

bitwise (finish) / SEQ part 1

# Changelog

Changes made in this version not seen in first lecture:

14 September 2017: slide 16-17: the x86 arithmetic shift instruction is sar, not sra

# last time

## bitwise strategies:

construct/apply mask = number w/1s to mark important bits

AND/& — keep only marked

OR/| — set marked

XOR/^ — flipped marked

shift bits to desired positions

divide and conquer — find subproblems

## bitwise-like parallelism —

multiple copies of operation in different part of number

example: OR all pairs of bits, not just last and second-to-last

## 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)
```



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



## aside: homework

random types of lists (of shorts)

sentinel-terminated array — special value at end

range — structure of pointer + size

linked list

convert first to second type

append second type to second type

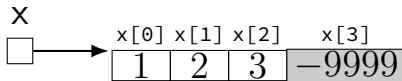
*modify* the list **pointed to** by first argument

remove\_if\_equal *all* elements equal to a value from second type

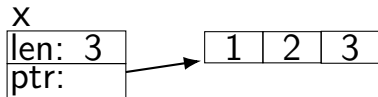
*modify* the list **pointed to** by first argument

# some lists

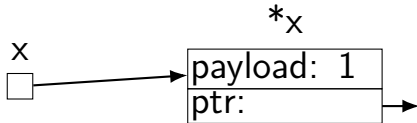
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```



```
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;
...
```

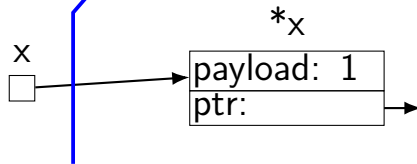
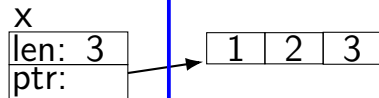
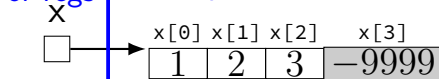
```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

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

← on stack

or regs

on heap →



# multiplication

$$10 \ll 2 == 10 * 4 = 10 + 10 + 10 + 10$$

$$10 \ll 3 == 10 * 8$$

$$(10 \ll 3) + (10 \ll 2) == 10 * 12$$

$$-10 \ll 2 == -10 * 4 == (-10) + (-10) + (-10) + (-10)$$

$$-10 \ll 3 == -10 * 8$$

$$(-10 \ll 3) + (-10 \ll 2) == -10 * 12$$

## more division

```
int divide_by_32(int x) {  
    return x / 32;  
}
```

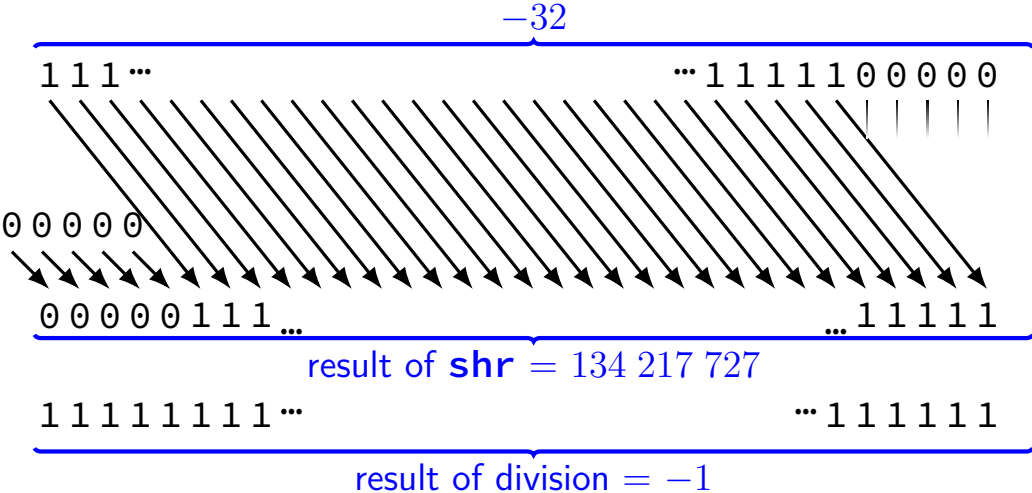
*// INCORRECT generated code*

```
divide_by_32:  
    shrl $5, %edi // ← this is WRONG  
    mov %edi, %eax
```

example input with wrong output:  $-32$

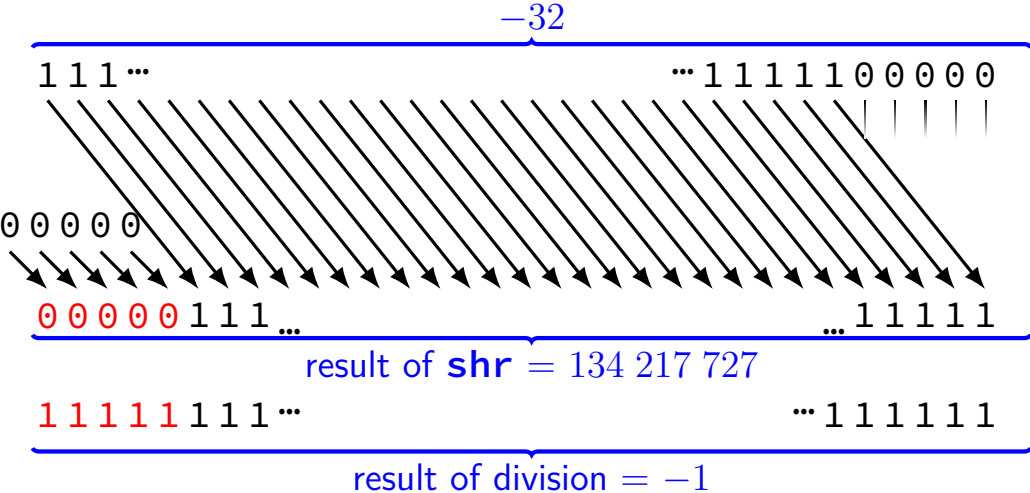
exercise: what does this assembly return? what is the correct result?

# wrong division





# wrong division



# dividing negative by two

start with  $-x$

flip all bits and add one to get  $+x$

right shift by one to get  $+x/2$

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

# dividing negative by two

start with  $-x$

flip all bits and add one to get  $+x$

right shift by one to get  $+x/2$

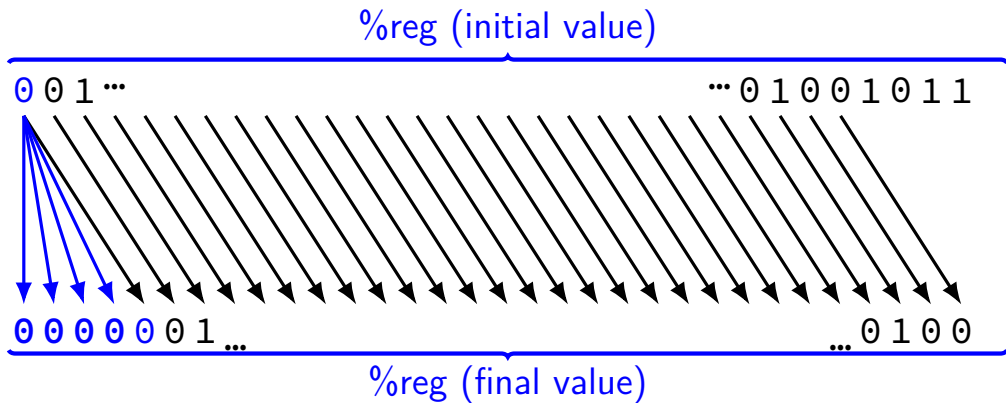
flip all bits and add one to get  $-x/2$

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

# arithmetic right shift

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

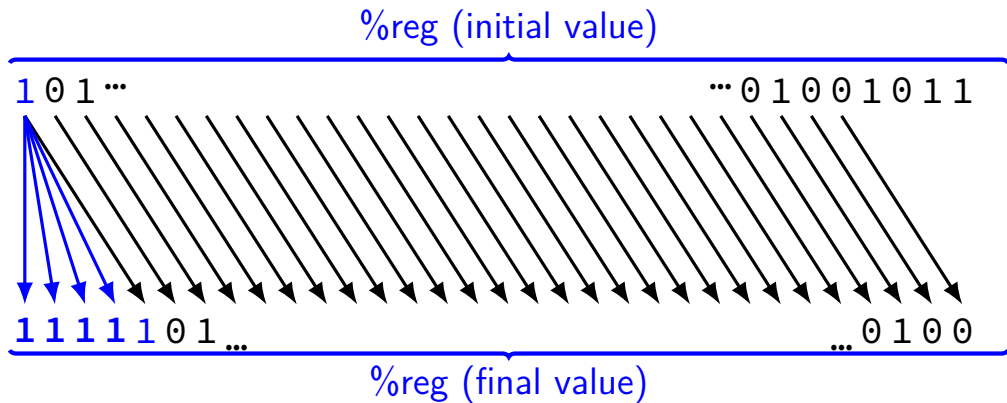
**sar** *\$amount*, %reg (or variable: **sar** %cl, %reg)



# arithmetic right shift

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

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



## right shift in C

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

---

shift\_signed:

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

shift\_unsigned:

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

# dividing negative by two

start with  $-x$

flip all bits and add one to get  $+x$

right shift by one to get  $+x/2$

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

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

# divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: “bias” adjustments — described in textbook



# divide with proper rounding

C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: “bias” adjustments — described in textbook

```
divide_by_32: // GCC generated code  
    leal    31(%rdi), %eax // eax ← edi + 31  
    testl   %edi, %edi     // set cond. codes based on %edi  
    cmovns %edi, %eax      // if (edi sign bit = 0) eax ← edi  
    sarl    $5, %eax       // arithmetic shift  
    ret
```

# standards and shifts in C

signed right shift is **implementation-defined**

compilers can choose which type of shift to do  
all compilers I know of — arithmetic (copy sign bit)

unsigned right shift is always logical (fill with zeroes)

shift amount  $\geq$  width of type: undefined behavior

x86 assembly: only uses lower bits of shift amount

## miscellaneous bit manipulation

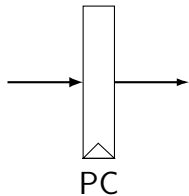
common bit manipulation instructions are not in C:

rotate (x86: `ror`, `rol`) — like shift, but wrap around

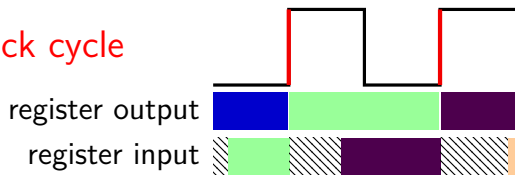
index of first/last bit set (x86: `bsf`, `bsr`)

population count (some x86: `popcnt`) — number of bits set

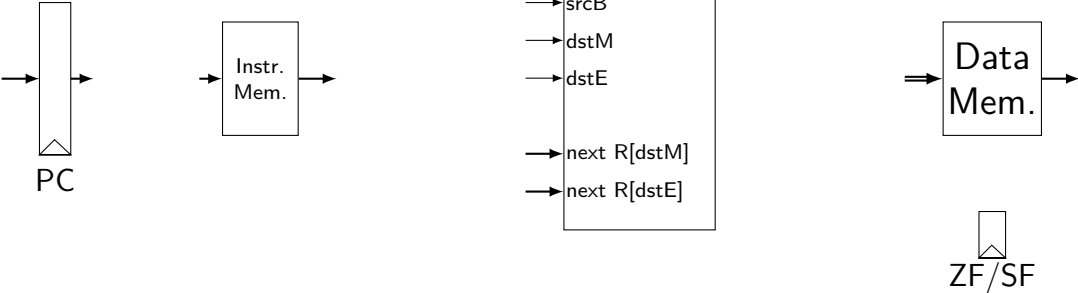
# registers



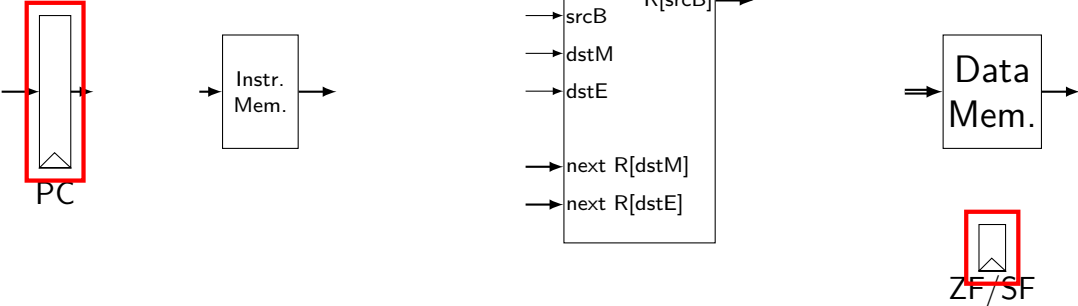
updates every **clock cycle**



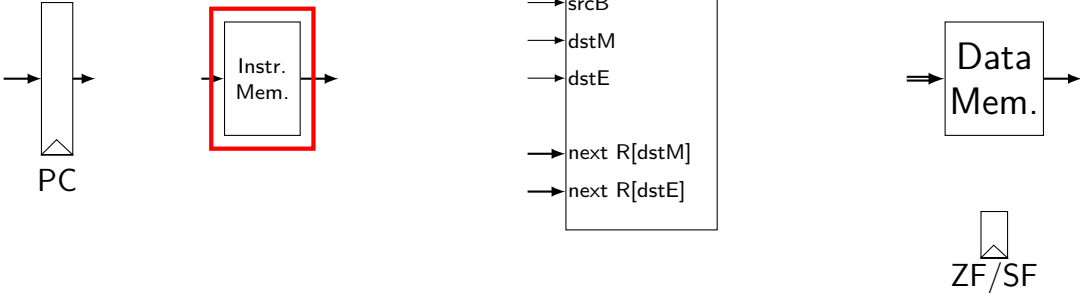
# state in Y86-64



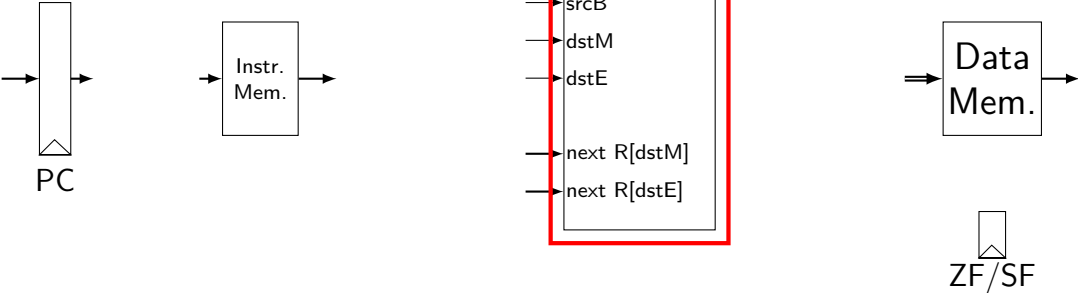
# state in Y86-64



# state in Y86-64

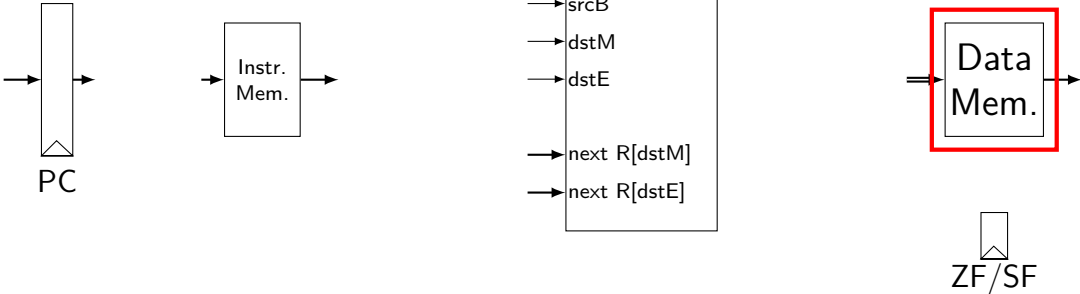


# state in Y86-64

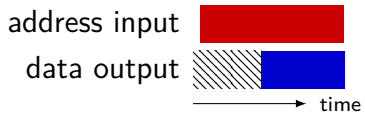
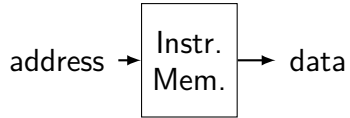




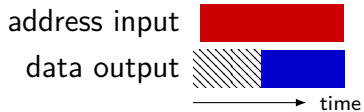
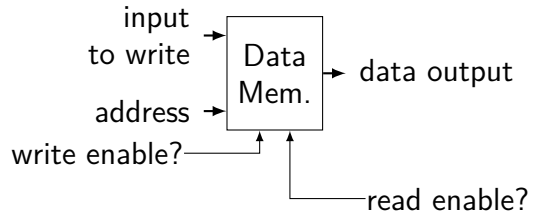
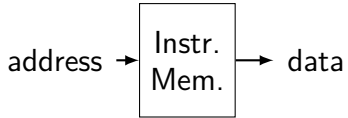
# state in Y86-64



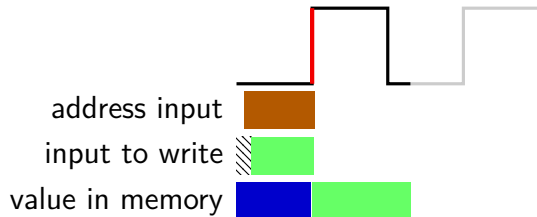
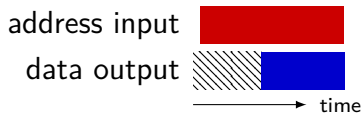
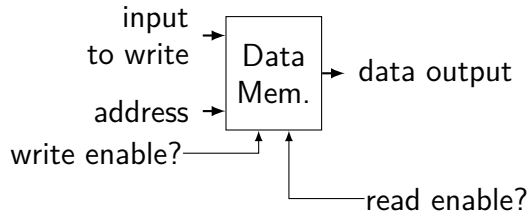
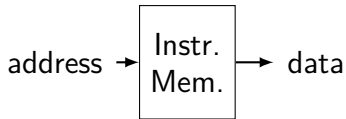
# memories



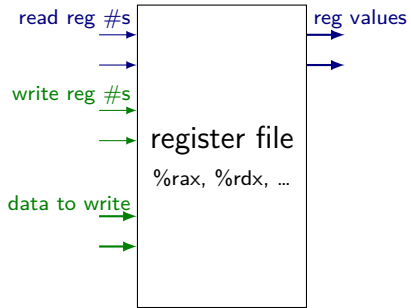
# memories



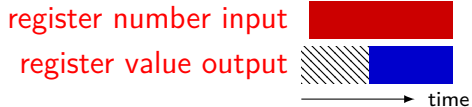
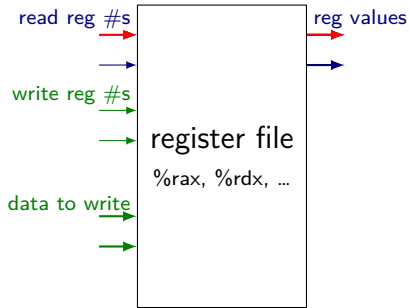
# memories



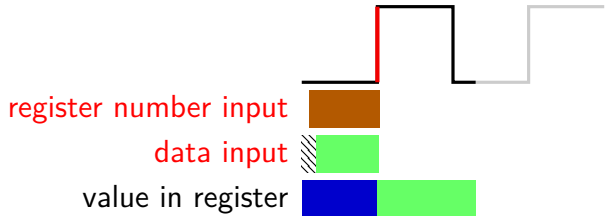
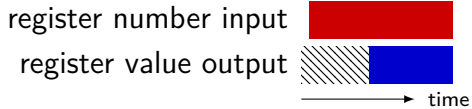
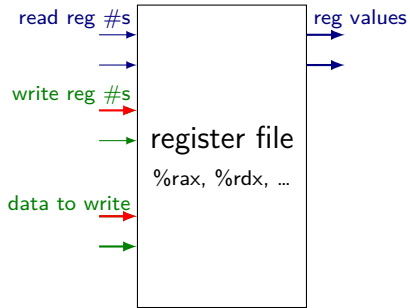
# register file



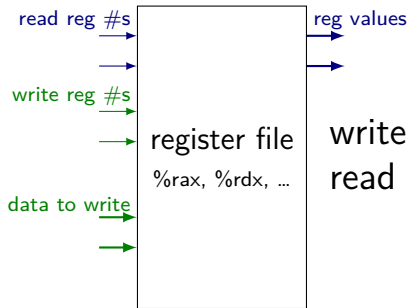
# register file



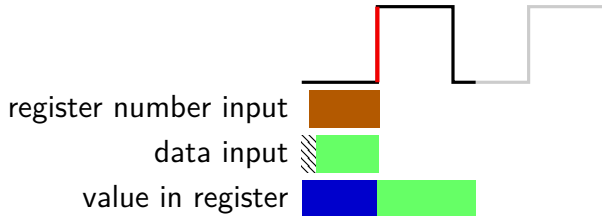
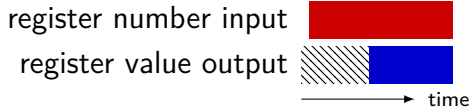
# register file



# register file

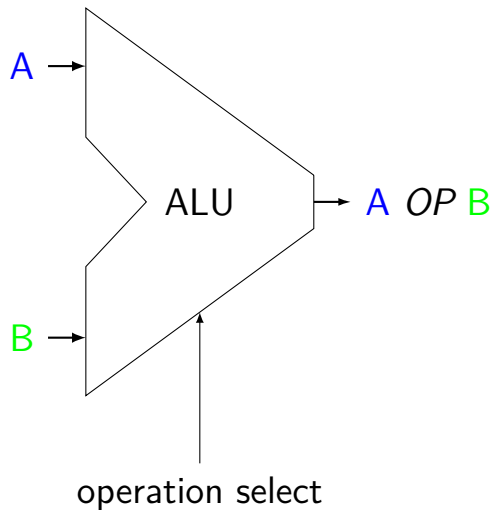


write register #15: write is ignored  
read register #15: value is always 0





# ALUs



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

# simple ISA 1: addq

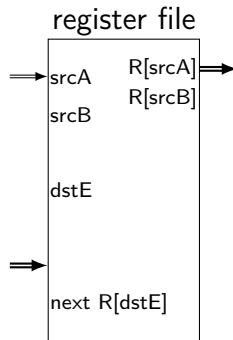
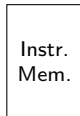
addq %rXX, %rYY

encoding: %rXX %rYY (two 4-bit register #s)

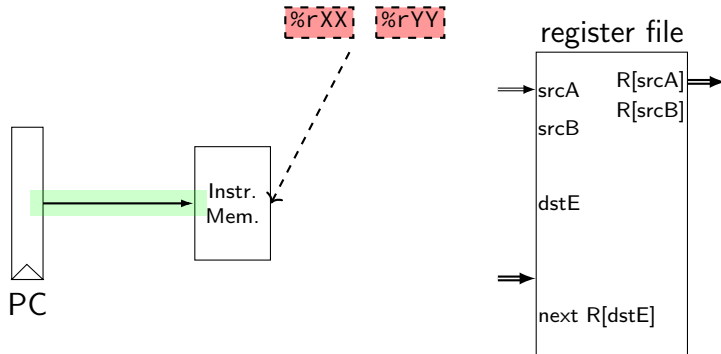
1 byte instructions, no opcode

no other instructions

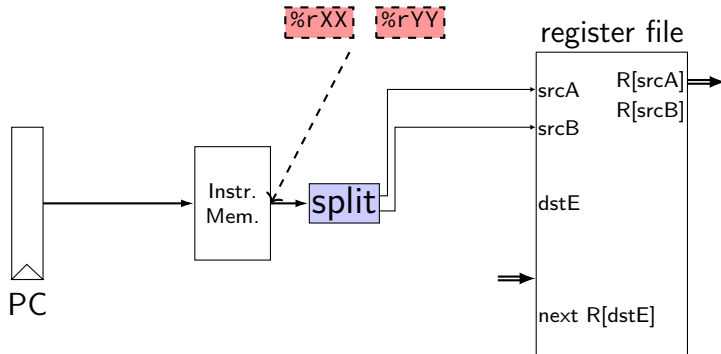
# addq CPU



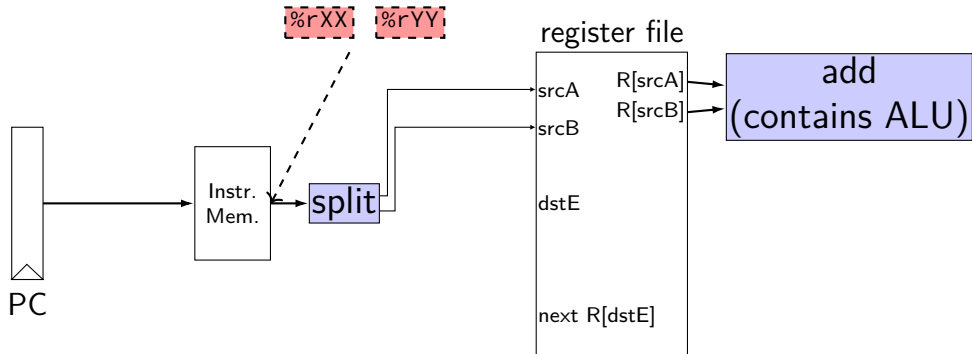
# addq CPU



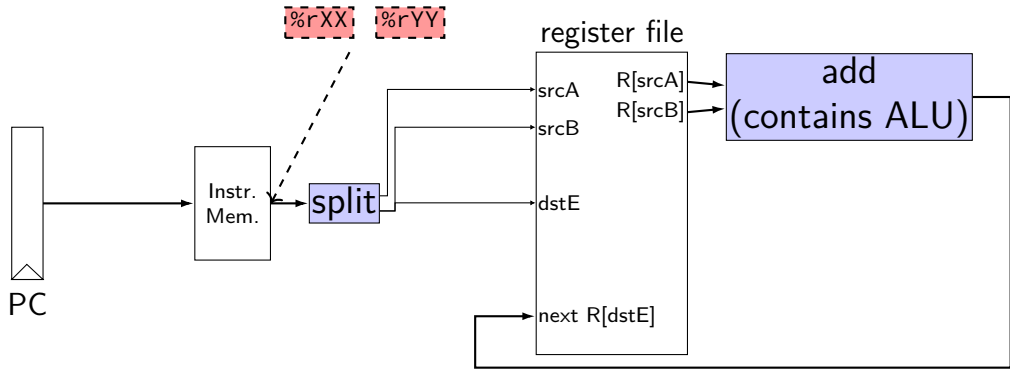
# addq CPU



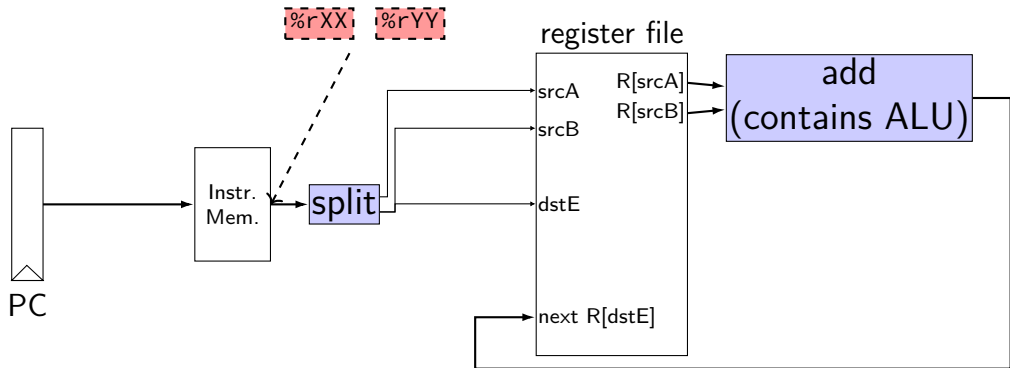
# addq CPU



# addq CPU



# addq CPU



```
/* 0x00: */ addq %rax, %rdx
```

```
/* 0x01: */ addq %rbx, %rdx
```

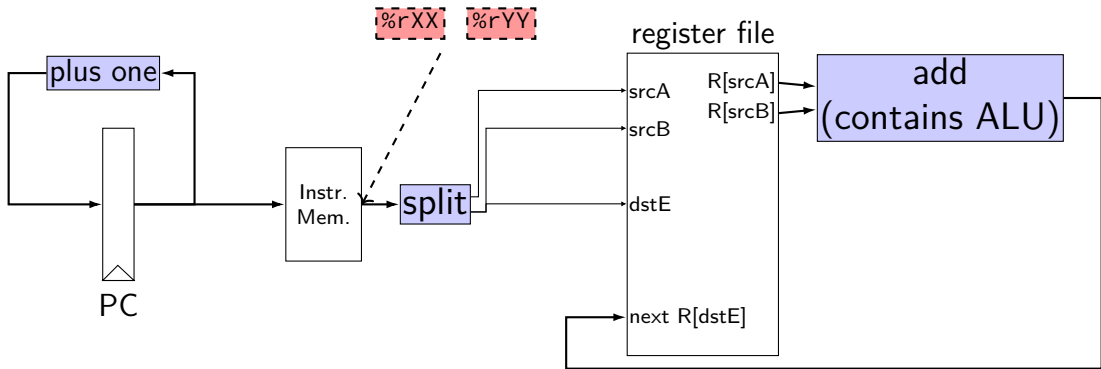
initially: PC = 0x00, rax = 1, rbx = 2, rdx = 3

after cycle 1: PC = ????, rax = 1, rbx = 2, rdx = 4

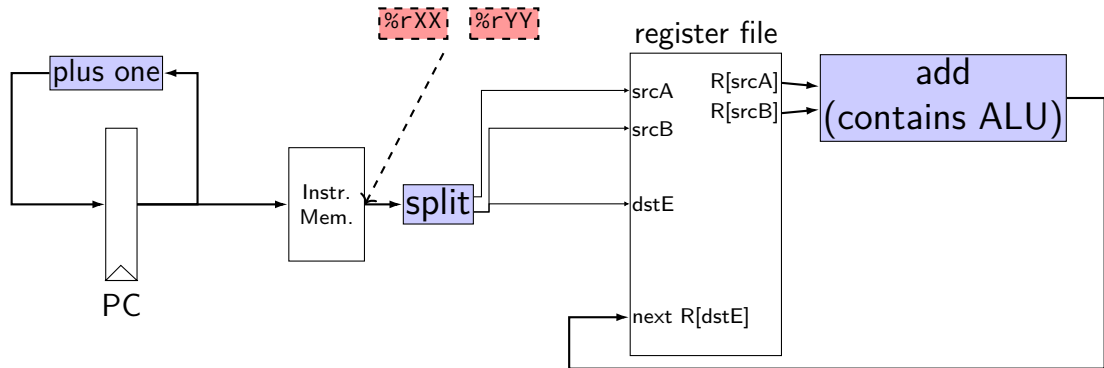
after cycle 2: PC = ????, rax = ??, rbx = ??, rdx = ??



# addq CPU



# addq CPU



```
/* 0x00: */ addq %rax, %rdx
```

```
/* 0x01: */ addq %rbx, %rdx
```

initially: PC = 0x00, rax = 1, rbx = 2, rdx = 3

after cycle 1: PC = 0x01, rax = 1, rbx = 2, rdx = 4

after cycle 2: PC = 0x02, rax = 1, rbx = 2, rdx = 6

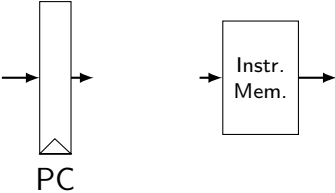
## Simple ISA 2: jmp

jmp label

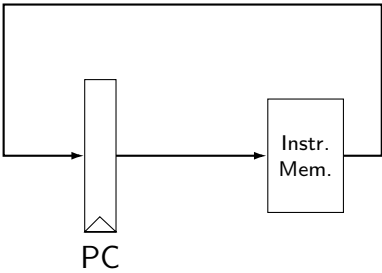
encoding: *8-byte little-endian address*

8 byte instructions, no opcode

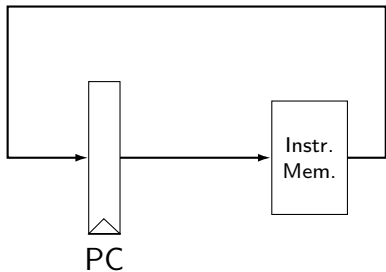
# jmp CPU



# jmp CPU



# jmp CPU



```
/* 0x00: */ jmp 0x10
```

```
/* 0x08: */ jmp 0x00
```

```
/* 0x10: */ jmp 0x08
```

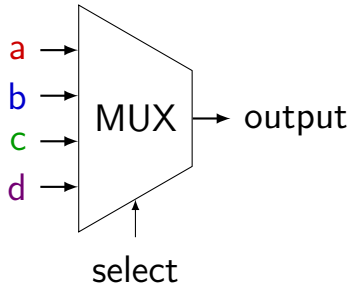
initially: PC = 0x00

after cycle 1: PC = 0x10

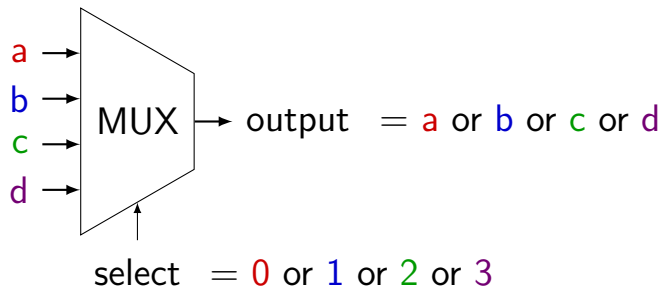
after cycle 2: PC = 0x08

after cycle 3: PC = 0x00

# multiplexers

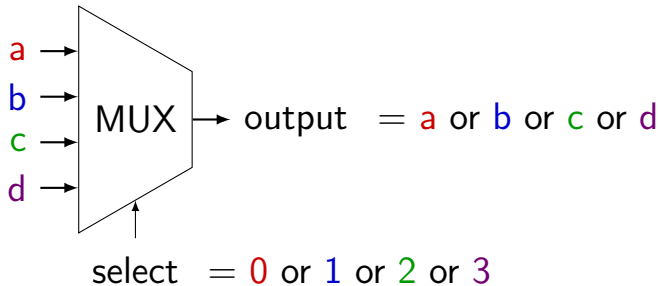


# multiplexers





# multiplexers



truth table:

select bit 1	select bit 0	output (many bits)
0	0	a
0	1	b
1	0	c
1	1	d

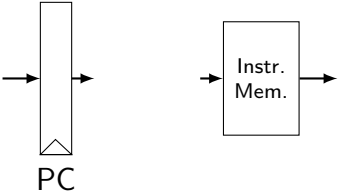
## Simple ISA 3: Jmp or No-Op

actual subset of Y86-64

`jmp LABEL` — encoded as `0x70` + address

`nop` — encoded as `0x10`

# jmp+nop CPU



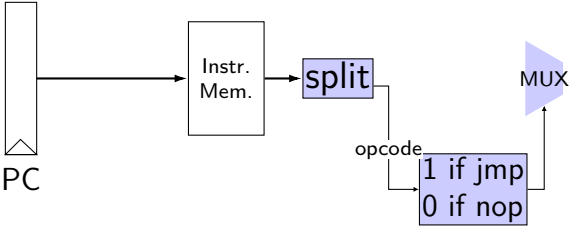
**nop**



**jmp** *Dest*



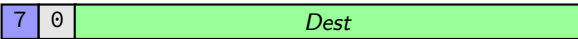
# jmp+nop CPU



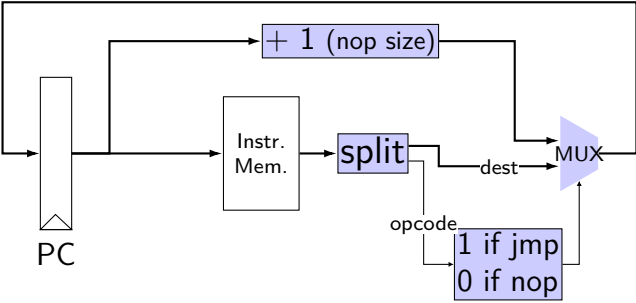
nop



jmp *Dest*



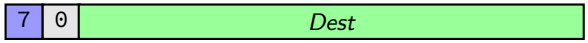
# jmp+nop CPU



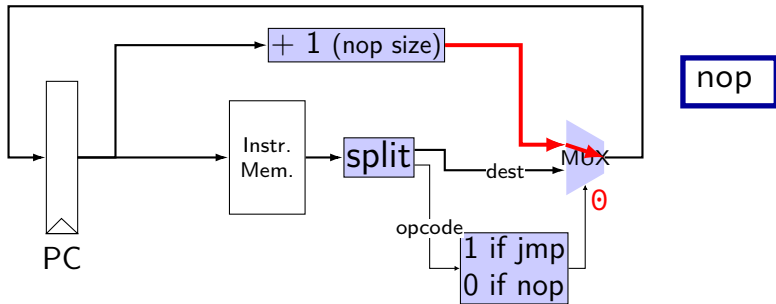
**nop**



**jmp** *Dest*



# jmp+nop CPU



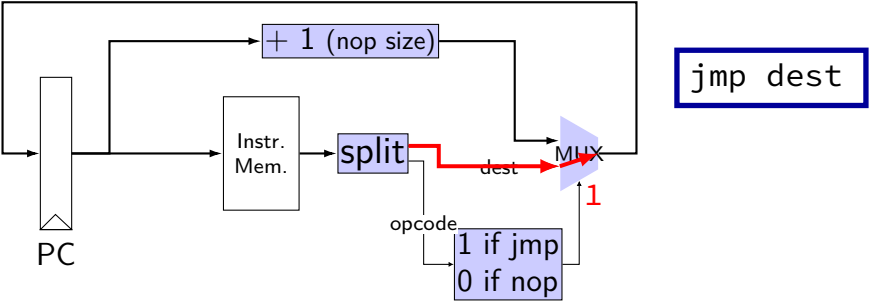
`nop`



`jmp Dest`



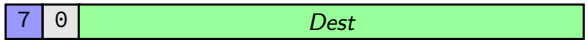
# jmp+nop CPU



nop



jmp Dest



## 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



# Summary

each instruction takes one cycle

divided into stages for **design convenience**

read values **from previous cycle**

send **new values** to state components

control what is sent with **MUXes**

# Backup Slides

# conditional movs

absoluteValueJumps:

```
    andq %rdi, %rdi
    jge same          ; if rdi >= 0, goto same
    irmovq $0, %rax  ; rax ← 0
    subq %rdi, %rax  ; rax ← rax (0) - rdi
    ret
```

```
same: rrmovq %rdi, %rax
      ret
```

absoluteValueCMov:

```
    irmovq $0, %rax
    subq %rdi, %rax  ; rax ← -rdi
    andq %rdi, %rdi
    cmovge %rdi, %rax ; if (rdi > 0) rax ← rdi
    ret
```

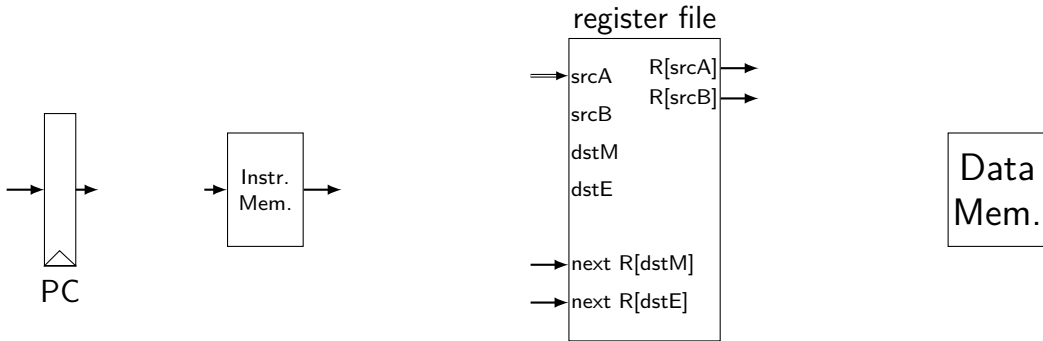
# Stages: pushq/popq

stage	pushq	popq
fetch	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$
decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\%rsp]$	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
execute	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow \text{valB} + 8$
memory	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valA}]$
write back	$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

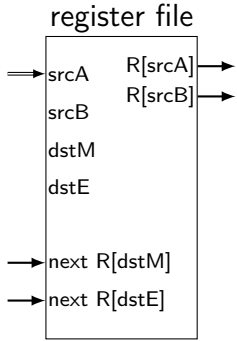
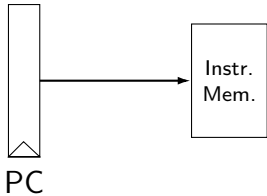
# Stages: pushq/popq

stage	pushq	popq
fetch	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$
decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\%rsp]$	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
execute	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow \text{valB} + 8$
memory	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valA}]$
write back	$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

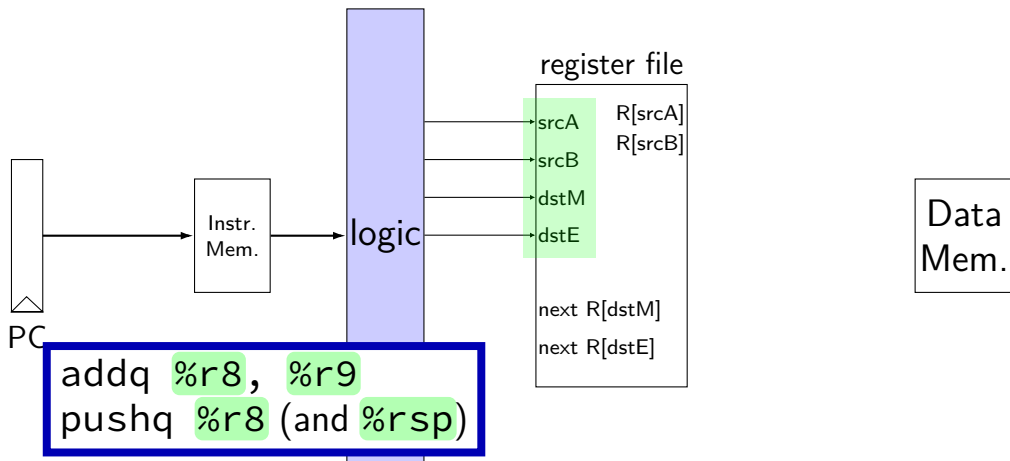
# connections in Y86-64



# connections in Y86-64

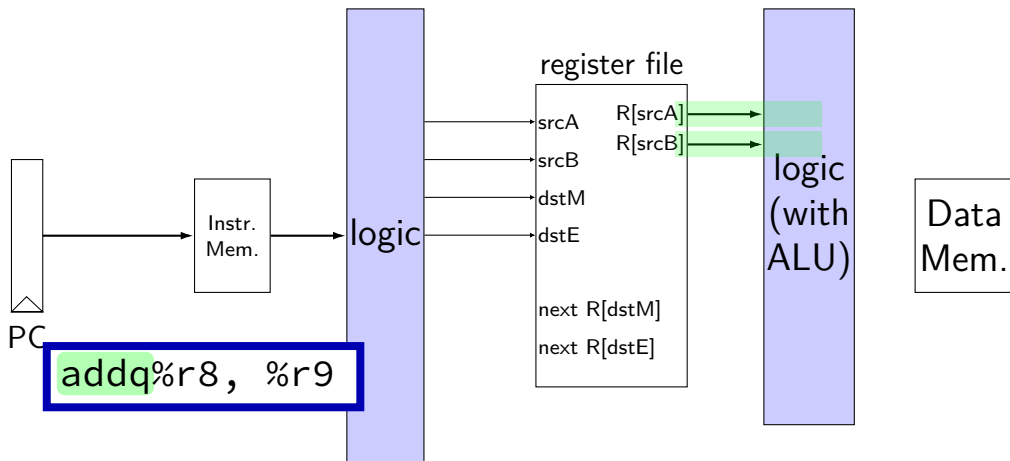


# connections in Y86-64

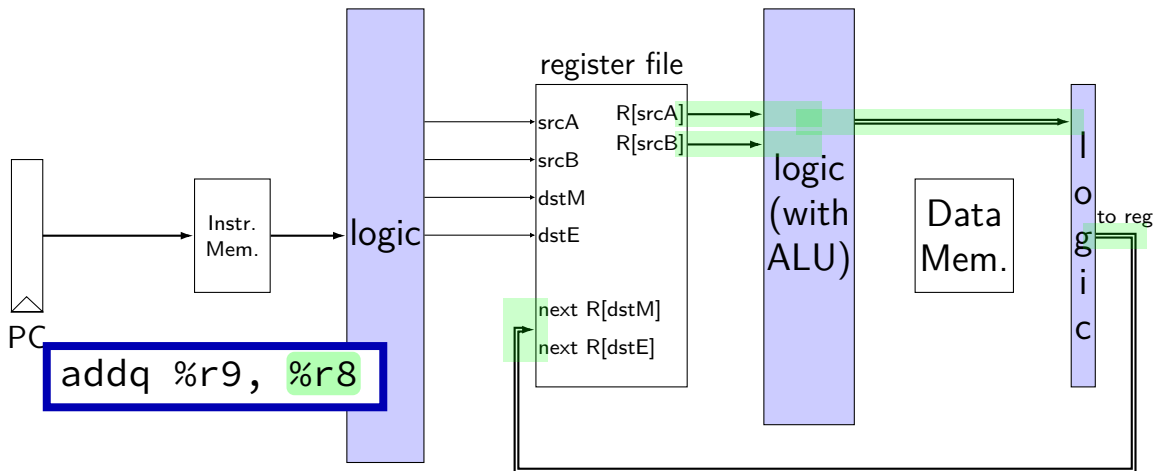




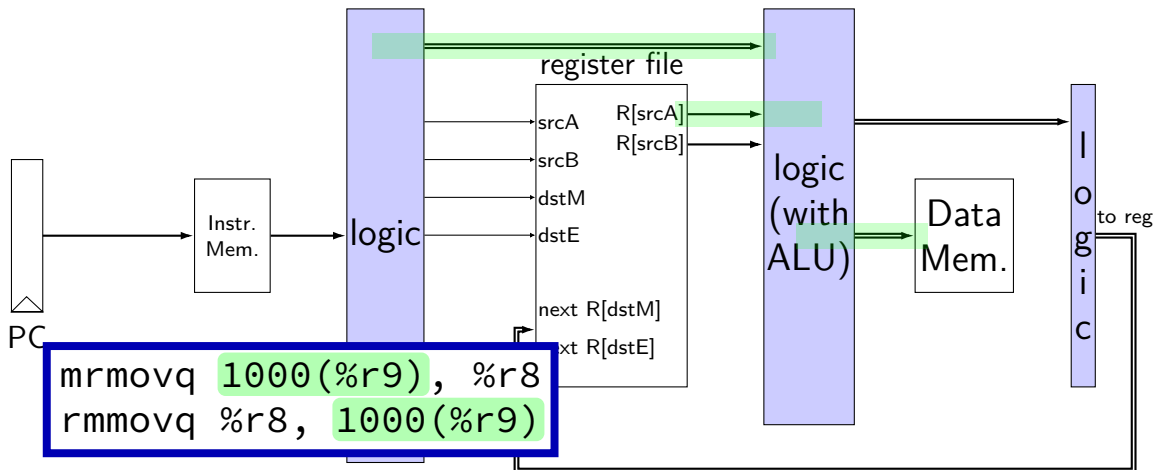
# connections in Y86-64



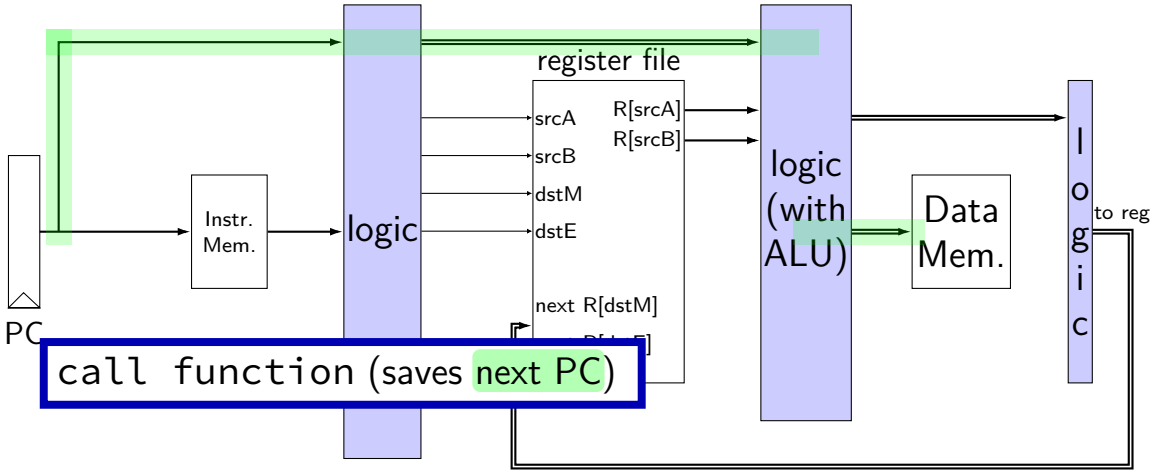
# connections in Y86-64



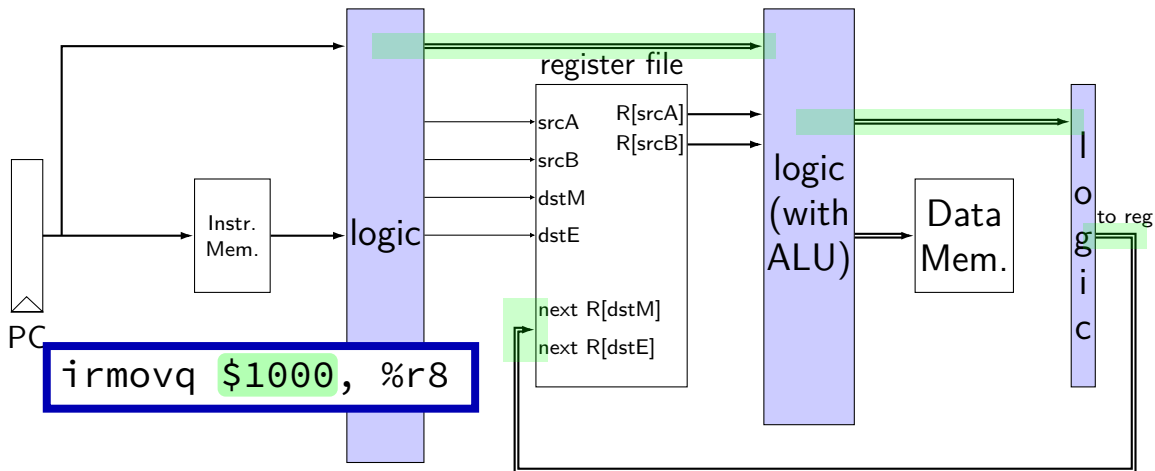
# connections in Y86-64



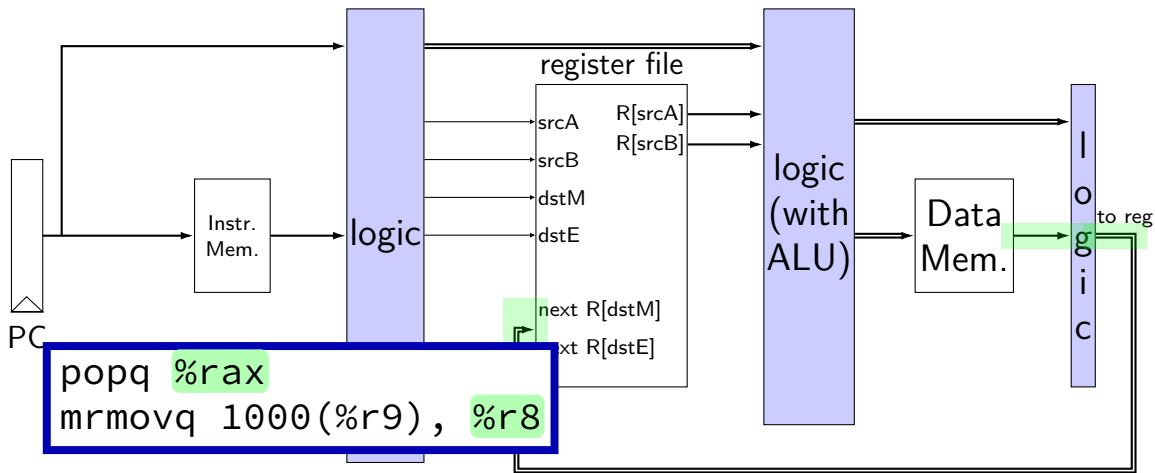
# connections in Y86-64



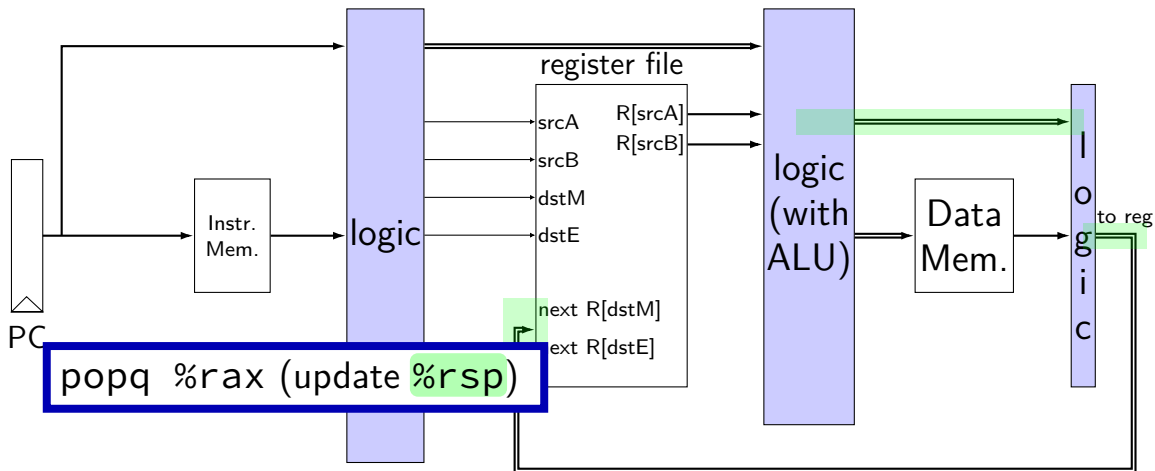
# connections in Y86-64



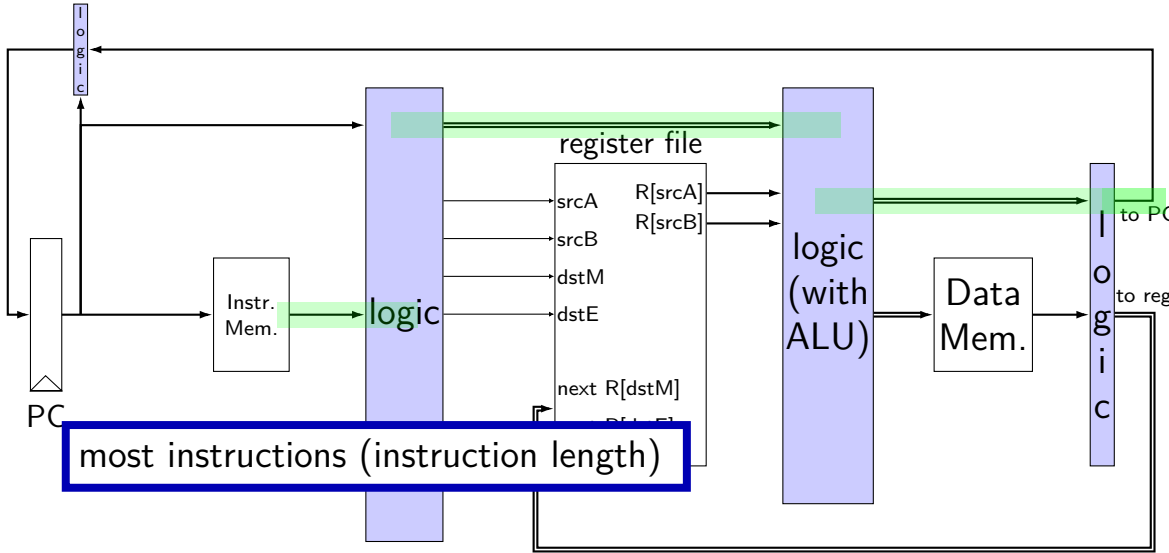
# connections in Y86-64



# connections in Y86-64

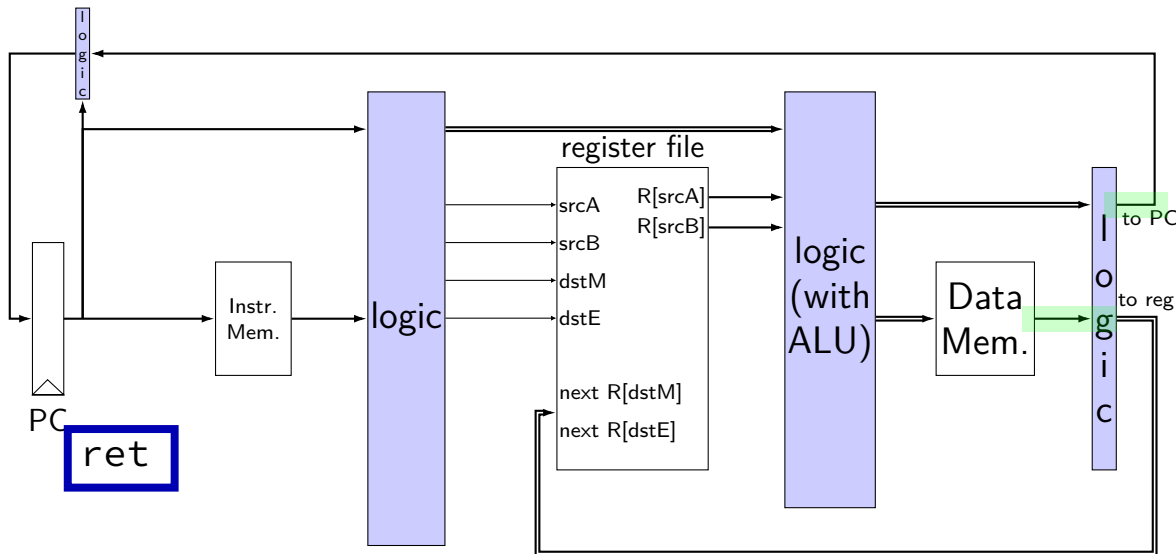


# connections in Y86-64

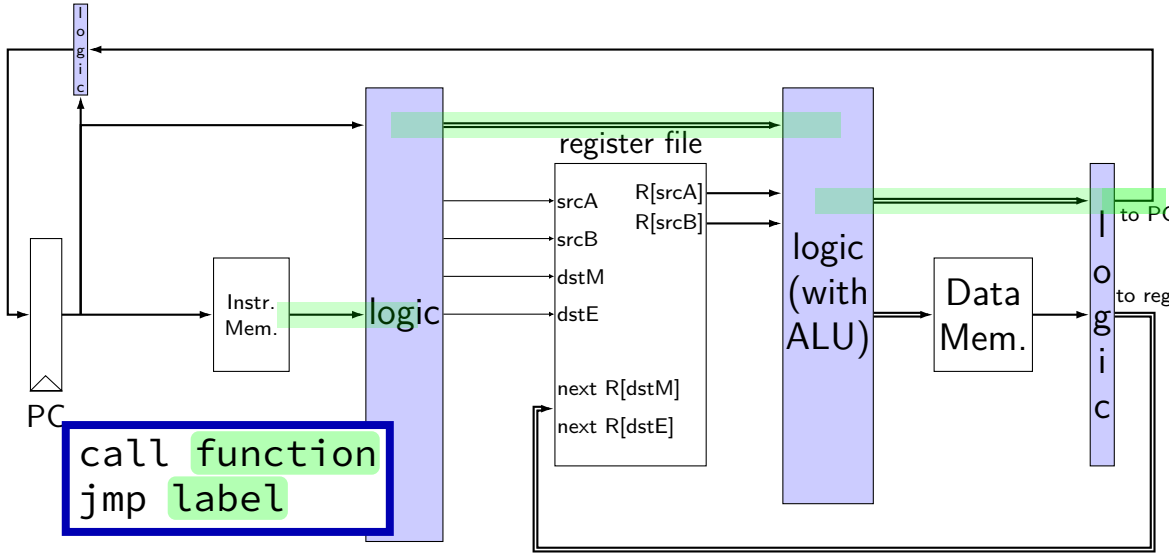




# connections in Y86-64

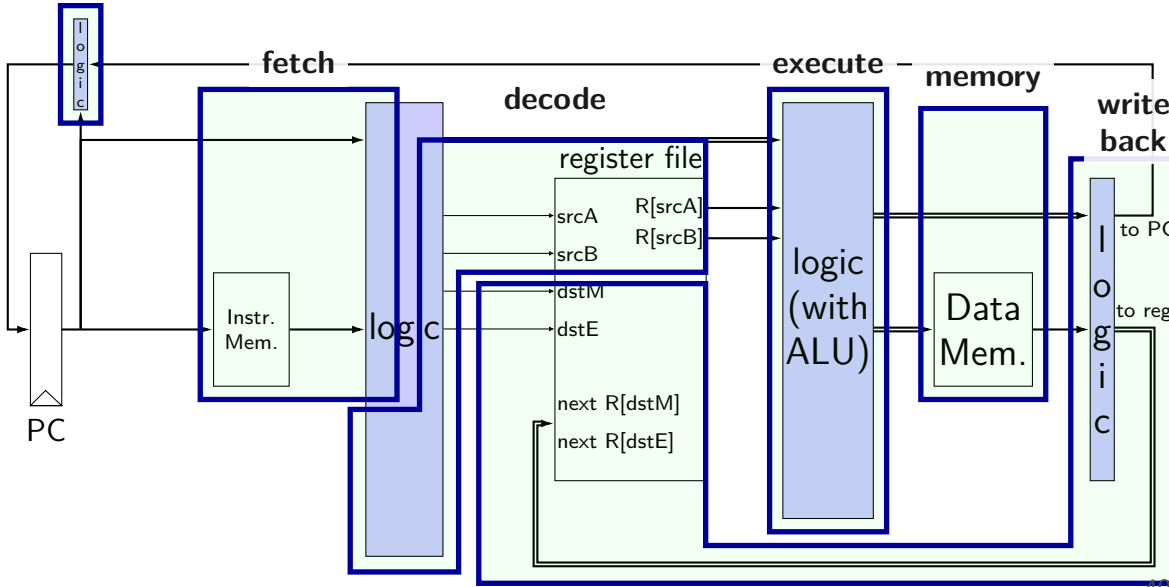


# connections in Y86-64



# stages

PC update

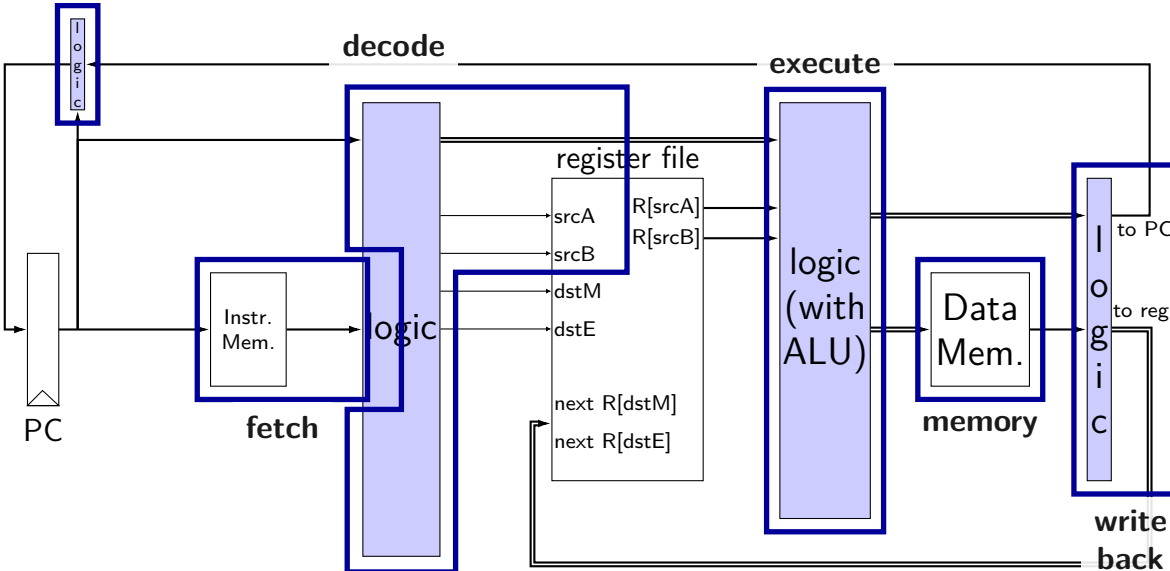


# Systematic construction

MUX	OPq	ret	callq	pushq	...
next PC	PC + len	memory out	from instr	PC + len	...
srcB	rB	—	—	%rsp	...

# stages

PC update



# Stages

conceptual division of instruction:

**fetch** — read instruction memory, split instruction

**decode** — read register file

**execute** — arithmetic (including of addresses)

**memory** — read or write data memory

**write back** — write to register file

**PC update** — compute next value of PC