Exam Review

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?

on office hour locations

on the homework

can use multiple statements and temporary variables

only arithmetic shifts available

on this week's quiz

layers of abstraction

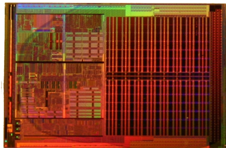
`x += y`	"Higher-level" language: C
`add %rbx, %rax`	Assembly: X86-64
60 03_{SIXTEEN}	Machine code: Y86
	???
	Gates / Transistors / Wires / Registers

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	
$\mathbf{2^{10}}$	**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	
	…	
$\mathbf{2^{20}}$	1 048 576	**M** (or Mi)
	…	
$\mathbf{2^{30}}$	1 073 741 824	**G** (or Gi)
2^{31}	2 147 483 648	
2^{32}	4 294 967 296	
	…	

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

endianness

address	value
0xFFFFFFFF	0x14
0xFFFFFFFE	0x45
0xFFFFFFFD	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
0xFFFFFFFE	0x45
0xFFFFFFFD	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
0xFFFFFFFE	0x45
0xFFFFFFFD	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
0xFFFFFFFE	0x45
0xFFFFFFFD	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
0xFFFFFFFE	0x45
0xFFFFFFFD	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

```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

```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

```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) …

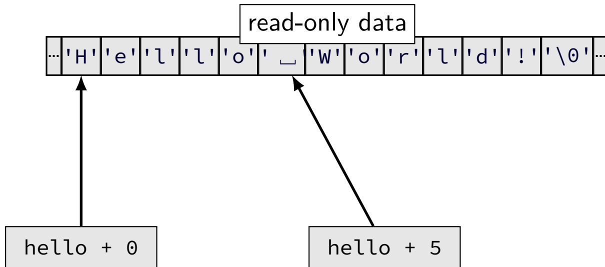
hello.s

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

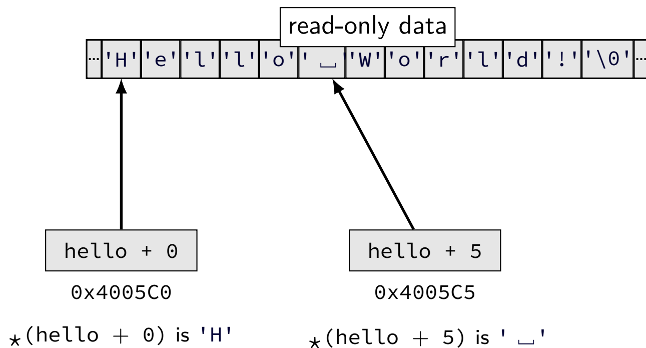
hello.o

```
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-0x0000000000

Contents of section .text:
 0000 4883ec08 bf000000 00e80000 0000b800  H.......
 0010 00000048 83c408c3                    H
```

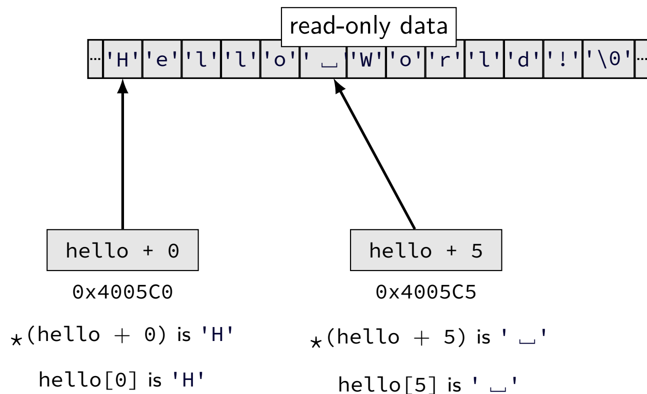
pointer arithmetic



pointer arithmetic

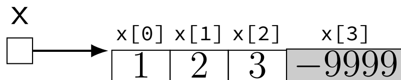


pointer arithmetic

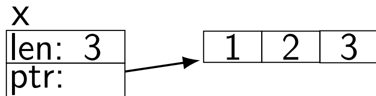


some lists

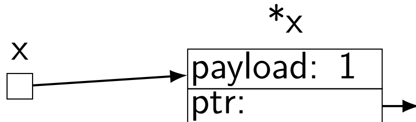
```c
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```




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



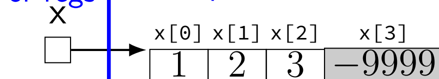

```c
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```



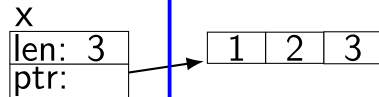
some lists

← on stack
or regs | on heap →

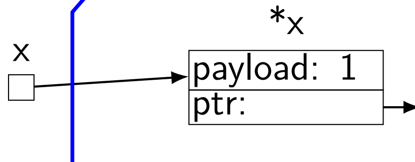
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```




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



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 (defualts to `1`)

`$` means constant

plain number/label means value in memory

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

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

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

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

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

compiling switches (1)

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

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

compiling switches (2)

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

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

compiling switches (3)

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

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

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

computed jumps

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

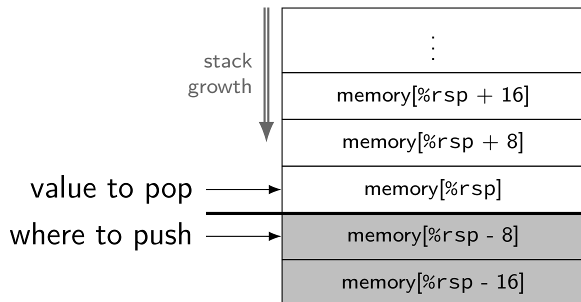
push/pop

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*							
OP`q` *rA*, *rB*	6	*fn*	*rA*	*rB*								
`j`*CC* *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: *OP*q

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*							
OP`q` *rA*, *rB*	6	*fn*	*rA*	*rB*								
`j`*CC Dest*	7	*cc*	*Dest*									
`call` *Dest*	8	0	*Dest*									
`ret`	9	0										
`pushq` *rA*	A	0	*rA*	F								
`popq` *rA*	B	0	*rA*	F								

fn	operation
0	`add`
1	`sub`
2	`and`
3	`xor`

Registers: *rA*, *rB*

byte:	0		1	
`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*
`mrmovq` *D(rB)*, *rA*	5	0	*rA*	*rB*
OP`q` *rA*, *rB*	6	*fn*	*rA*	*rB*
`j`*CC 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*							
`mrmovq` *D(rB)*, *rA*	5	0	*rA*	*rB*	*D*							
OP`q` *rA*, *rB*	6	*fn*	*rA*	*rB*								
`j`*CC 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*							
mrmovq *D(rB), rA*	5 0	*rA* *rB*	*D*							
*OP***q** *rA, rB*	6 *fn*	*rA* *rB*								
j*CC 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)
```

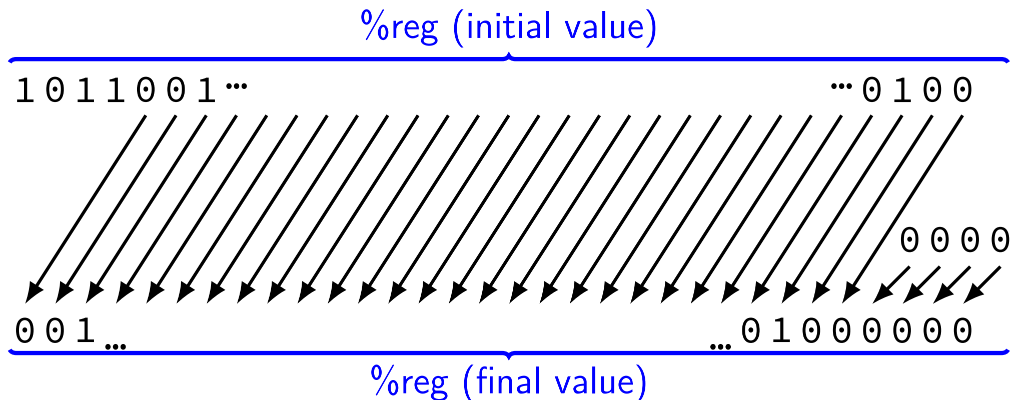
becomes

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

shift left

x86 instruction: **`shl`** — shift left

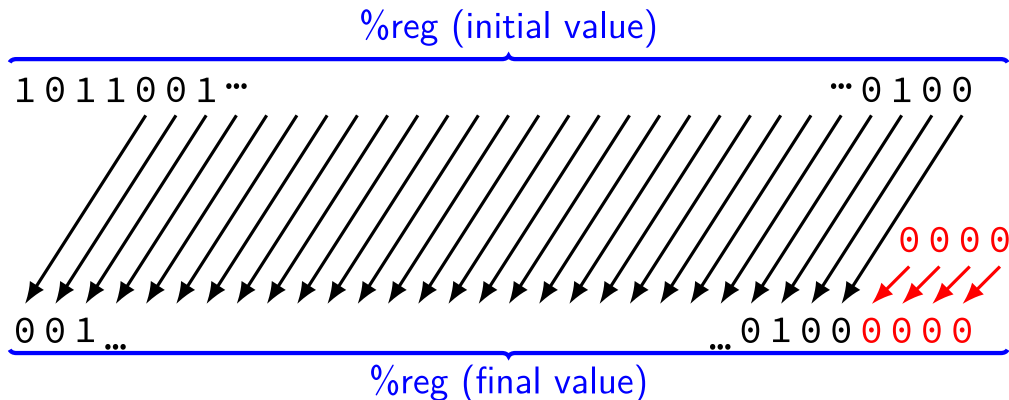
`shl` `$`*`amount`*`, %reg` (or variable: **`shr`** `%cl, %reg`)



shift left

x86 instruction: **`shl`** — shift left

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



left shift in math

```
1 << 0 == 1              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
```

left shift in math

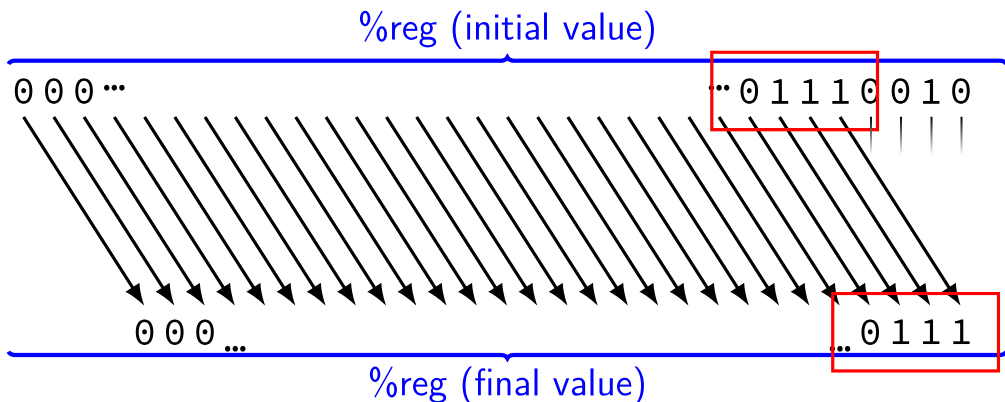
```
1 << 0 == 1                0000 0001
1 << 1 == 2                0000 0010
1 << 2 == 4                0000 0100

10 << 0 == 10              0000 1010
10 << 1 == 20              0001 0100
10 << 2 == 40              0010 1000
```

$$x \texttt{<<} y = x \times 2^y$$

logical right shift

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

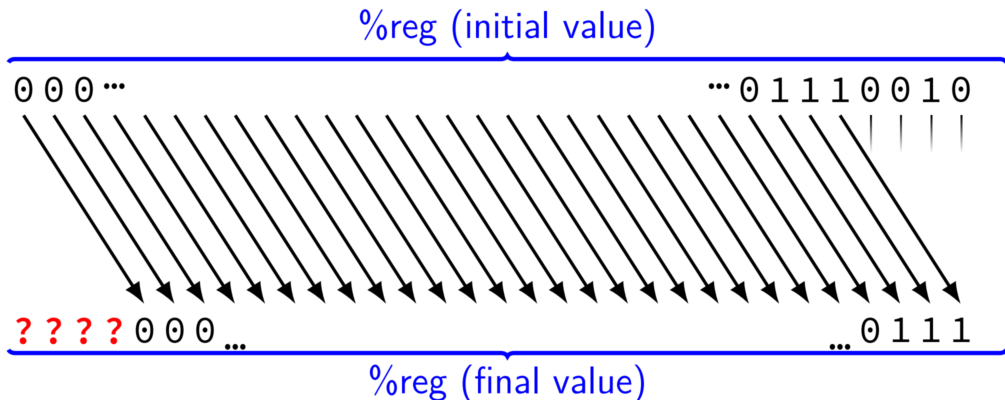
`shr` `$`*`amount`*`, %reg` (or variable: **`shr`** `%cl, %reg`)



logical right shift

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

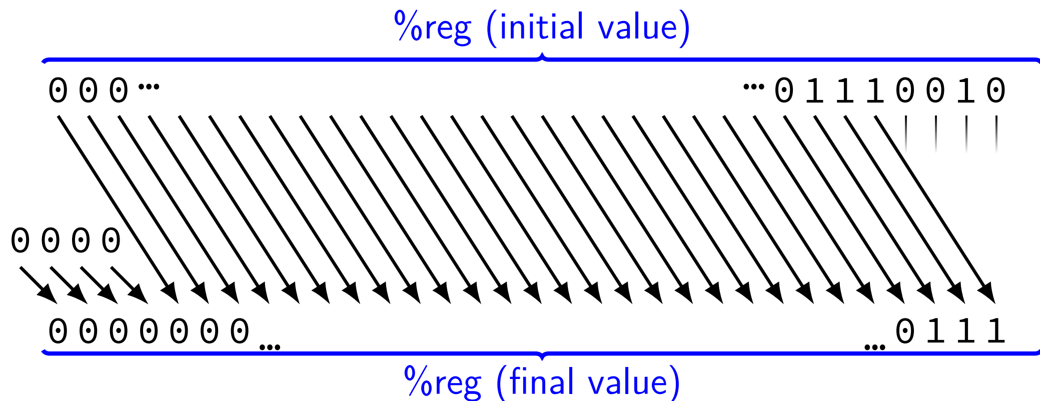
`shr` `$`*`amount`*`, %reg` (or variable: **`shr`** `%cl, %reg`)



logical right shift

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

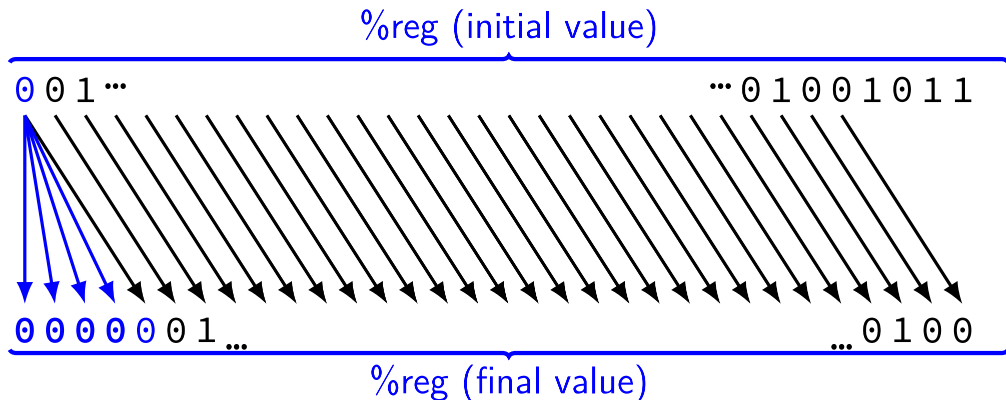
shr $*amount*, %reg (or variable: **shr** %cl, %reg)



arithmetic right shift

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

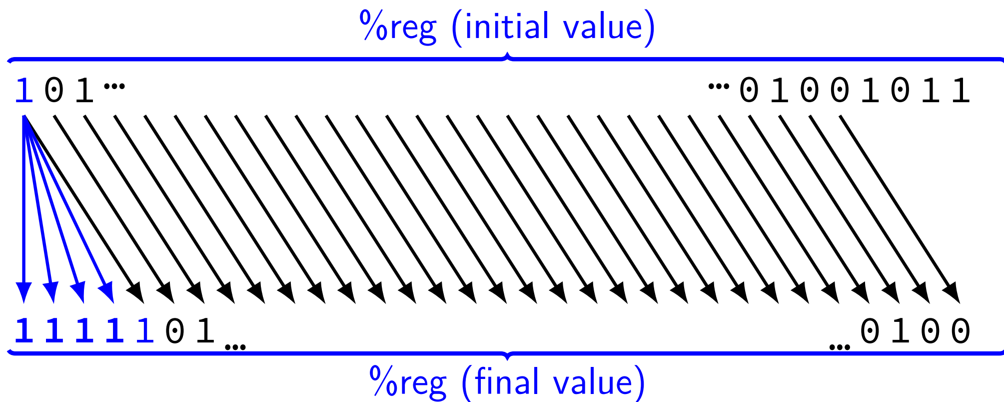
`sar` `$`*`amount`*`, %reg` (or variable: **`sar`** `%cl, %reg`)



arithmetic right shift

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

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



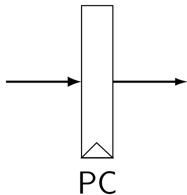
right shift in C

```
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
}
```

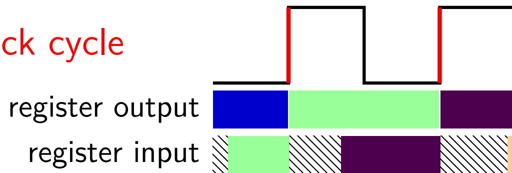
```
shift_signed:
    movl %edi, %eax
    sarl $5, %eax
    ret
```

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

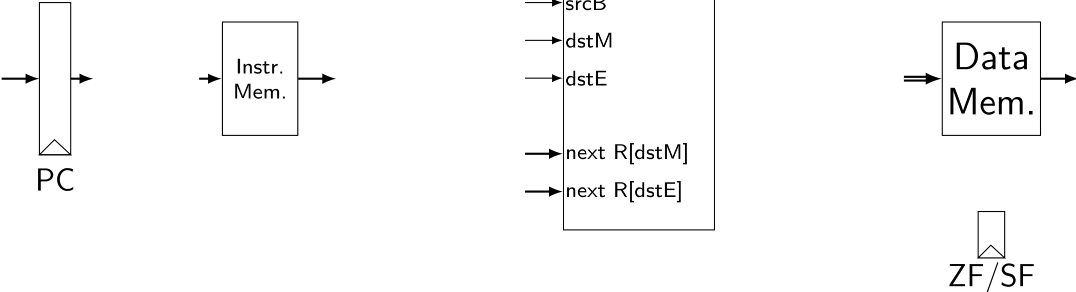
registers



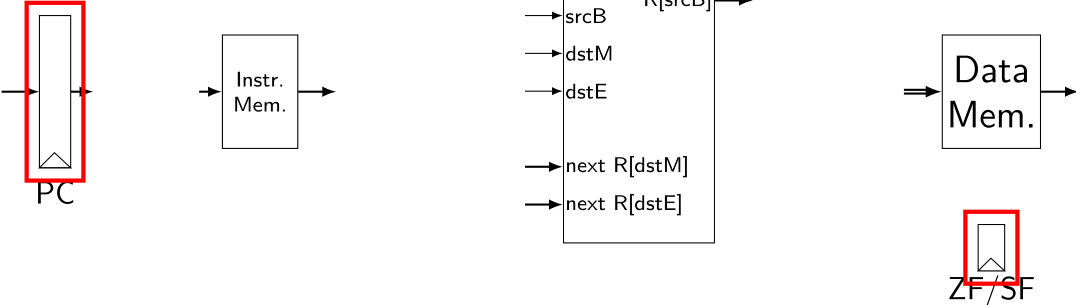
updates every clock cycle



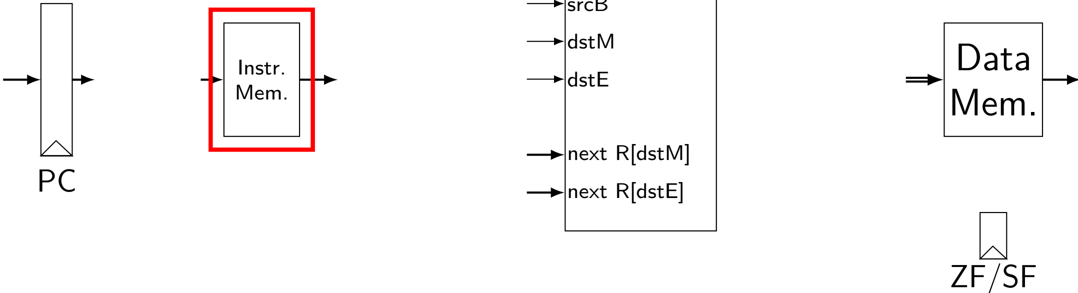
state in Y86-64



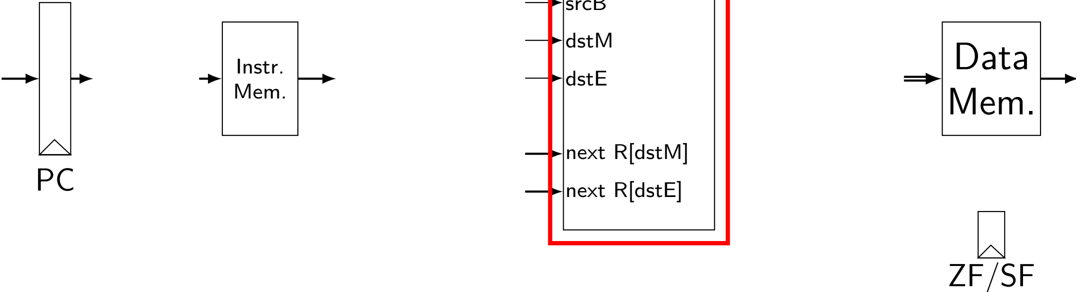
state in Y86-64



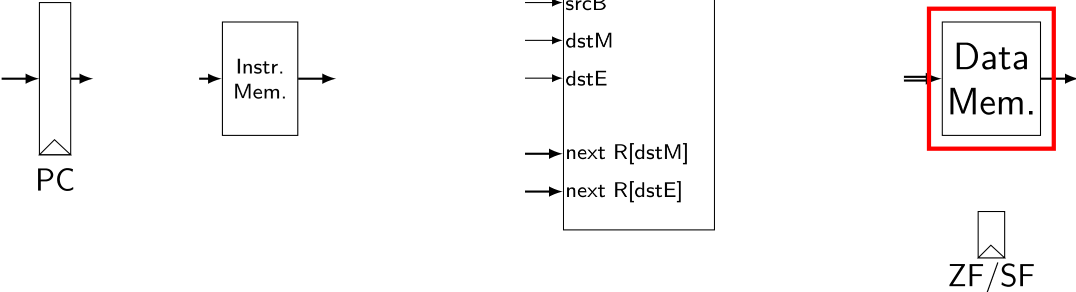
state in Y86-64



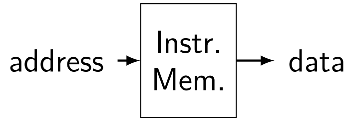
state in Y86-64

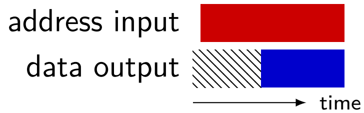


state in Y86-64

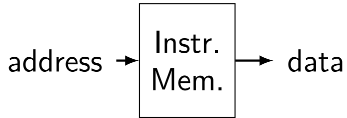


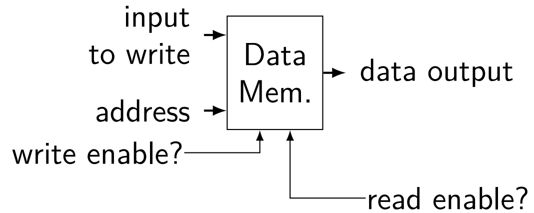
memories

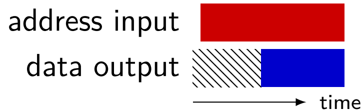




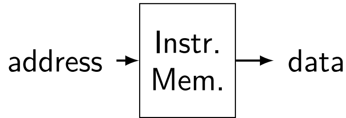
memories

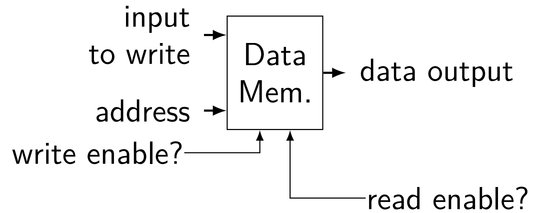


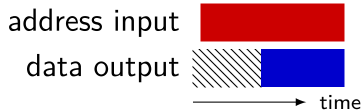


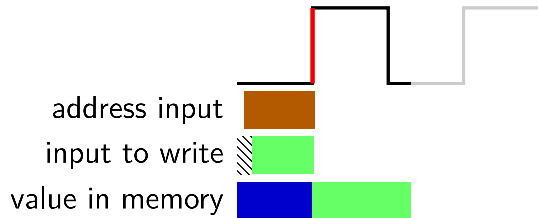


memories

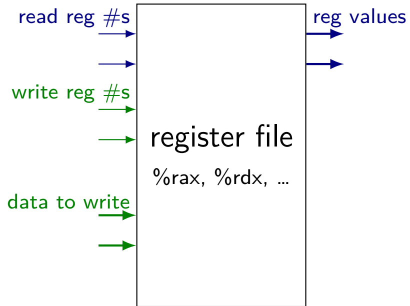




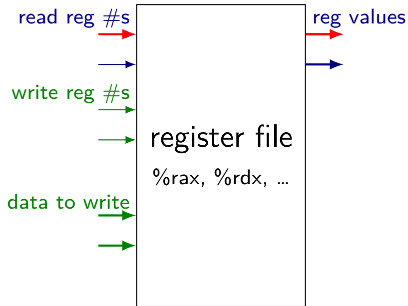


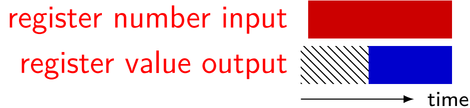


register file

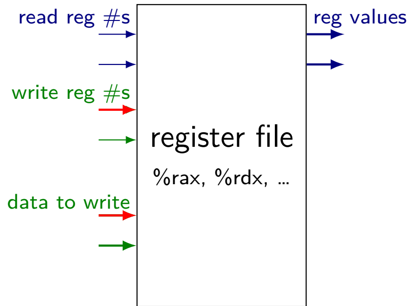


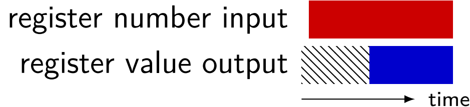
register file

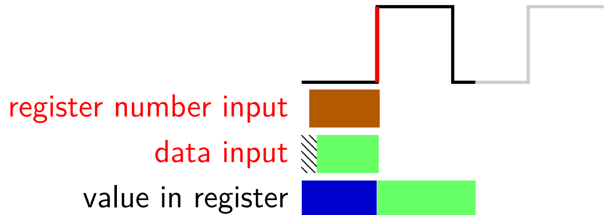




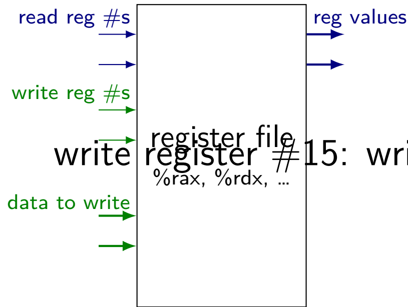
register file



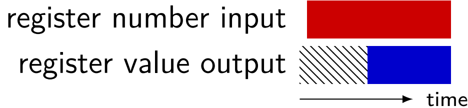


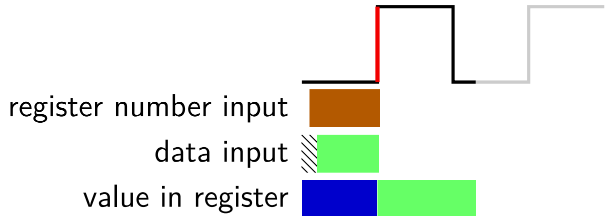


register file

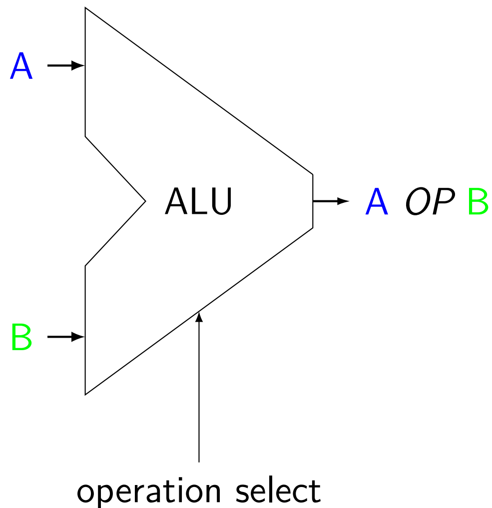


write register #15: write is ignoredread register #15: value is alway



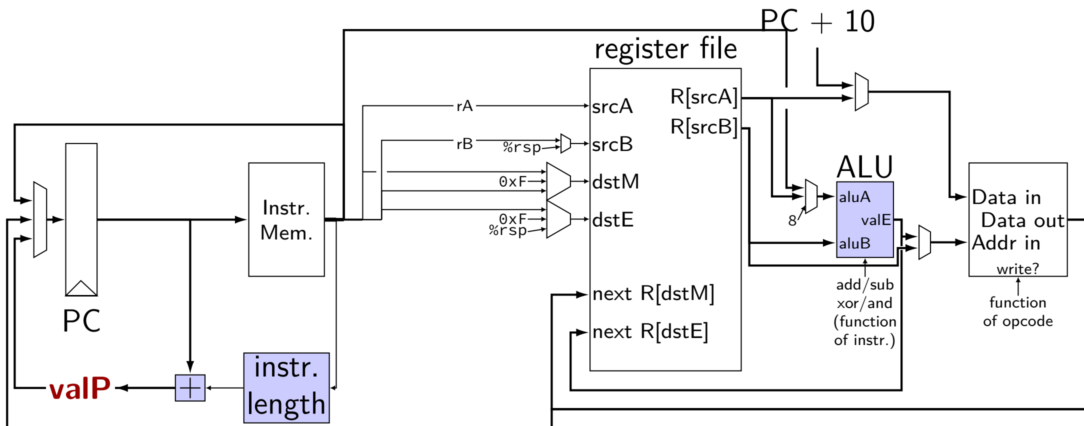


ALUs

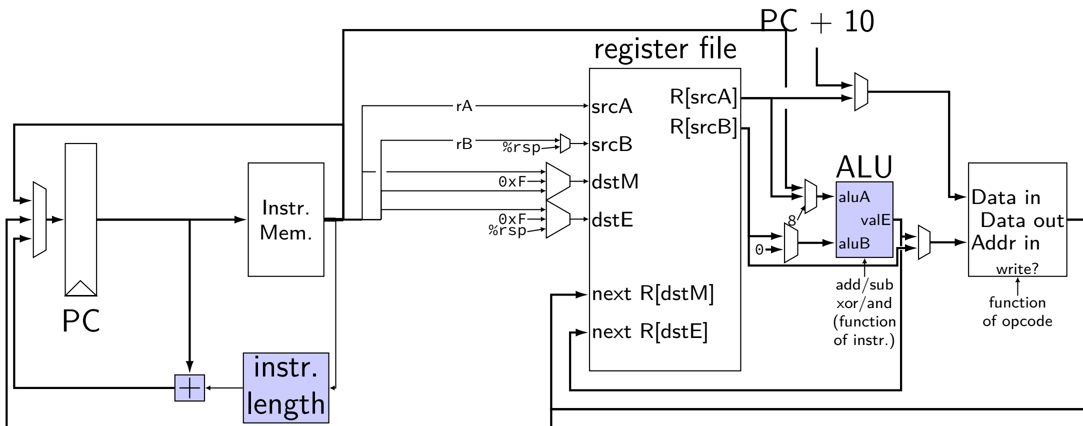


Operations needed:
add — **addq**, addresses
sub — **subq**
xor — **xorq**
and — **andq**
more?

SEQ circuit



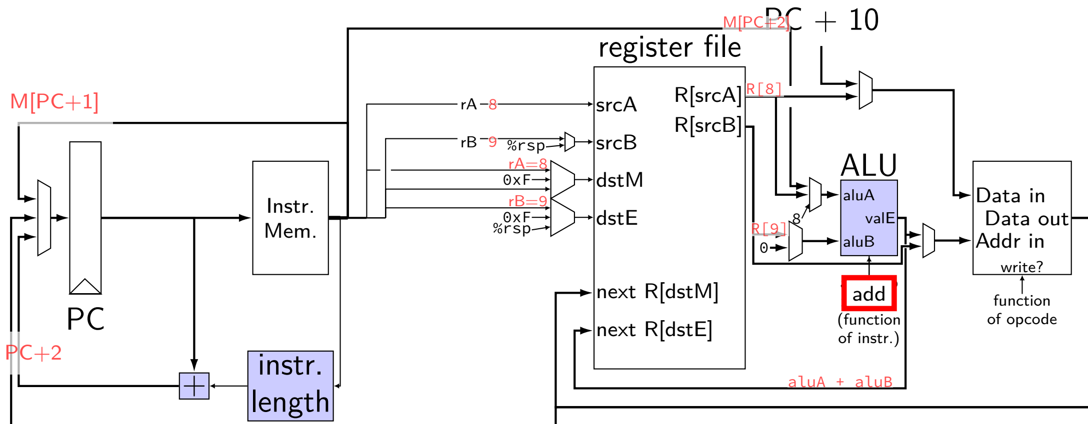
circuit: setting MUXes



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

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

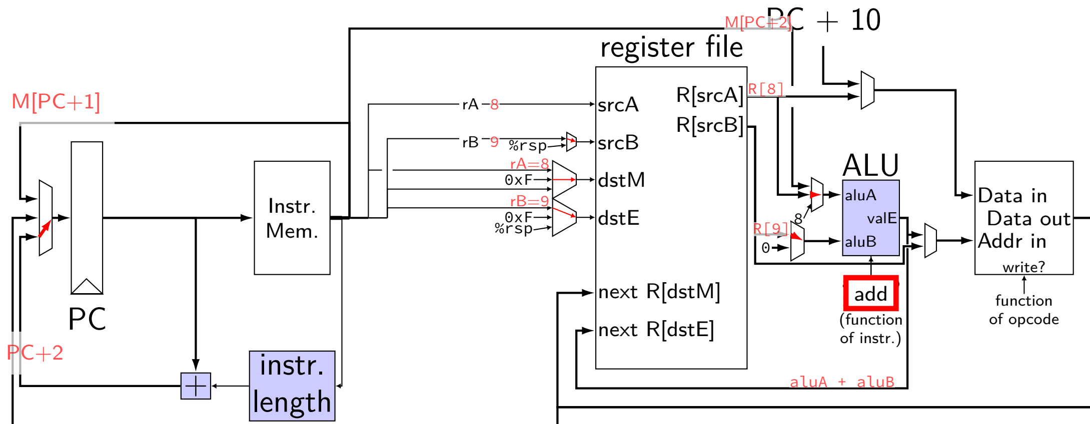
circuit: setting MUXes



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

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

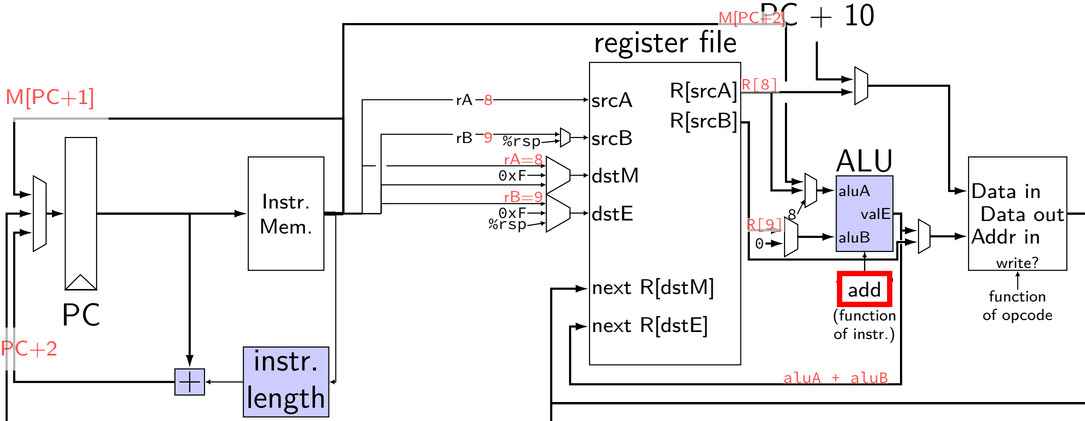
circuit: setting MUXes



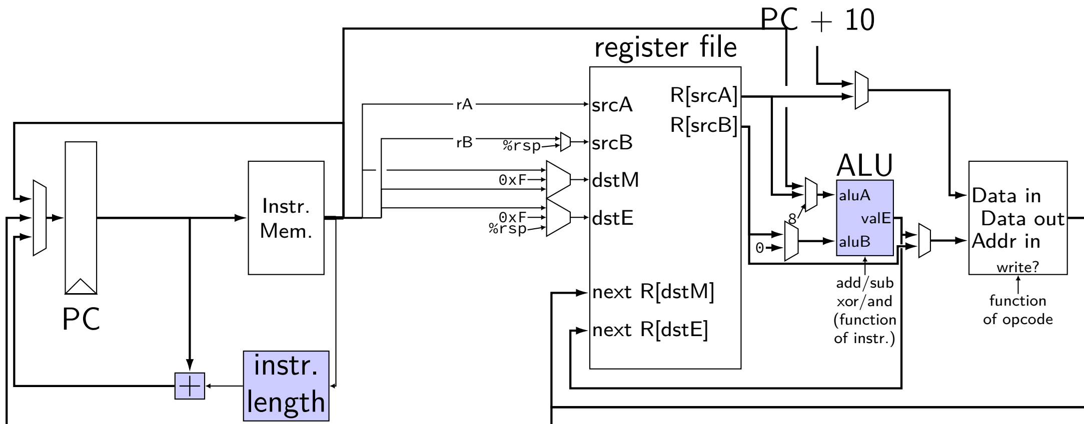
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

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

circuit: setting MUXes



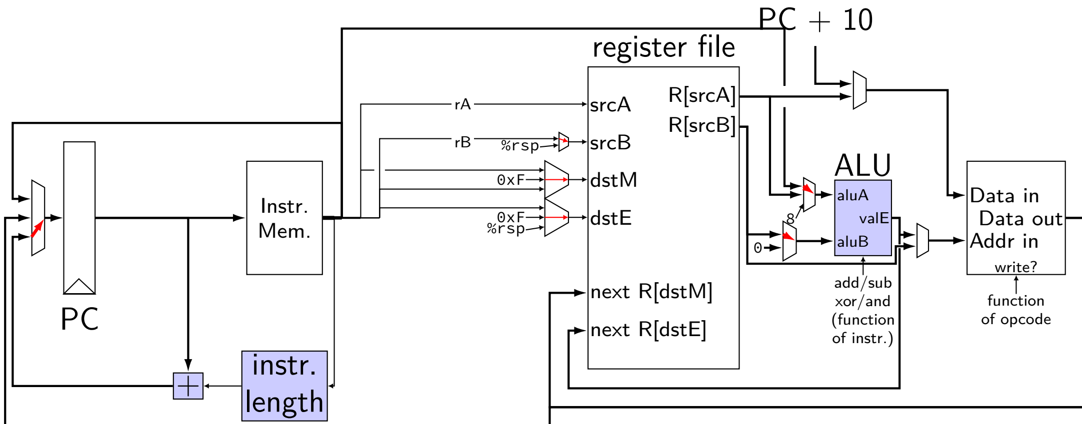
circuit: setting MUXes



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

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

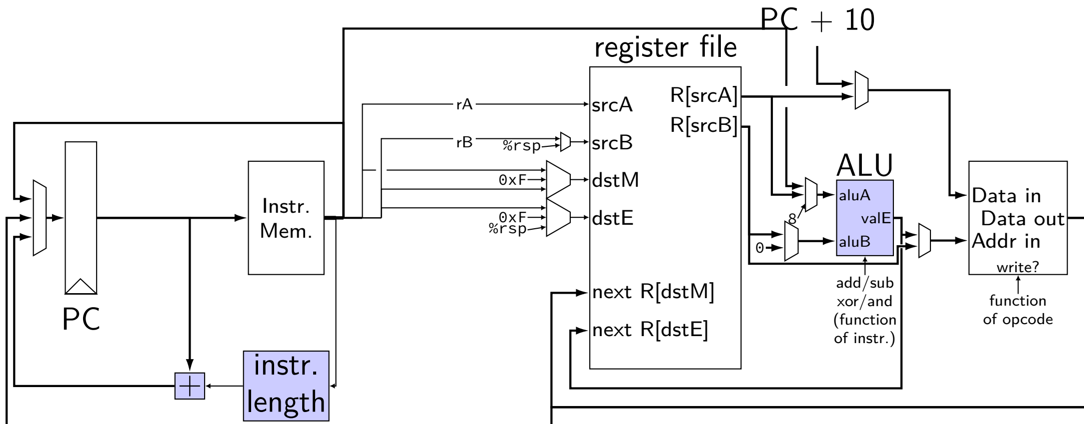
circuit: setting MUXes



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`**?