

## Exam Review

1

## last time

hardware description language  
programming language that compiles to circuits

stages as conceptual division  
not the order things happen  
easier to figure out wiring stage-by-stage?

2

## on office hour locations

3

## on the homework

can use multiple statements and temporary variables  
only arithmetic shifts available

4

## on this week's quiz

5

6

## layers of abstraction

`x += y`

“Higher-level” language: C

`add %rbx, %rax`

Assembly: X86-64

60 03<sub>SIXTEEN</sub>

Machine code: Y86

???

Gates / Transistors / Wires / Registers

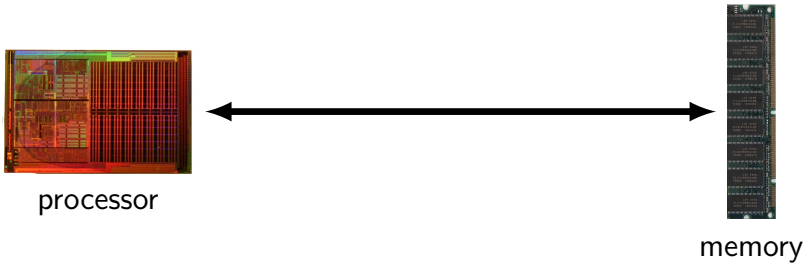
7

## interlude: powers of two

	...		...
$2^0$	1	$2^{11}$	2 048
$2^1$	2	$2^{12}$	4 096
$2^2$	4	$2^{13}$	8 192
$2^3$	8	$2^{14}$	16 384
$2^4$	16	$2^{15}$	32 768
$2^5$	32	$2^{16}$	65 536
$2^6$	64	...	...
$2^7$	128	$2^{20}$	1 048 576 <b>M</b> (or Mi)
$2^8$	256	...	...
$2^9$	512	$2^{30}$	1 073 741 824 <b>G</b> (or Gi)
$2^{10}$	1 024 <b>K</b> (or Ki)	$2^{31}$	2 147 483 648
		$2^{32}$	4 294 967 296
		...	...

8

# 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

# endianness

address	value
0xFFFFFFFF	0x14
0xFFFFFFFFE	0x45
0xFFFFFFFFD	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
0xFFFFFFFFE	0x45
0xFFFFFFFFD	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
0xFFFFFFFFE	0x45
0xFFFFFFFFD	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
0xFFFFFFFFE	0x45
0xFFFFFFFFD	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
0xFFFFFFFFE	0x45
0xFFFFFFFFD	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)

# 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;
}
    
```

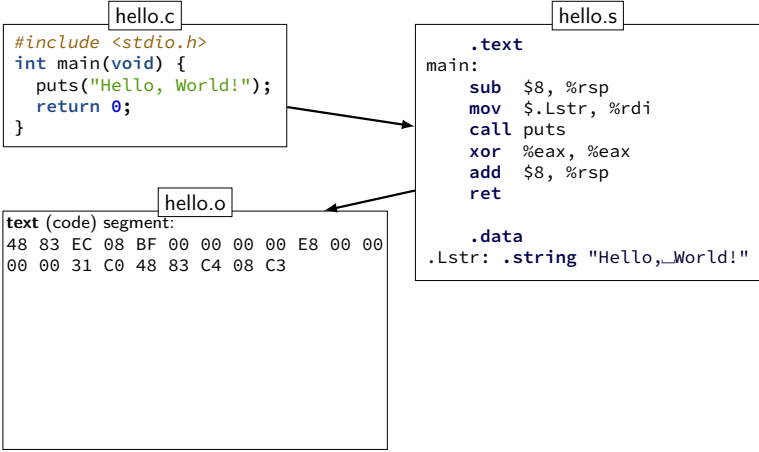
→

```

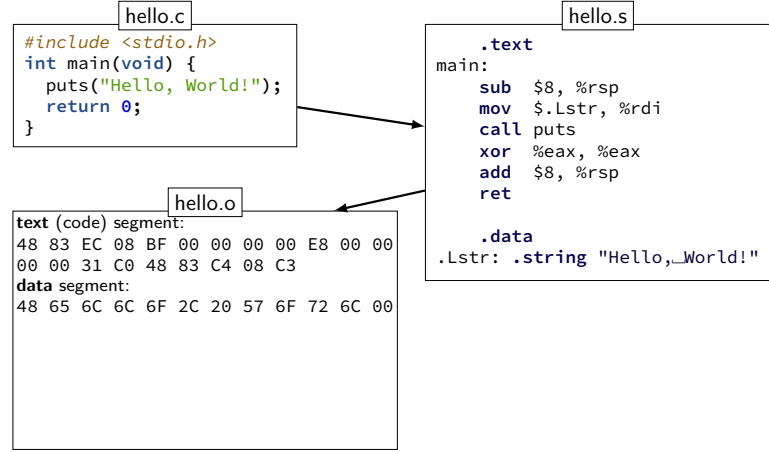
hello.s
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

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

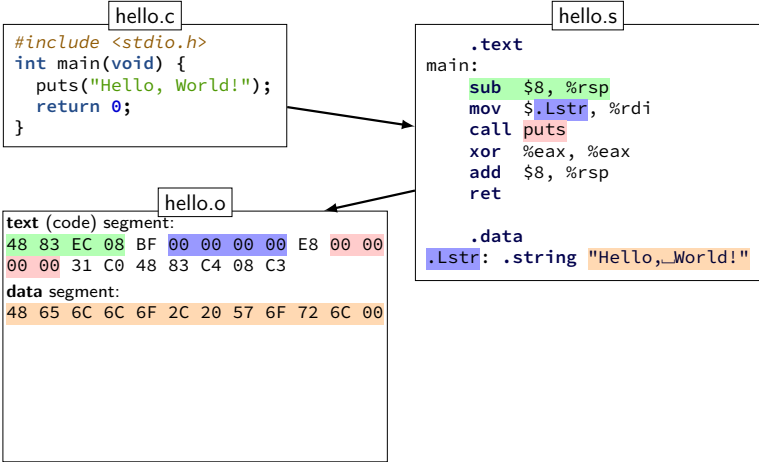
# what's in those files?



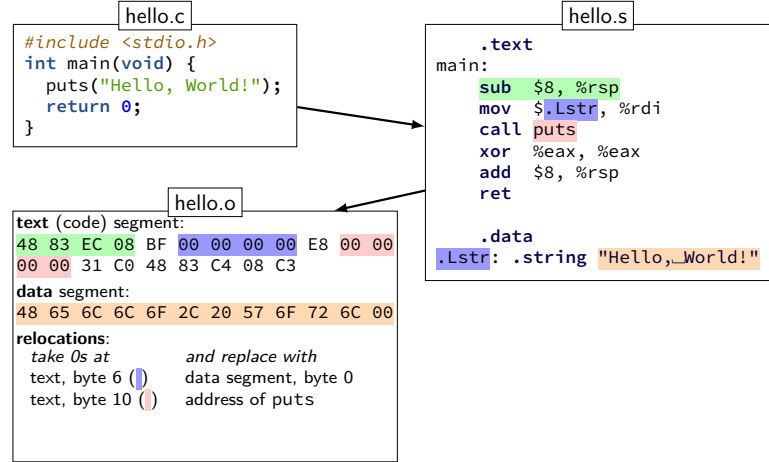
# what's in those files?



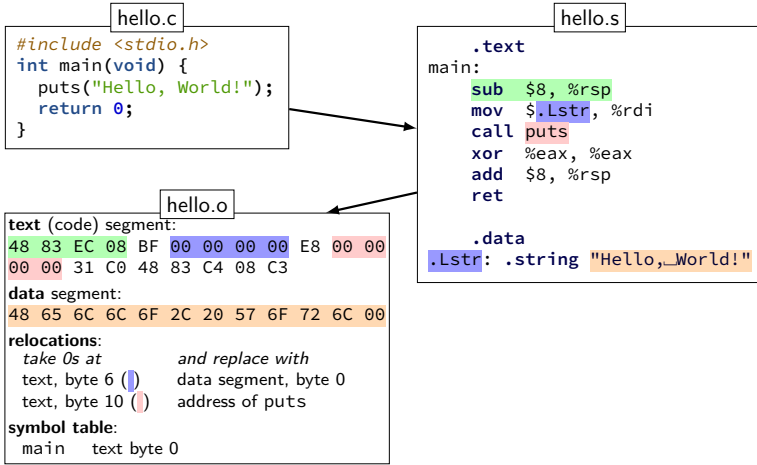
# what's in those files?



# what's in those files?

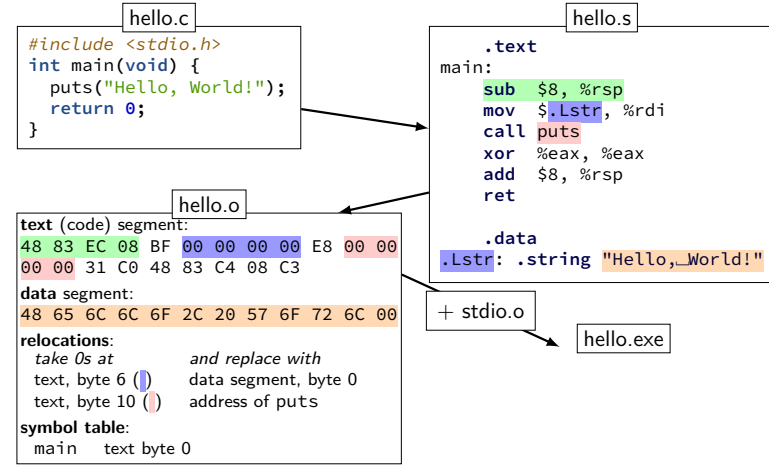


## what's in those files?



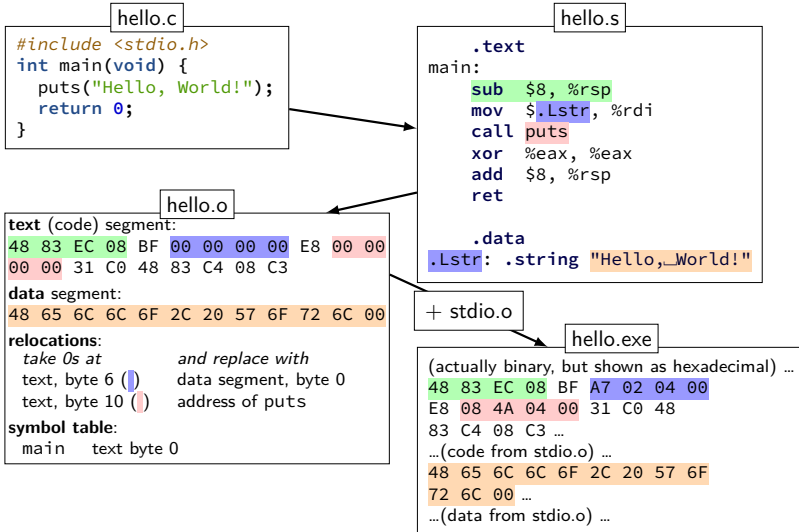
9

## what's in those files?



9

## what's in those files?



9

## hello.s

```

.section          .rodata.str1.1,"aMS",@progb-
.LC0:
    .string "Hello, World!"
    .text
    .globl main
main:
    subq    $8, %rsp
    movl   $.LC0, %edi
    call   puts
    movl   $0, %eax
    addq   $8, %rsp
    ret
    
```

10

# hello.o

hello.o: file format elf64-x86-64

## SYMBOL TABLE:

```
0000000000000000 g      F .text 0000000000000018 ma
0000000000000000          *UND* 0000000000000000 put
```

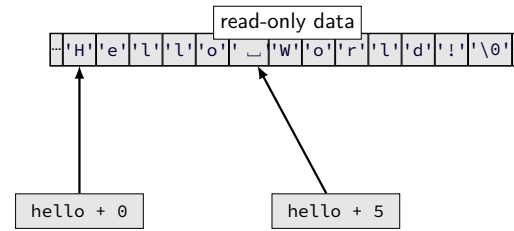
## RELOCATION RECORDS FOR [ .text ]:

OFFSET	TYPE	VALUE
0000000000000005	R_X86_64_32	.rodata.str1.1
000000000000000a	R_X86_64_PC32	puts-0x0000000000000000

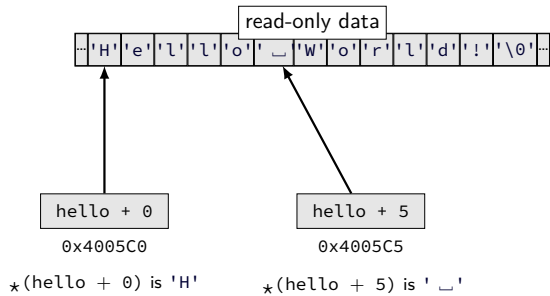
## Contents of section .text:

```
0000 4883ec08 bf000000 00e80000 0000b800 H.....
0010 00000048 82c408c2
```

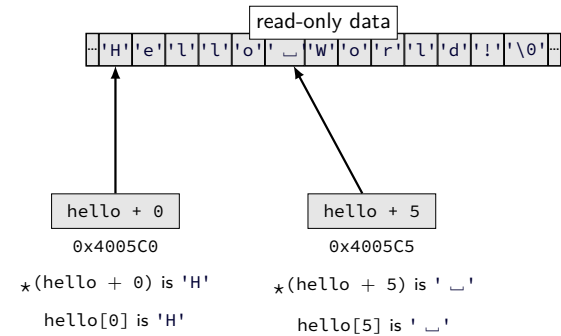
# pointer arithmetic



# pointer arithmetic

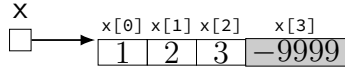


# pointer arithmetic

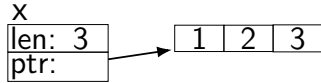


## some lists

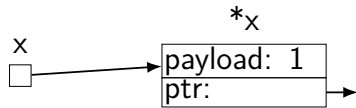
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```



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



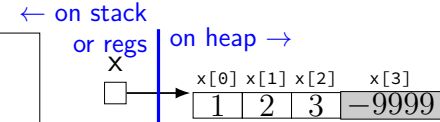
```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```



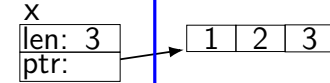
13

## some lists

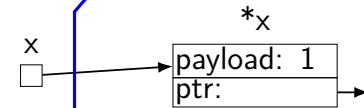
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```



```
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));
...
```



13

## 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

14

## 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

15



## 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

15

## 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

15

## 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

15

## 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

15

## 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

16

## 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

17

## 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
```

18

## 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
  ...
```

19

## 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

```

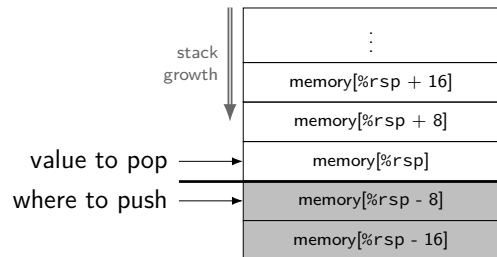
## push/pop

```

pushq %rbx
  %rsp ← %rsp - 8
  memory[%rsp] ← %rbx

popq %rbx
  %rbx ← memory[%rsp]
  %rsp ← %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					
mrmovq 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								
rrmovq/cmovCC <i>rA</i> , <i>rB</i>	2	cc	<i>rA</i>	<i>rB</i>						
irmovq <i>V</i> , <i>rB</i>	3	0	F	<i>rB</i>	V					
rmmovq <i>rA</i> , <i>D(rB)</i>	4	0	<i>rA</i>	<i>rB</i>	D					
mrmovq <i>D(rB)</i> , <i>rA</i>	5	0	<i>rA</i>	<i>rB</i>	D					
<i>OPq</i> <i>rA</i> , <i>rB</i>	6	fn	<i>rA</i>	<i>rB</i>						
<i>jCC Dest</i>	7	cc	Dest							
call <i>Dest</i>	8	0	Dest							
ret	9	0								
pushq <i>rA</i>	A	0	<i>rA</i>	F						
popq <i>rA</i>	B	0	<i>rA</i>	F						

0	add
1	sub
2	and
3	xor

24

## Registers: *rA*, *rB*

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

25

## Immediates: *V*, *D*, *Dest*

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

26

## Immediates: *V*, *D*, *Dest*

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

26

## bitwise strategies

use paper, find subproblems, etc.

mask and shift

$(x \& 0xF0) \gg 4$

factor/distribute

$(x \& 1) \mid (y \& 1) == (x \mid y) \& 1$

divide and conquer

common subexpression elimination

$\text{return } ((\sim!x) \& y) \mid ((\sim!x) \& z)$

becomes

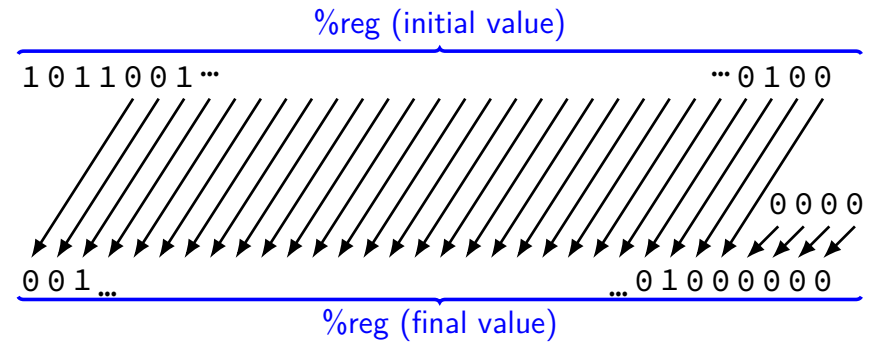
$d = !x; \text{return } ((\sim!d) \& y) \mid ((\sim!d) \& z)$

27

## shift left

x86 instruction: **shl** — shift left

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

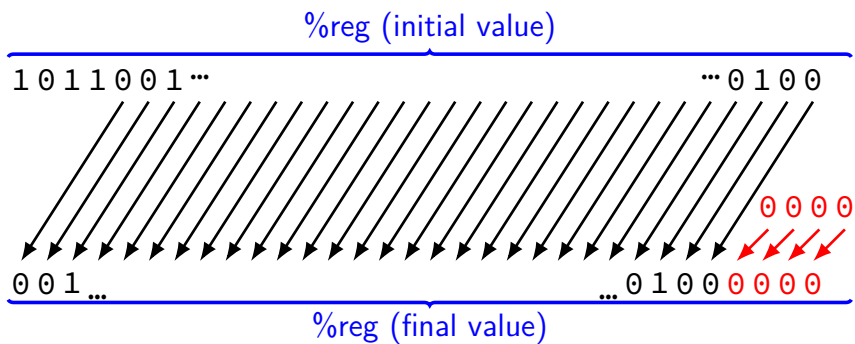


28

## shift left

x86 instruction: **shl** — shift left

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



28

## left shift in math

$1 \ll 0 == 1$                       0000 0001  
 $1 \ll 1 == 2$                       0000 0010  
 $1 \ll 2 == 4$                       0000 0100

$10 \ll 0 == 10$                     0000 1010  
 $10 \ll 1 == 20$                     0001 0100  
 $10 \ll 2 == 40$                     0010 1000

29

## left shift in math

$1 \ll 0 == 1$       0000 0001  
 $1 \ll 1 == 2$       0000 0010  
 $1 \ll 2 == 4$       0000 0100

$10 \ll 0 == 10$      0000 1010  
 $10 \ll 1 == 20$      0001 0100  
 $10 \ll 2 == 40$      0010 1000

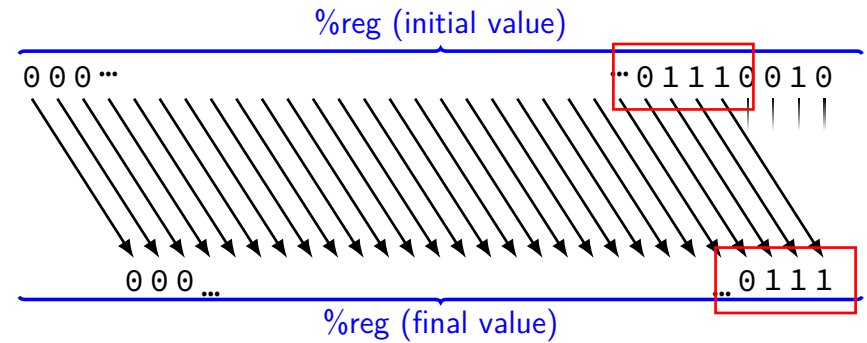
$$x \ll y = x \times 2^y$$

29

## logical right shift

x86 instruction: **shr** — logical shift right

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

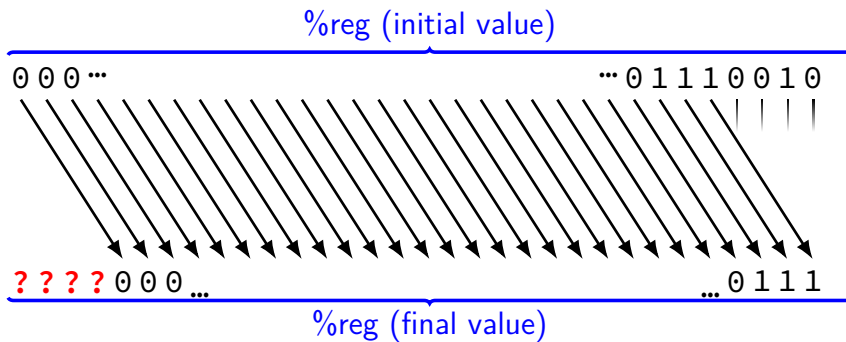


30

## logical right shift

x86 instruction: **shr** — logical shift right

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

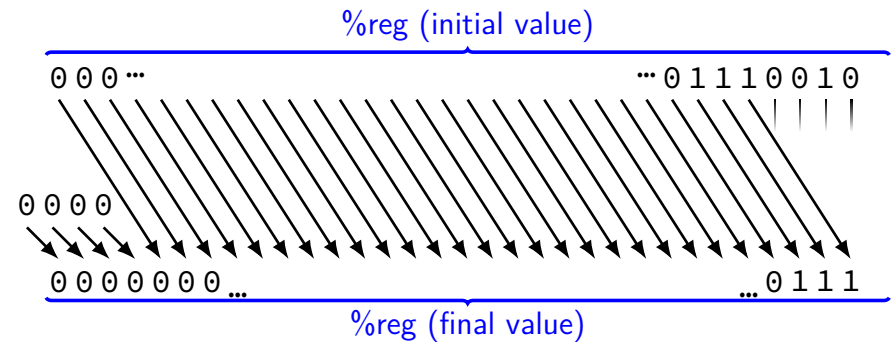


30

## logical right shift

x86 instruction: **shr** — logical shift right

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



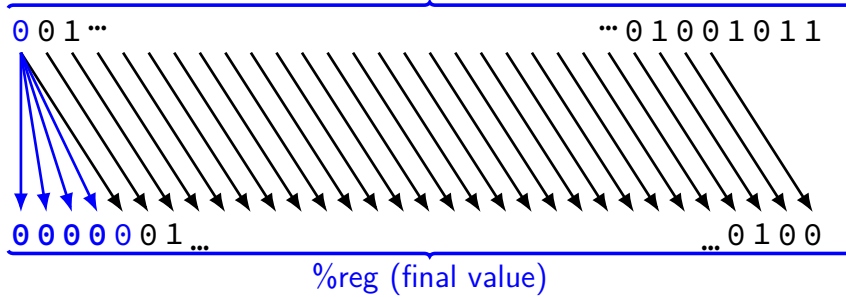
30

## arithmetic right shift

x86 instruction: **sar** — arithmetic shift right

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

%reg (initial value)



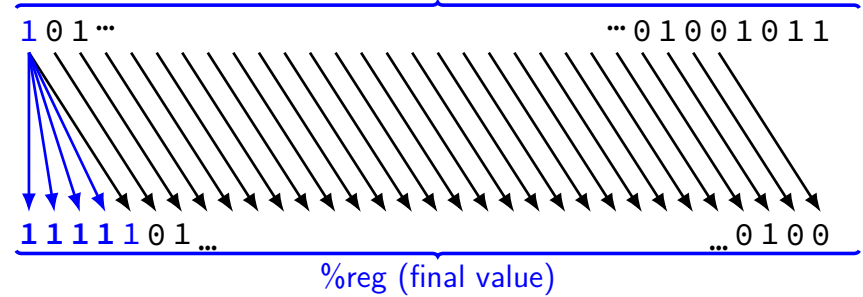
31

## arithmetic right shift

x86 instruction: **sar** — arithmetic shift right

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

%reg (initial value)



31

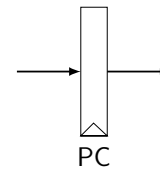
## right shift in C

```
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
}
```

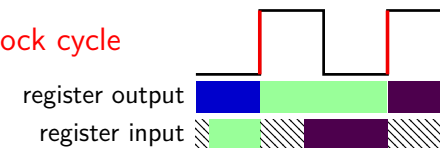
<pre>shift_signed:     movl %edi, %eax     sarl \$5, %eax     ret</pre>	<pre>shift_unsigned:     movl %edi, %eax     shrl \$5, %eax     ret</pre>
---	---

32

## registers

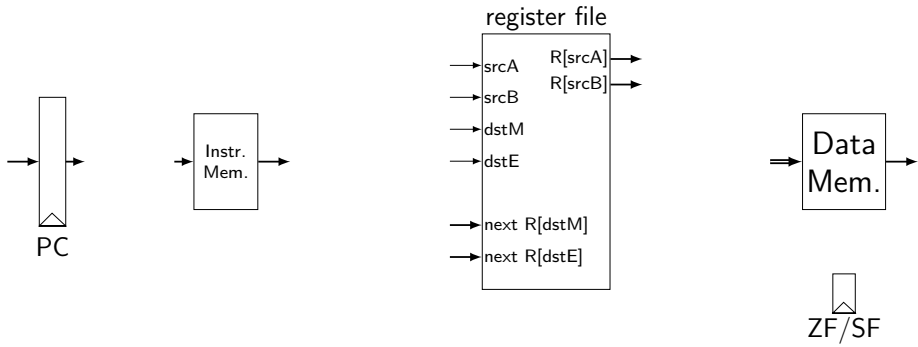


updates every **clock cycle**



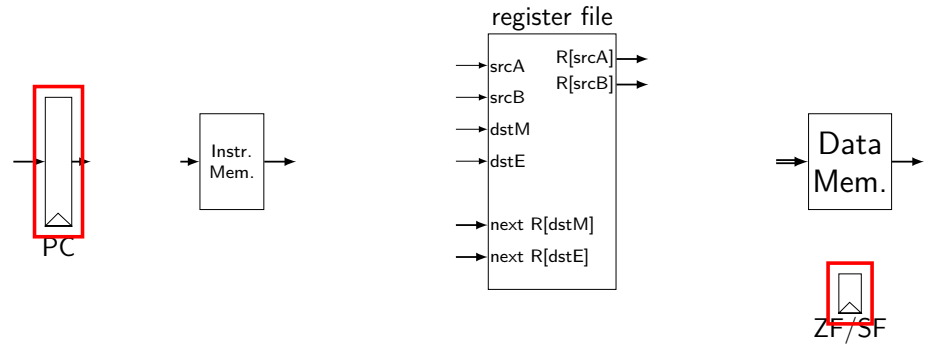
33

## state in Y86-64



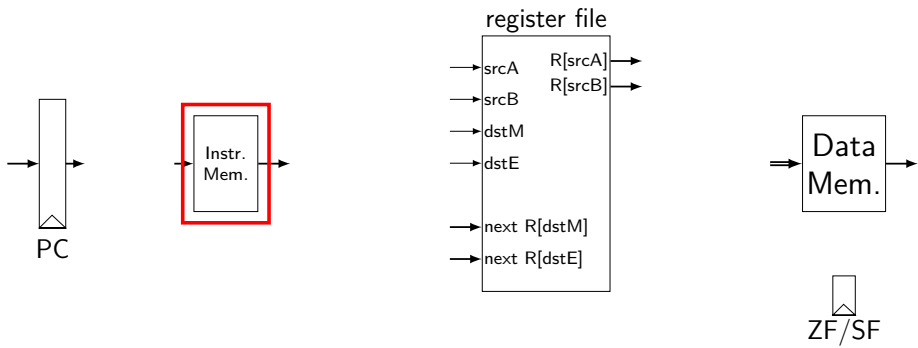
34

## state in Y86-64



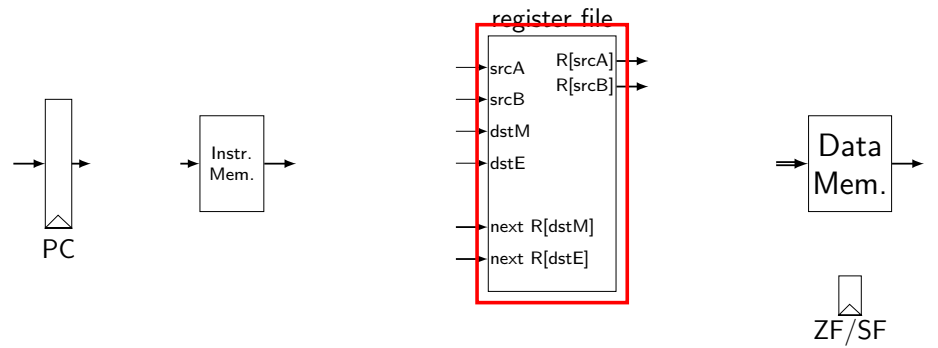
34

## state in Y86-64



34

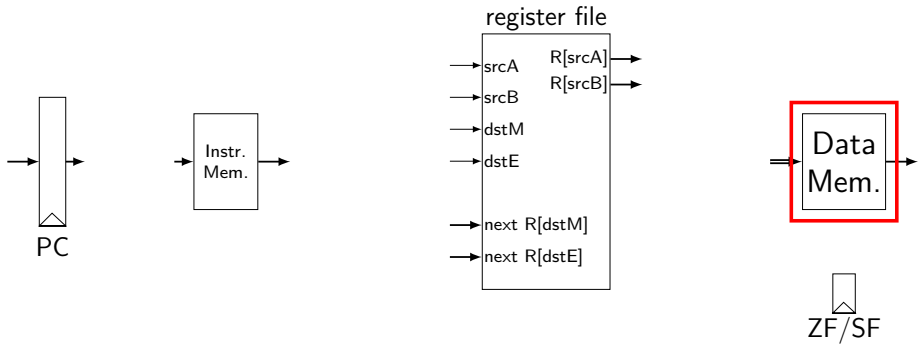
## state in Y86-64



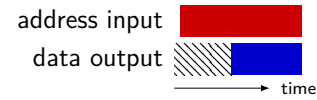
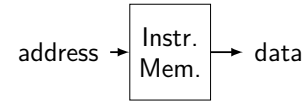
34



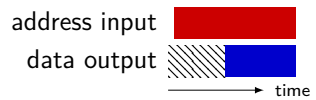
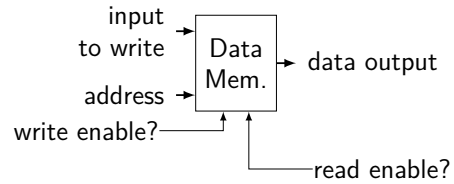
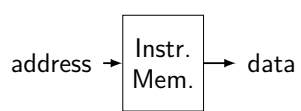
# state in Y86-64



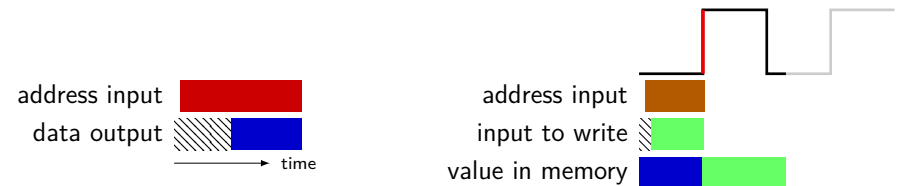
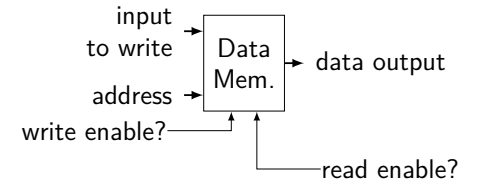
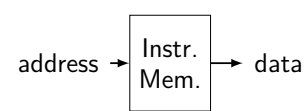
# memories



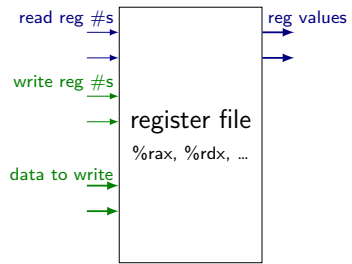
# memories



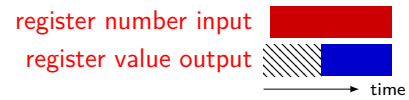
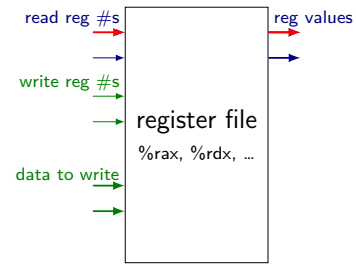
# memories



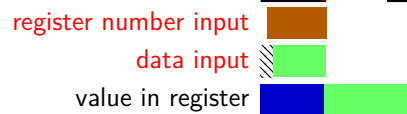
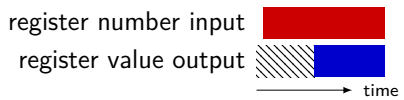
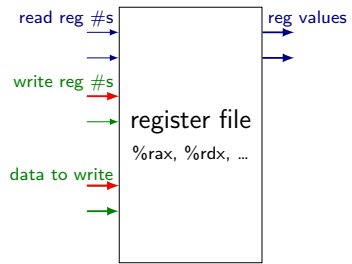
# register file



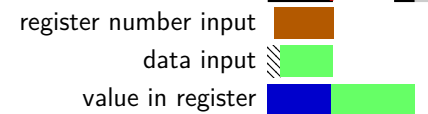
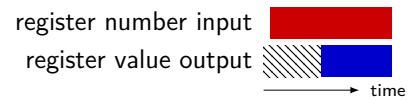
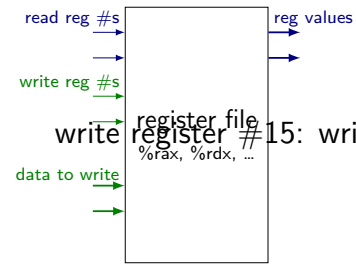
# register file



# register file

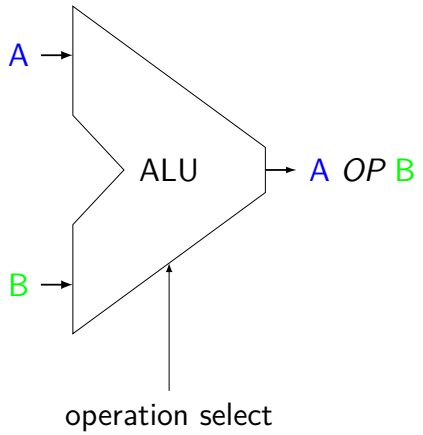


# register file



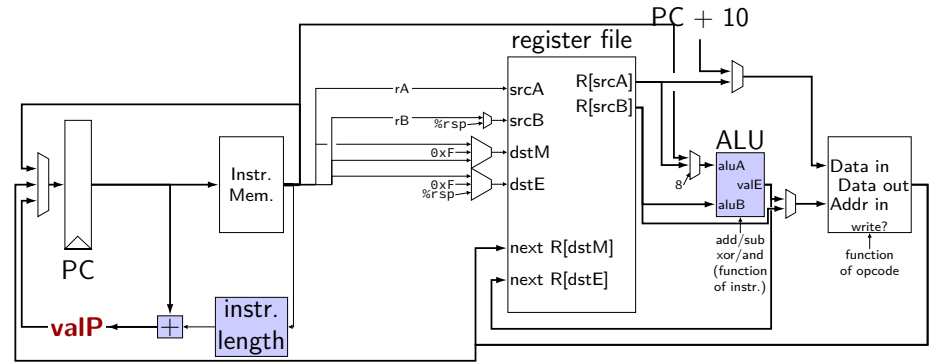
write register #15: write is ignored read register #15: value is always

# ALUs

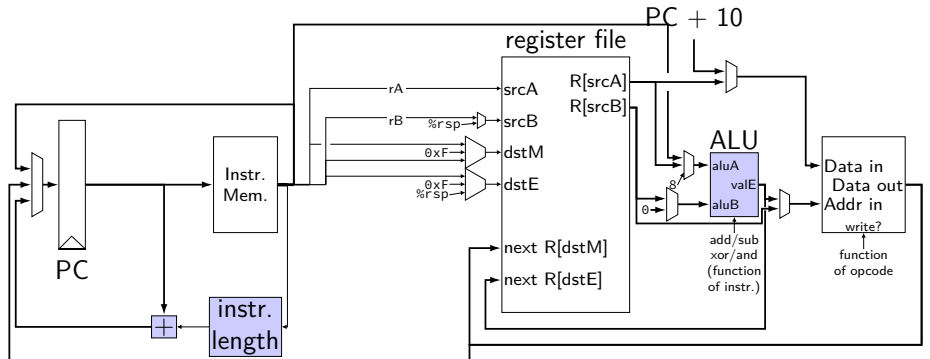


Operations needed:  
 add — **addq**, addresses  
 sub — **subq**  
 xor — **xorq**  
 and — **andq**  
 more?

# SEQ circuit

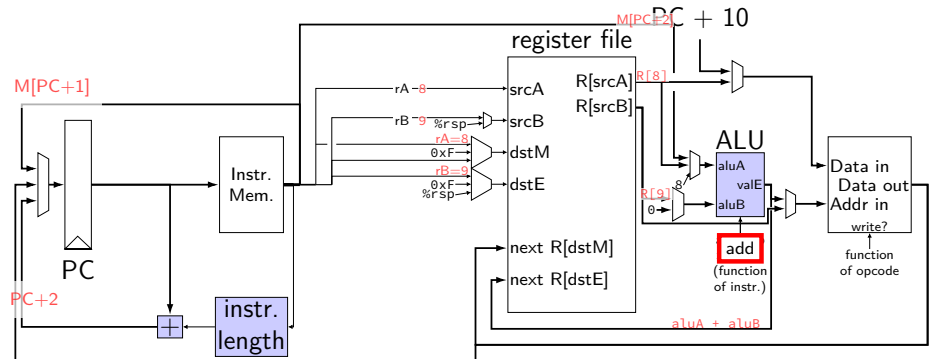


# circuit: setting MUXes



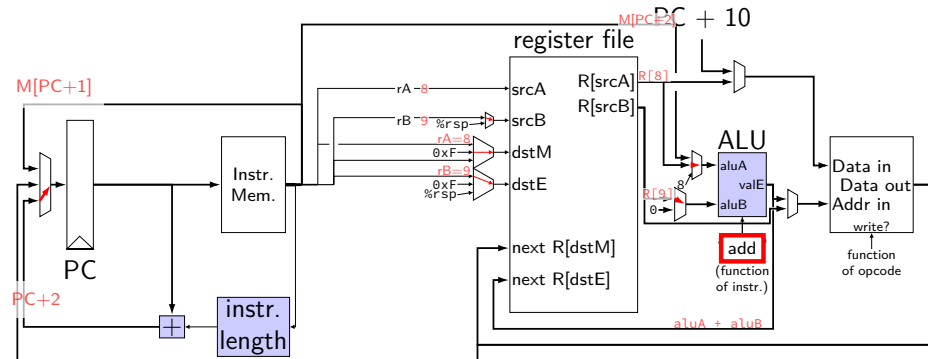
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn  
 Exercise: what do they select when running **addq** %r8, %r9?

# circuit: setting MUXes



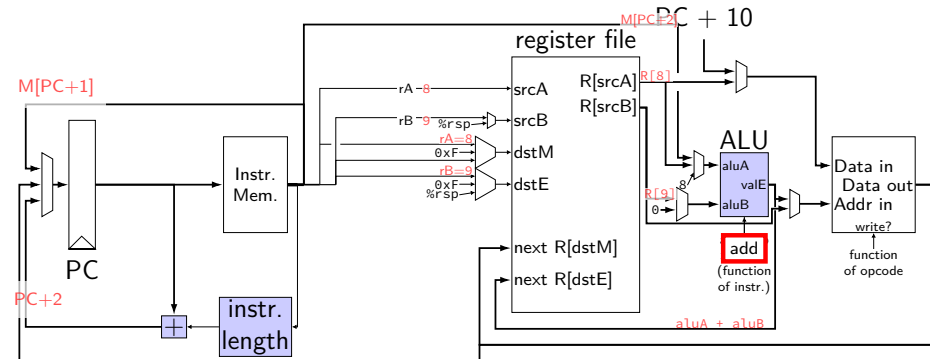
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn  
 Exercise: what do they select when running **addq** %r8, %r9?

## circuit: setting MUXes

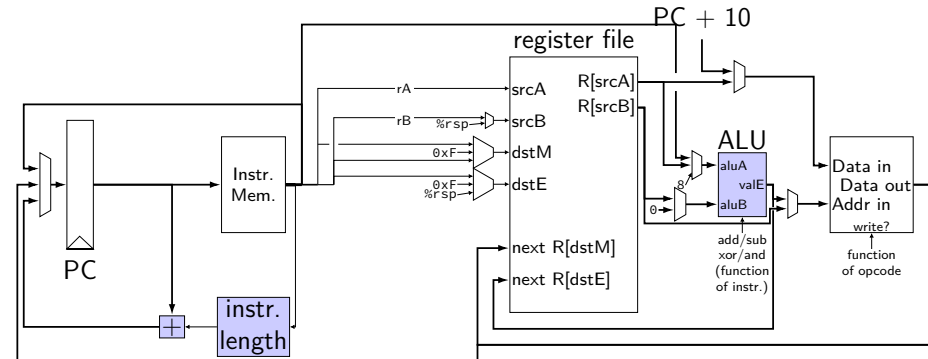


MUXes — PC, dstM, dstE, aluA, aluB, dmemIn  
 Exercise: what do they select when running `addq %r8, %r9`?

## circuit: setting MUXes

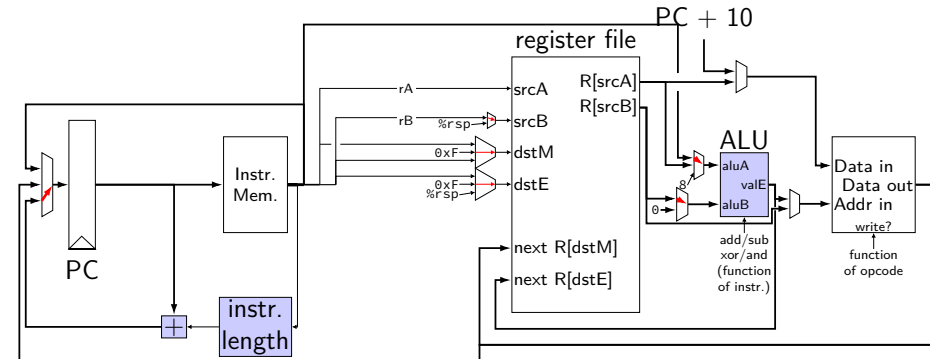


## circuit: setting MUXes



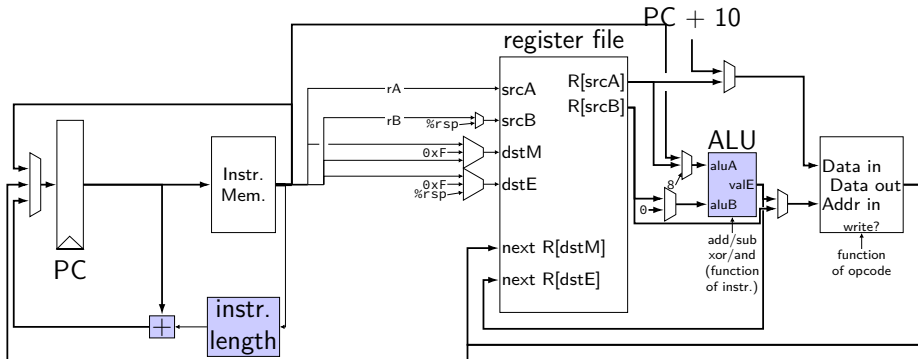
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn  
 Exercise: what do they select for `rmmovq`?

## 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**?