Midterm 1 Review

non-exam/quiz anonymous feedback

It is very unclear to what extent we are allowed to use stackoverflow for labs. Could you clarify this please?

- can work with other students in the class
- web info is okay for parts of the lab
- you should not ask questions to random people online
- cite in comment if you hand in a C/HCL2D file
- rules for homeworks are different

non-exam/quiz anonymous feedback

Would it be possible to make recordings that have a speedup option?

> right-click to download, then use your own player (e.g. VLC) with this option

bit numbering and `i10bytes` (1)

confusion 1: English language writes numbers in big-endian

most significant part first

confusion 2: normally, no “order” to bits in bytes

e.g. in memory, no way of addressing “bit 3” of a byte

instruction listings write bytes in memory address order

...but parts of bytes in English order

HCL2D: least-significant part has lowest number always

Y86 encoding table

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								

Y86 encoding table

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								

byte 0

Y86 encoding table

byte:	0	1	2–9
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* (bytes 1–8)	
call *Dest*	8 0	*Dest* (bytes 1–8)	
ret	9 0		
pushq *rA*	A 0	*rA* F	
popq *rA*	B 0	*rA* F	

least sig. 4 bits of byte 0

Y86 encoding table

byte:	0		1		2–9
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	

most sig. 4 bits of byte 0

bits 4–8

Y86 encoding table

byte:	0		1		2–9
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	

most sig. 4 bits of byte 1 / least sig. 4 bits of byte 1

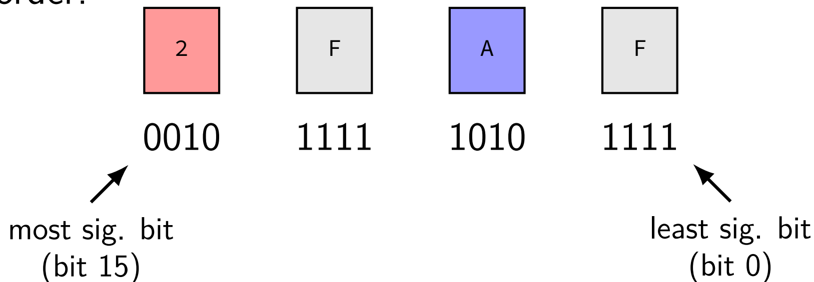
alternate view

`pushq %rbx` at memory address x

memory at $x+0$: `pushq` `F`; at $x+1$: `rbx` `F`

$x+0$: `A` `F`; at $x+1$: `2` `F`

as a little-endian 2-byte number in typical English order:



HCL2D on the exam

maybe reading (instead of using pictures of circuits)

not writing or debugging HCL2D

other review

material for exam

lecture

labs

homeworks

readings

exam format

75 minutes

multiple choice or short answer

approx. 20–25 questions

during lecture time

closed book/notes/etc.

non-exam topics

floating point

- was part of course in previous semesters

example exam topics

C — esp. pointer arithmetic

X86-64 assembly

bit-fiddling

basic circuits

ISA tradeoffs

Y86-64 ISA

SEQ — single-cycle Y86-64 processor

full list of exam topics?

look at the schedule

it's what we do when making the exam

exam review

via anonymous feedback

pretty much everything on the last post-quiz was confusing

questions

backup slides

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

```
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) …
```

C arrays/pointers

```
TYPE array[100];
TYPE *x = array;
    /* x points to array[0] */

x[0] == *x == *(x + 0)
x[1] == *(x + 1)

x = array + 4;
    /* x points to array[4] */

sizeof(x)       == sizeof(TYPE *) == sizeof(void *
sizeof(array)   == sizeof(TYPE) * 100
sizeof(*x)      == sizeof(TYPE)
sizeof(*array) == sizeof(TYPE)
```

typedef struct (2)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;

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

typedef struct (2)

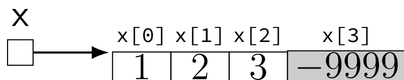
```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;

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

typedef struct (2)

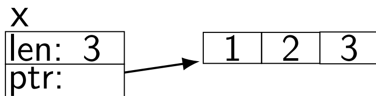
```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;

// same as:
typedef struct other_name_for_rational {
    int numerator;
    int denominator;
} rational;

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

lists homework

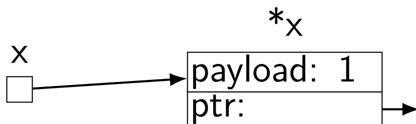
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```




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

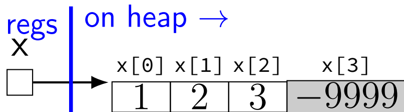



```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```

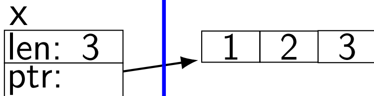


lists homework

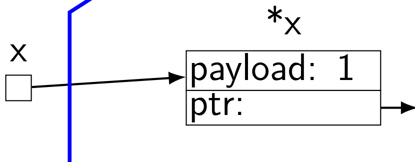
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```




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



undefined behavior

C has a standard

defines what “C” is

doesn’t specify everything:

- signed integer overflow
- out-of-bounds array access
- shifts by more than type width
- writing to string constants

compilers choose different things each time

- example: optimize away handling of overflow

undefined behavior example (2)

```
int test(int number) {
    return (number + 1) > number;
}
```

Optimized:

```
test:
    movl    $1, %eax        ; eax ← 1
    ret
```

Less optimized:

```
test:
    leal    1(%rdi), %eax ; eax ← rdi + 1
    cmpl    %eax, %edi
    setl    %al             ; al ← eax < edi
    movzbl  %al, %eax       ; eax ← al
    ret
```

x86-64 calling convention

registers for first 6 arguments:

- `%rdi` (or `%edi` or `%di`, etc.), then
- `%rsi` (or `%esi` or `%si`, etc.), then
- `%rdx` (or `%edx` or `%dx`, etc.), then
- `%rcx` (or `%ecx` or `%cx`, etc.), then
- `%r8` (or `%r8d` or `%r8w`, etc.), then
- `%r9` (or `%r9d` or `%r9w`, etc.)

rest on stack

return value in `%rax`

don't memorize: Figure 3.28 in book

AT&T syntax in one slide

destination last

`()` means value in memory

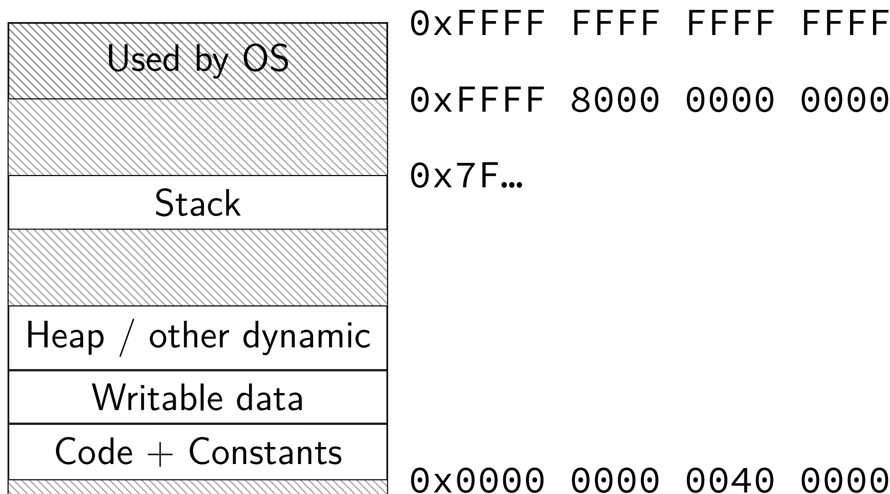
`disp(base, index, scale)` same as
`memory[disp + base + index * scale]`

- omit disp (defaults to `0`)
- and/or omit base (defaults to `0`)
- and/or scale (defualts to `1`)

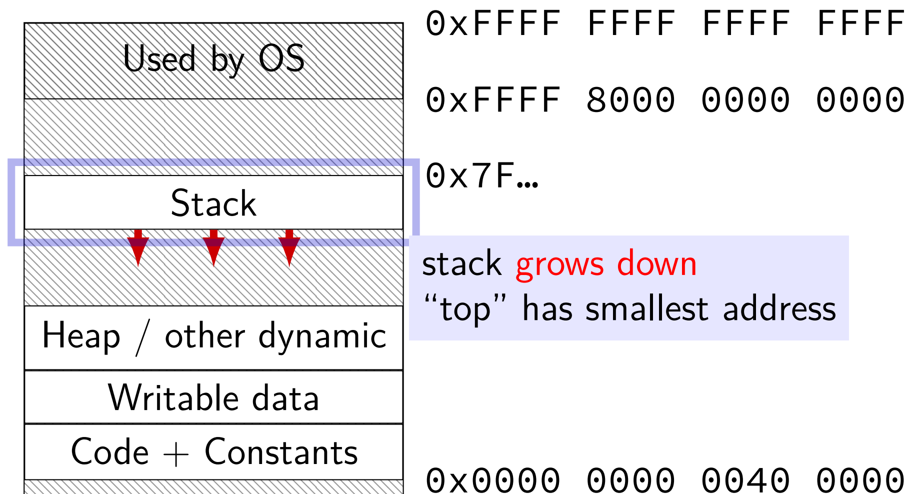
`$` means constant

plain number/label means value in memory

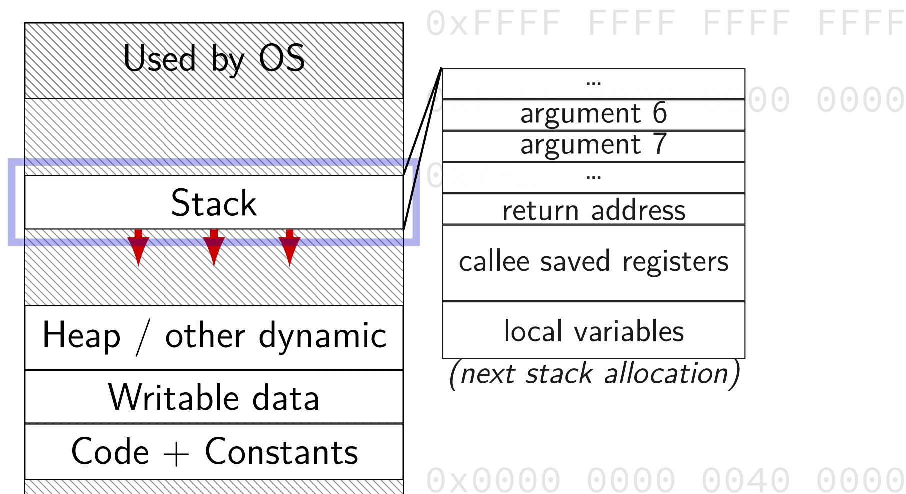
Program Memory (x86-64 Linux)



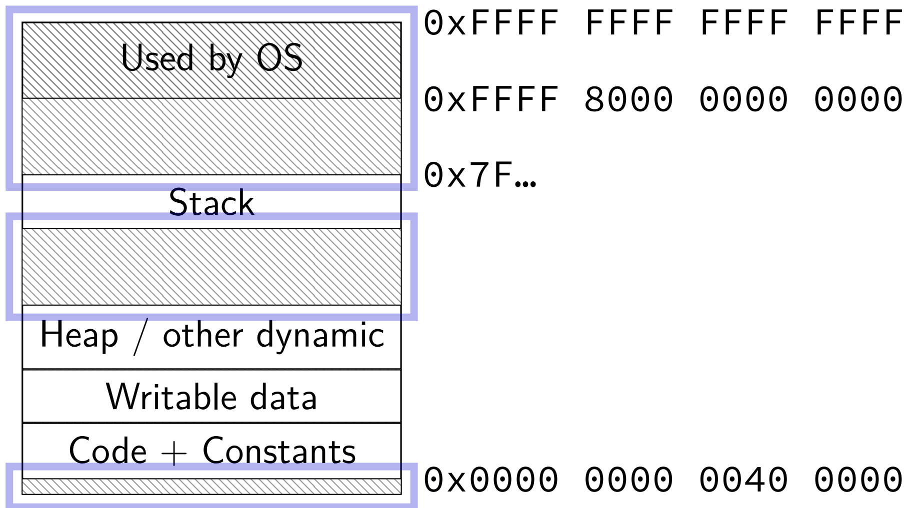
Program Memory (x86-64 Linux)



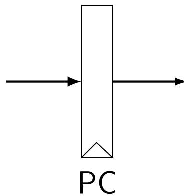
Program Memory (x86-64 Linux)



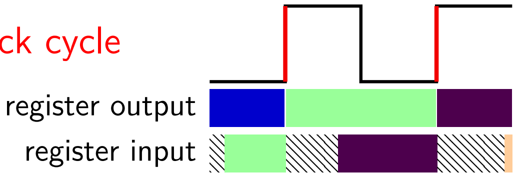
Program Memory (x86-64 Linux)



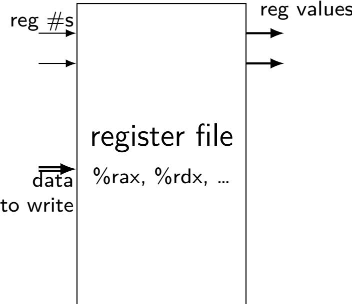
Registers



updates every clock cycle



Register file



Stages and Time

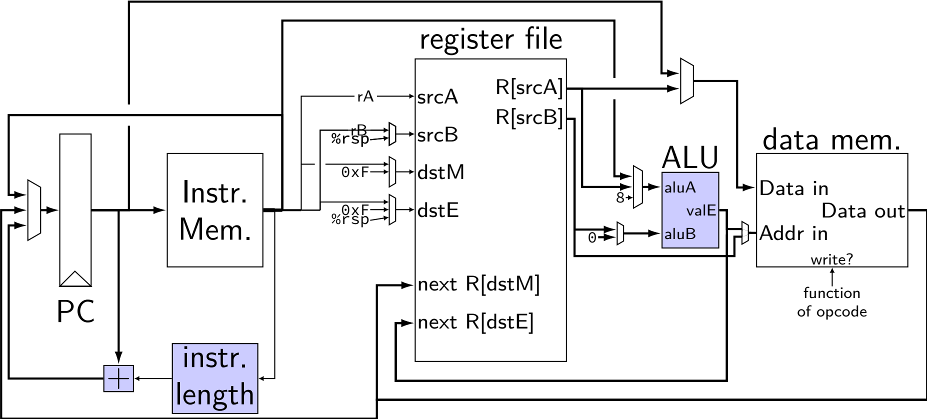
fetch / decode / execute / memory / write back / PC update

For the single-cycle design, order when these events happen `pushq %rax` instruction:

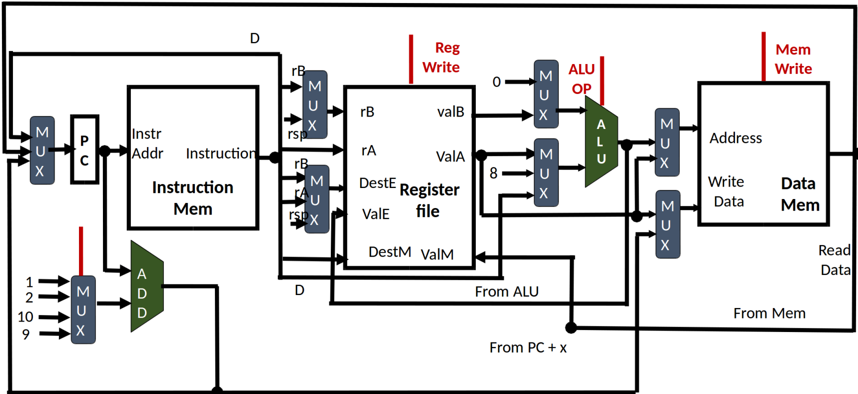
1. instruction read
2. memory changes
3. `%rsp` changes
4. PC changes

a. 1; then 2, 3, and 4 in any order
b. 1; then 2, 3, and 4 at almost the same time
c. 1; then 2; then 3; then 4
d. 1; then 3; then 2; then 4
e. 1; then 2; then 3 and 4 at almost the same time

SEQ



SEQ



Short-Circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

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

Short-Circuit (&&)

```
1  #include <stdio.h>
2  int zero() { printf("zero()\n"); return 0; }
3  int one() { printf("one()\n"); return 1; }
4  int main() {
5      printf(">␣%d\n", zero() && one());
6      printf(">␣%d\n", one() && zero());
7      return 0;
8  }
```

```
zero()
> 0
one()
zero()
> 0
```

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

Short-Circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">␣%d\n", zero() && one());
6     printf(">␣%d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

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

Short-Circuit (&&)

```
1  #include <stdio.h>
2  int zero() { printf("zero()\n"); return 0; }
3  int one() { printf("one()\n"); return 1; }
4  int main() {
5      printf("> %d\n", zero() && one());
6      printf("> %d\n", one() && zero());
7      return 0;
8  }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	0	1
0	0	0
1	0	1

Short-Circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">␣%d\n", zero() && one());
6     printf(">␣%d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	0	1
0	0	0
1	0	1

Short-Circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">␣%d\n", zero() || one());
6     printf(">␣%d\n", one() || zero());
7     return 0;
8 }
```

```
zero()
one()
> 1
one()
> 1
```

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

Short-Circuit (||)

```
1  #include <stdio.h>
2  int zero() { printf("zero()\n"); return 0; }
3  int one() { printf("one()\n"); return 1; }
4  int main() {
5      printf(">␣%d\n", zero() || one());
6      printf(">␣%d\n", one() || zero());
7      return 0;
8  }
```

```
zero()
one()
> 1
one()
> 1
```

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

Short-Circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf(">␣%d\n", zero() || one());
6     printf(">␣%d\n", one() || zero());
7     return 0;
8 }
```

```
zero()
one()
> 1
one()
> 1
```

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

Short-Circuit (||)

```
1  #include <stdio.h>
2  int zero() { printf("zero()\n"); return 0; }
3  int one() { printf("one()\n"); return 1; }
4  int main() {
5      printf(">␣%d\n", zero() || one());
6      printf(">␣%d\n", one() || zero());
7      return 0;
8  }
```

```
zero()
one()
> 1
one()
> 1
```

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

Short-Circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

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

Left shift

```
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

```
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
```

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

Right shift

Undefined: ~~`x << -1`~~ Instead: `x >> 1`

```
1 >> 0 == 1          0000 0001
1 >> 1 == 0          0000 0000
1 >> 2 == 0          0000 0000

10 >> 0 == 10        0000 1010
10 >> 1 == 5         0000 0101
10 >> 2 == 2         0000 0010
```

Right shift

Undefined: ~~`x << -1`~~ Instead: `x >> 1`

```
1 >> 0 == 1        0000 0001
1 >> 1 == 0        0000 0000
1 >> 2 == 0        0000 0000

10 >> 0 == 10      0000 1010
10 >> 1 == 5       0000 0101
10 >> 2 == 2       0000 0010
```

$$x \texttt{ >> } y = \lfloor x \times 2^{-y} \rfloor$$

Shifts and negative numbers

$-$`10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Shifts and negative numbers

$-$`10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Option 1: binary `1111 … 1011` $=$
$-5 = -10 \times 2^{-k}$

copy sign bit

Option 2: binary `0111 … 1011` $= 2^{31} - 5$

always use zero

Shifts and negative numbers

`-10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Option 1: binary `1111 … 1011` = $-5 = -10 \times 2^{-k}$

copy sign bit

arithmetic

Option 2: binary `0111 … 1011` = $2^{31} - 5$

always use zero

logical

Shifts and negative numbers

`-10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Option 1: binary `1111 … 1011` $= -5 = -10 \times 2^{-k}$

copy sign bit

arithmetic

Option 2: binary `0111 … 1011` $= 2^{31} - 5$

always use zero

logical

Typical RISC ISA properties

theme: simpler to implement

fewer, simpler instructions

seperate instructions to access memory

fixed-length instructions

more registers

no “loops” within single instructions

no instructions with two memory operands

few addressing modes

Y86-64: Simple condition codes (1)

If `%r9` is -1 and `%r10` is 1:

```
subq %r10, %r9
```

r9 becomes $-1 - (1) = -2$.

$\text{SF} = 1$ (negative)
$\text{ZF} = 0$ (not zero)

```
andq %r10, %r10
```

r10 becomes 1

$\text{SF} = 0$ (non-negative)
$\text{ZF} = 0$ (not zero)

Y86-64: Using condition codes

`subq SECOND, FIRST` (value = FIRST - SECOND)

`j__` or `cmov__`	condition code bit test	value test
`le`	$SF = 1$ or $ZF = 1$	value ≤ 0
`l`	$SF = 1$	value < 0
`e`	$ZF = 1$	value $= 0$
`ne`	$ZF = 0$	value $\neq 0$
`ge`	$SF = 0$	value ≥ 0
`g`	$SF = 0$ and $ZF = 0$	value > 0