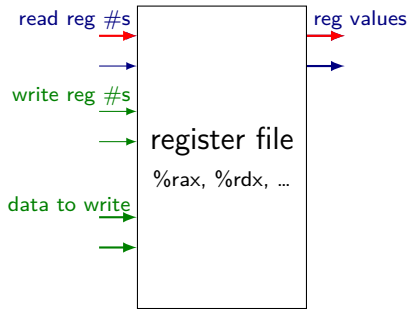
# memories



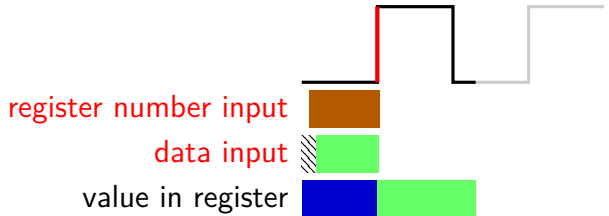
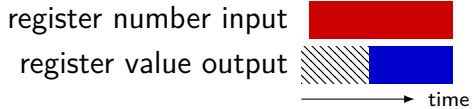
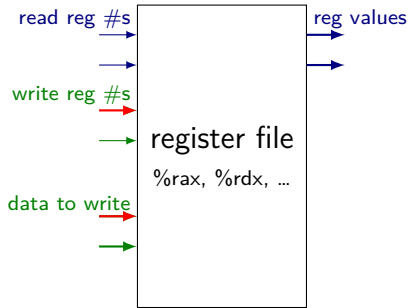
# register file



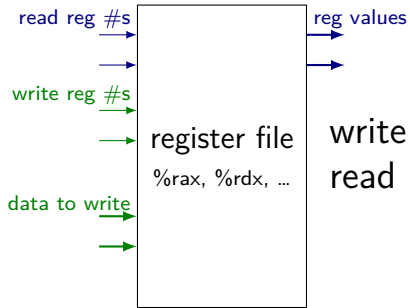
# register file



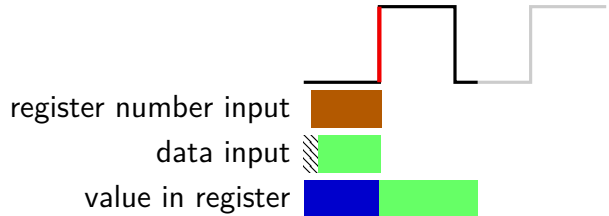
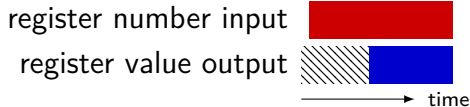
# register file



# register file

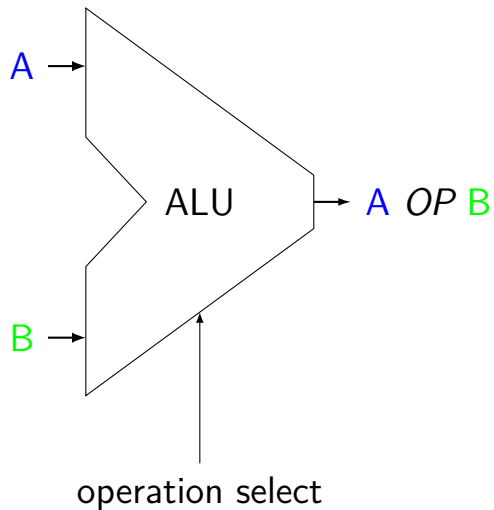


write register #15: write is ignored  
read register #15: value is always 0



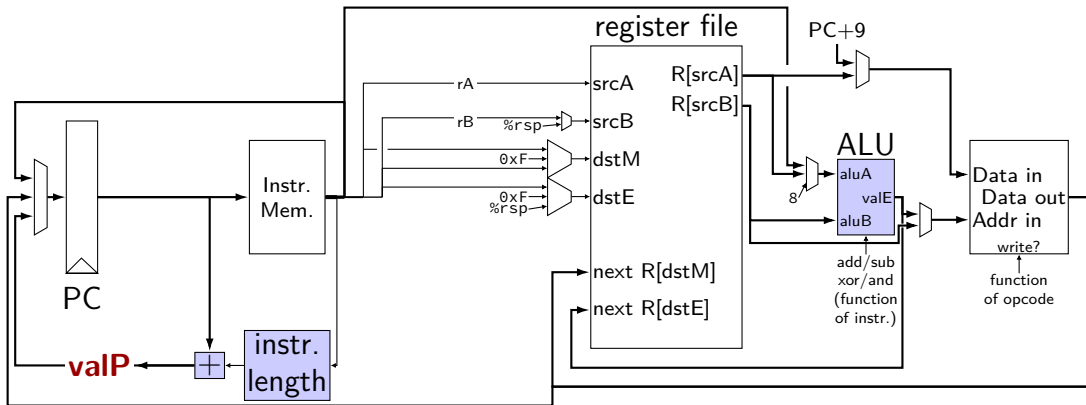


# ALUs

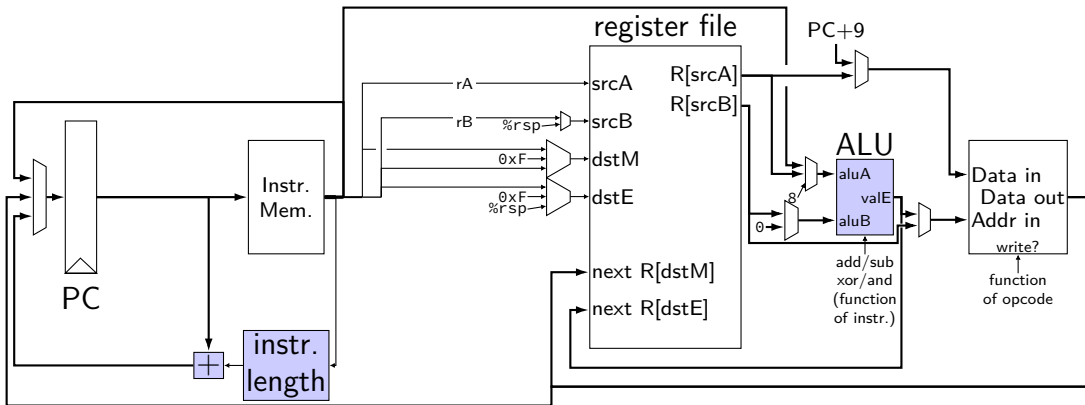


Operations needed:  
add — **addq**, addresses  
sub — **subq**  
xor — **xorq**  
and — **andq**  
more?

# SEQ circuit



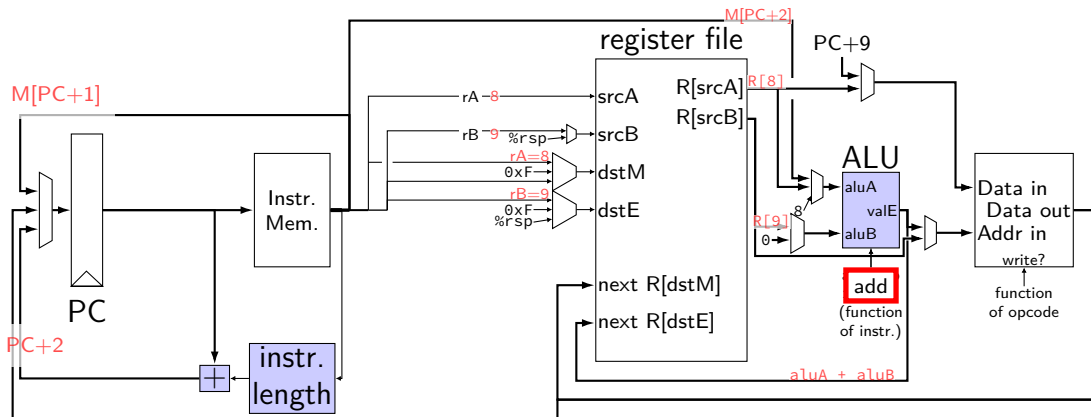
# circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

Exercise: what do they select when running `addq %r8, %r9`?

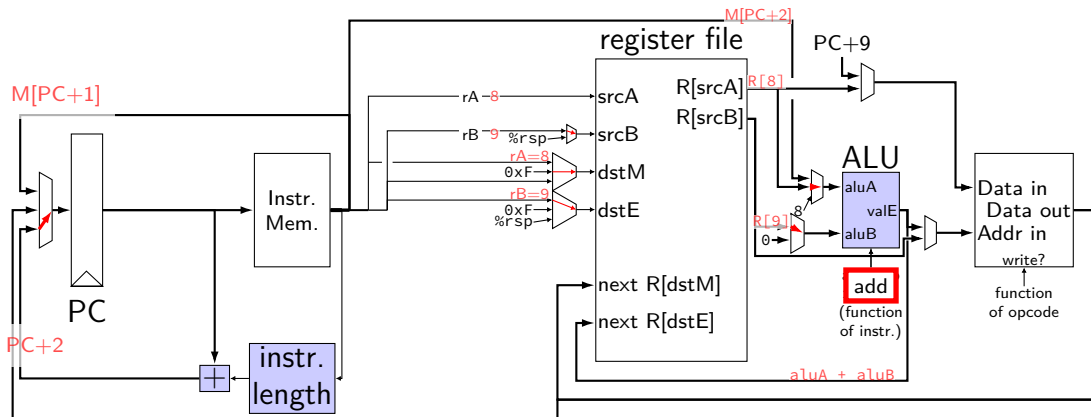
# circuit: setting MUXEs



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

Exercise: what do they select when running `addq %r8, %r9`?

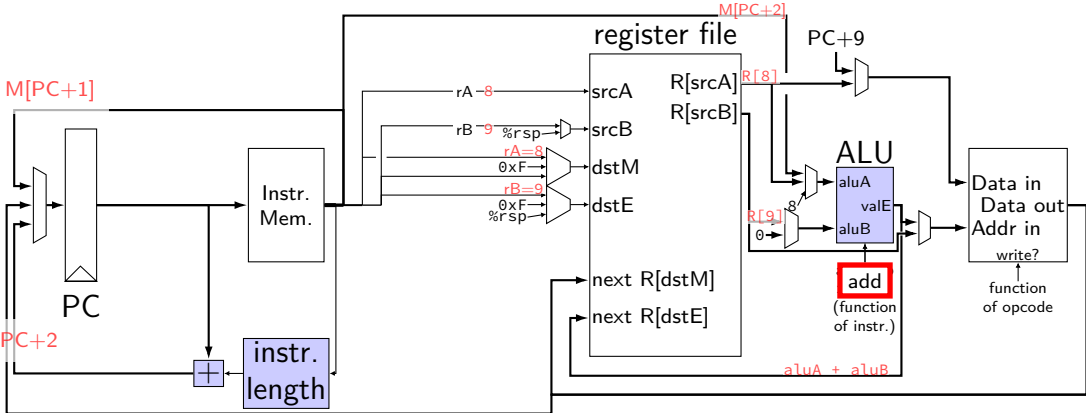
# circuit: setting MUXEs



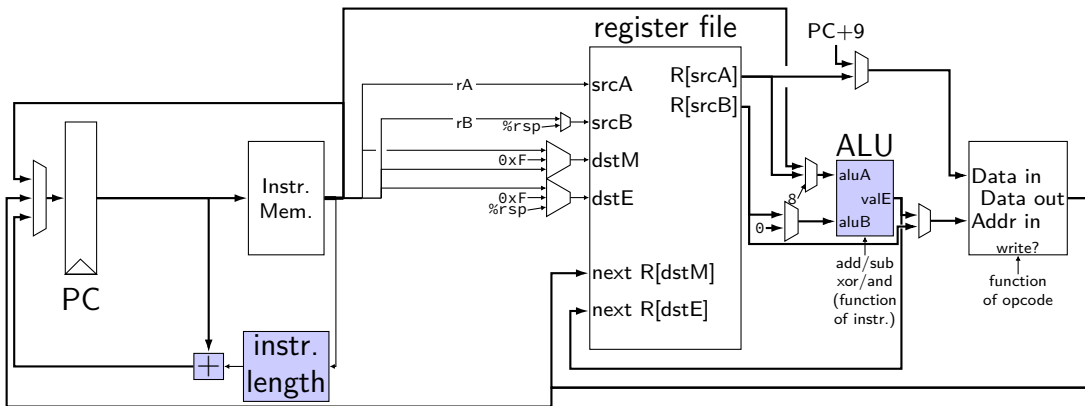
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

Exercise: what do they select when running `addq %r8, %r9`

# circuit: setting MUXEs



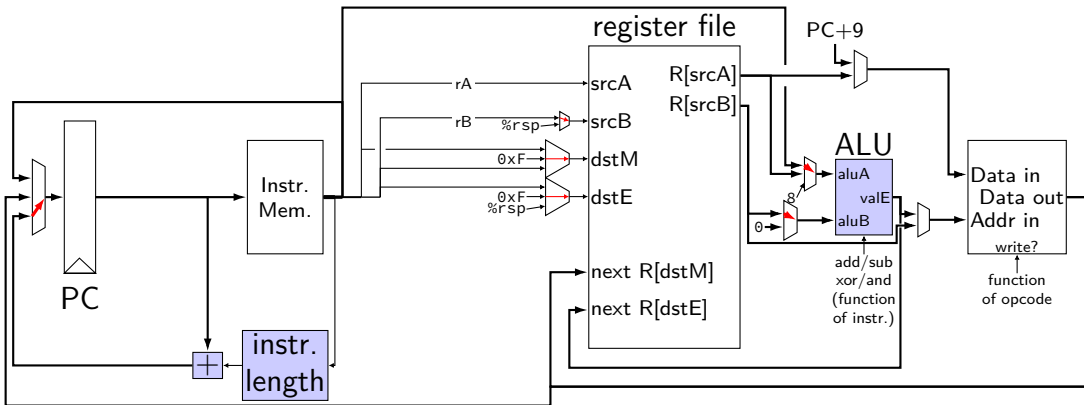
# circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

Exercise: what do they select for **rmmovq**?

# circuit: setting MUXes

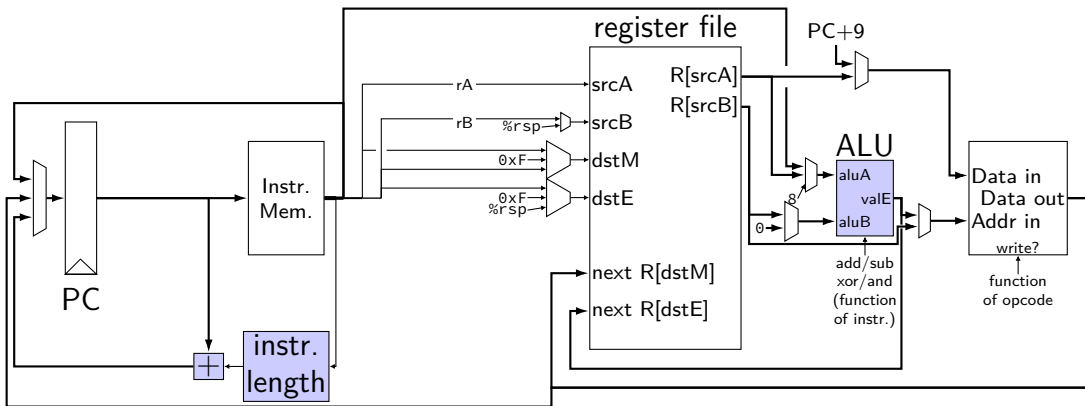


MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

Exercise: what do they select for **rmmovq**?



# circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

Exercise: what do they select for **call**?