

Midterm 1 Review

1

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**

2

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

3

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

4

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					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

5

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					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

byte 0

5

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					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

least sig. 4 bits of byte 0

bits 0-4

5

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					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

most sig. 4 bits of byte 0

bits 4-8

5

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					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

most sig. 4 bits of byte 1
 bit 15 of byte 0

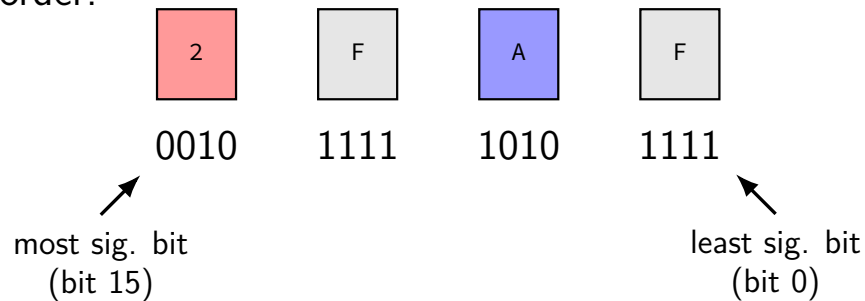
alternate view

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

9

exam format

75 minutes

multiple choice or short answer

approx. 20–25 questions

during lecture time

closed book/notes/etc.

10

non-exam topics

floating point

was part of course in previous semesters

11

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

12

full list of exam topics?

look at the schedule

it's what we do when making the exam

13

exam review

14

via anonymous feedback

pretty much everything on the last post-quiz was confusing

15

questions

16

backup slides

17

What's in those files?

hello.c

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

18

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!"
```

18

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

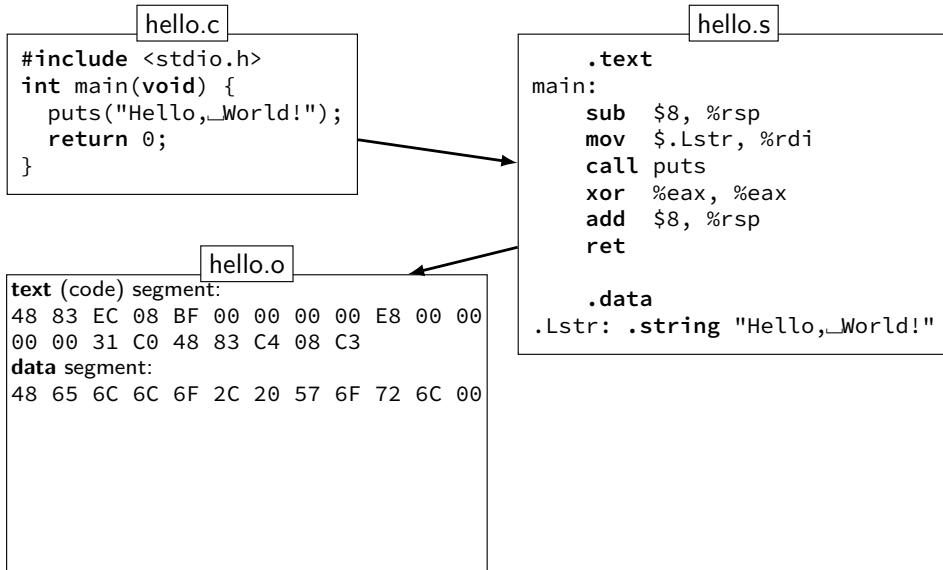
.data
.Lstr: .string "Hello, World!"
```

hello.o

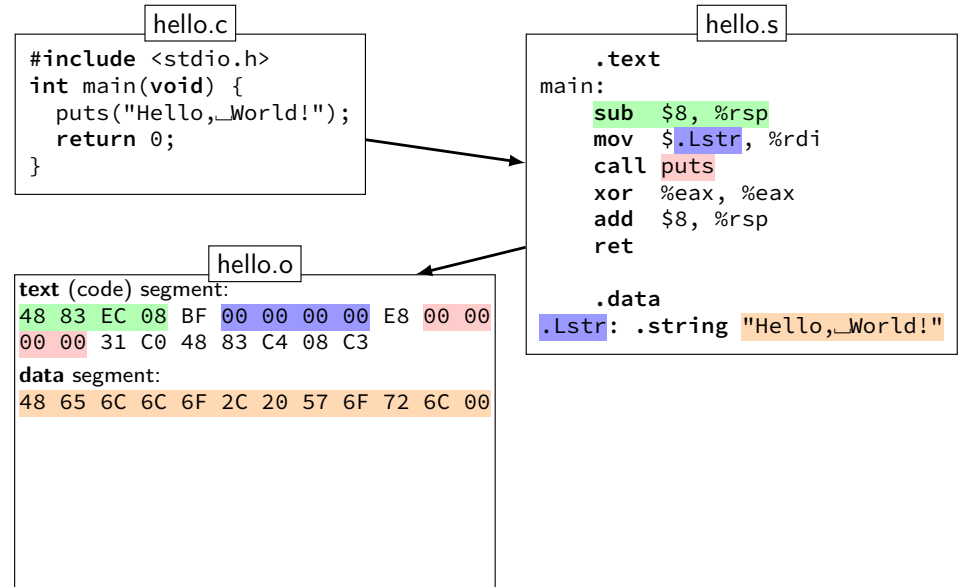
```
text (code) segment:
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

18

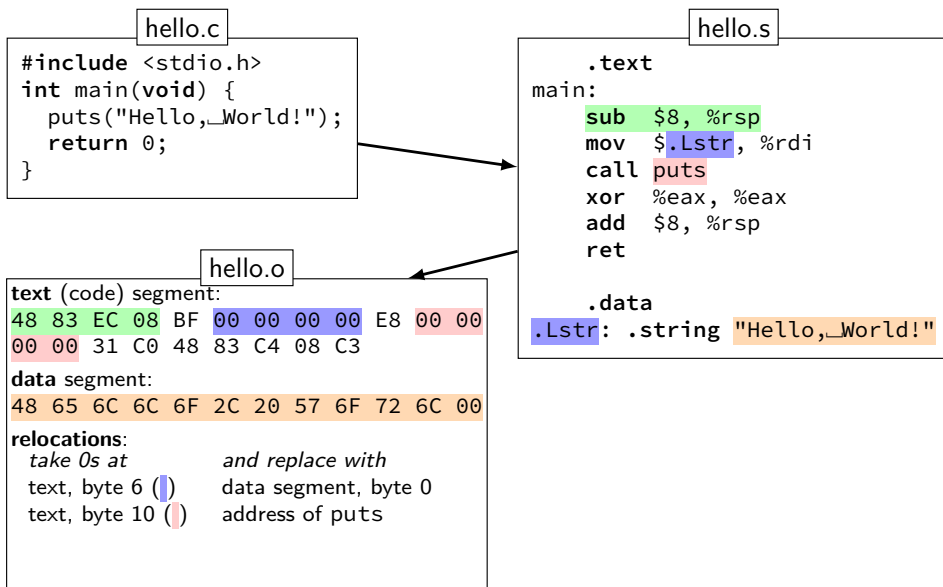
What's in those files?



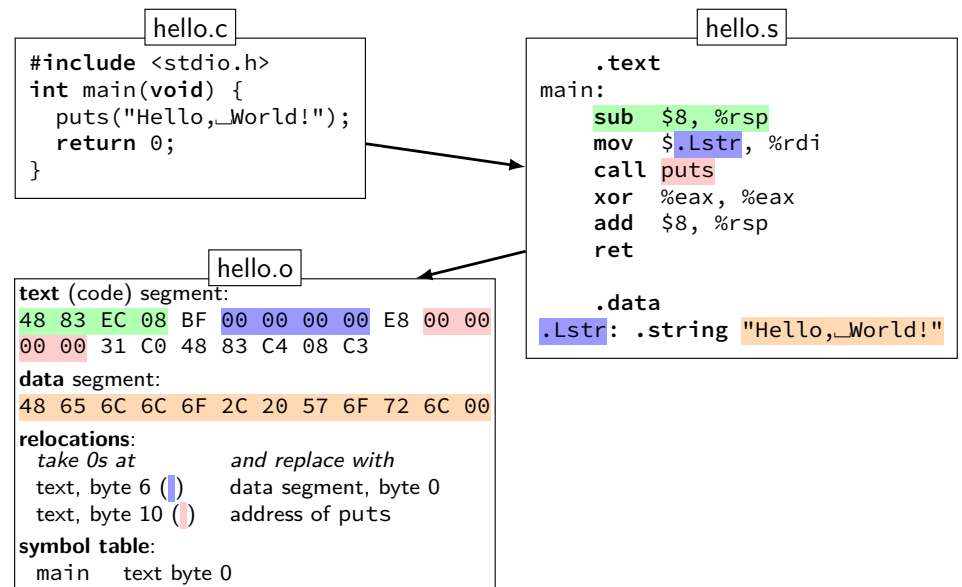
What's in those files?



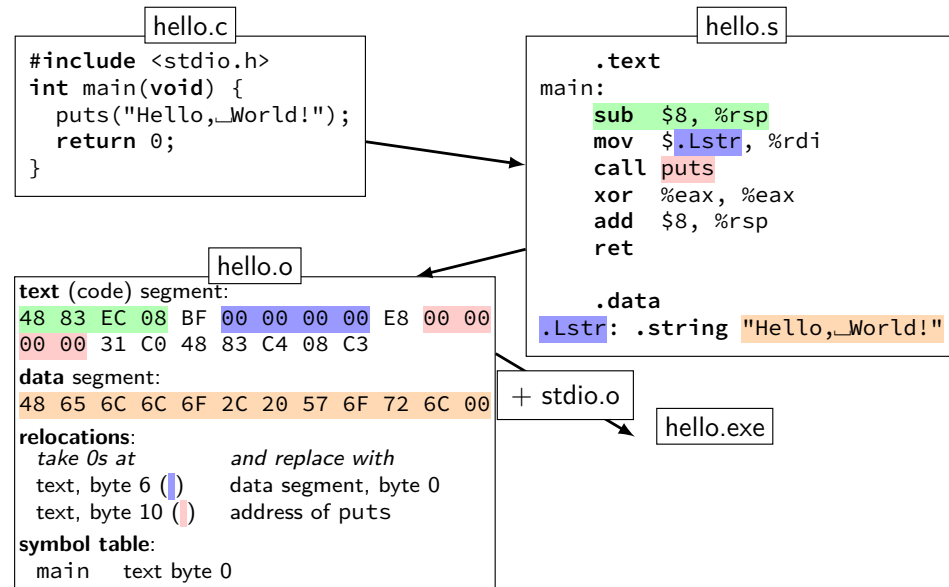
What's in those files?



What's in those files?

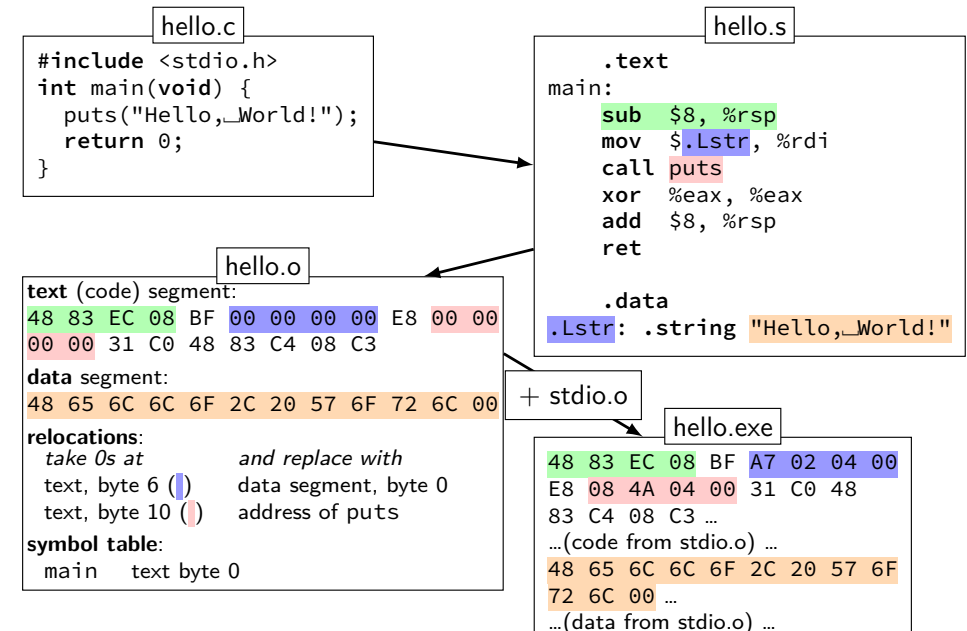


What's in those files?



18

What's in those files?



18

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)
    
```

19

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;
    
```

20

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;
```

20

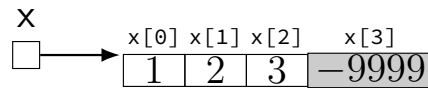
typedef struct (2)

```
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;
```

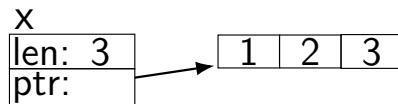
20

lists homework

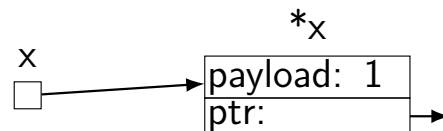
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```



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

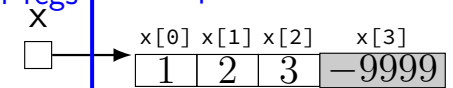


21

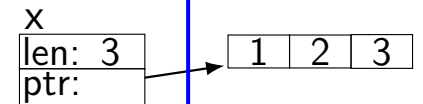
lists homework

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

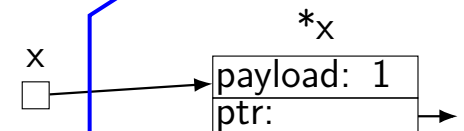
← on stack
or regs on heap →



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



21

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

22

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
```

23

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

24

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)

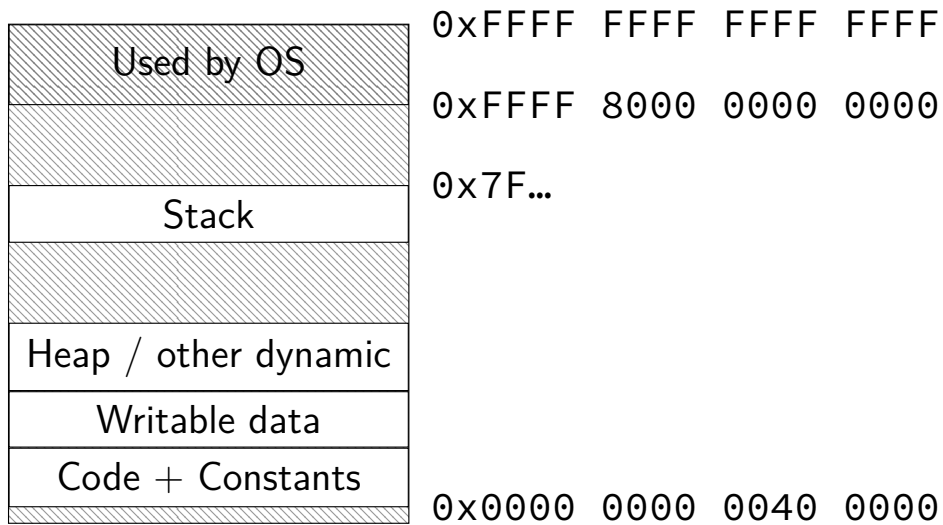
and/or scale (defaults to 1)

\$ means constant

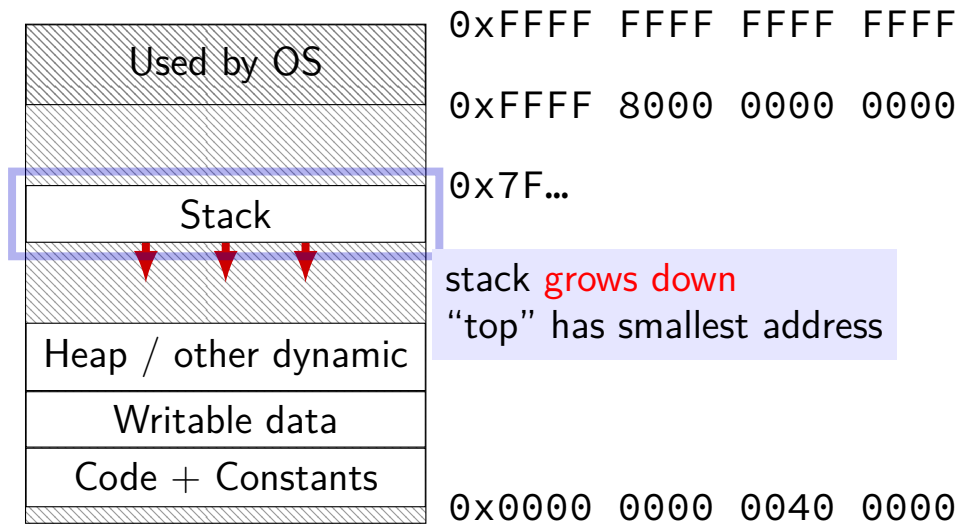
plain number/label means value in memory

25

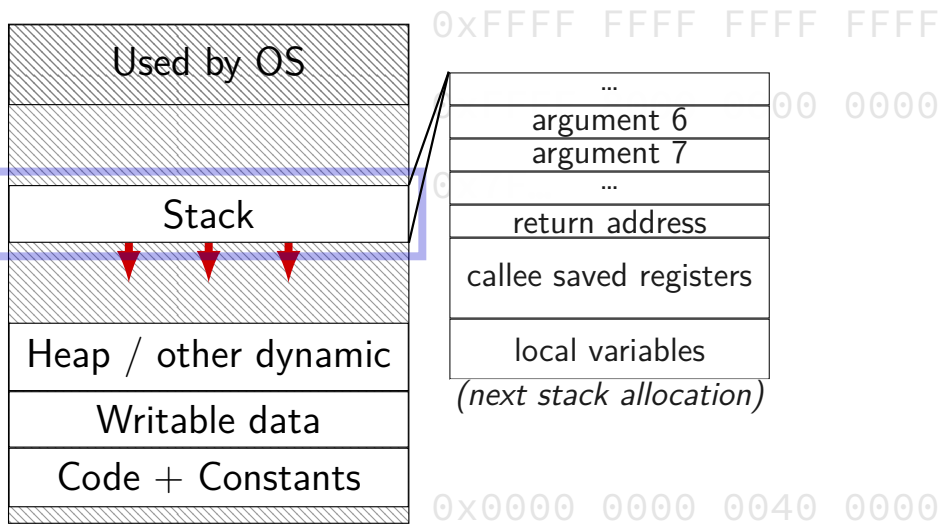
Program Memory (x86-64 Linux)



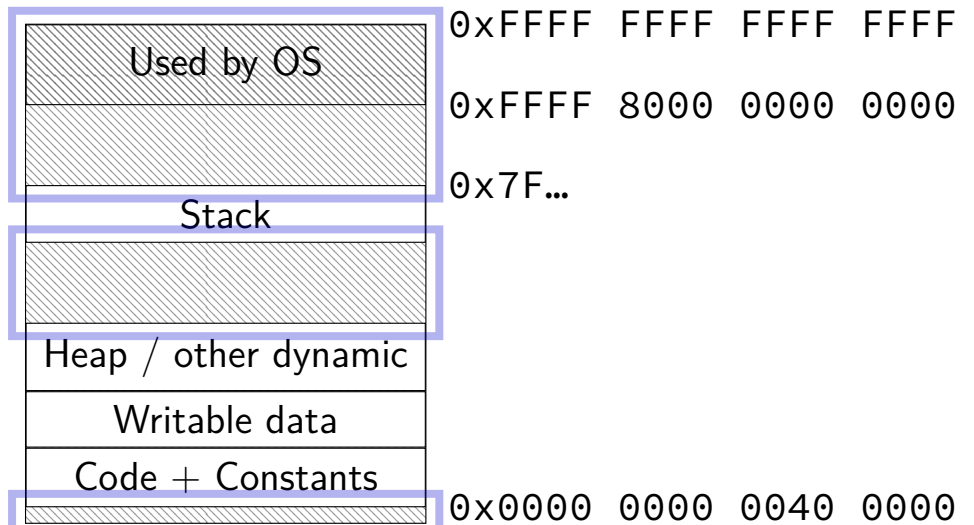
Program Memory (x86-64 Linux)



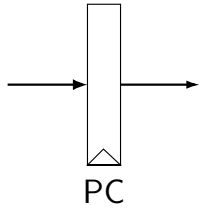
Program Memory (x86-64 Linux)



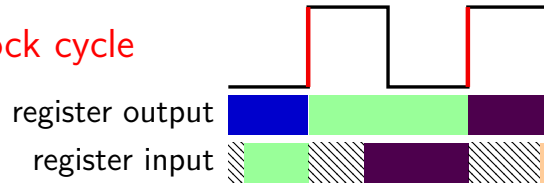
Program Memory (x86-64 Linux)



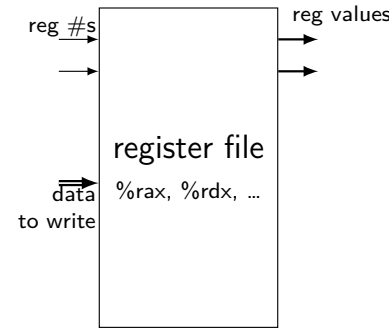
Registers



updates every **clock cycle**



Register file



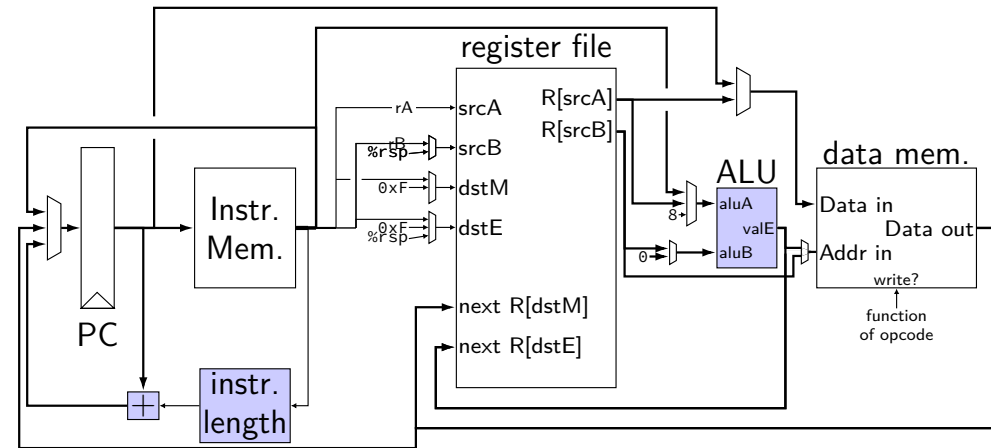
Stages and Time

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

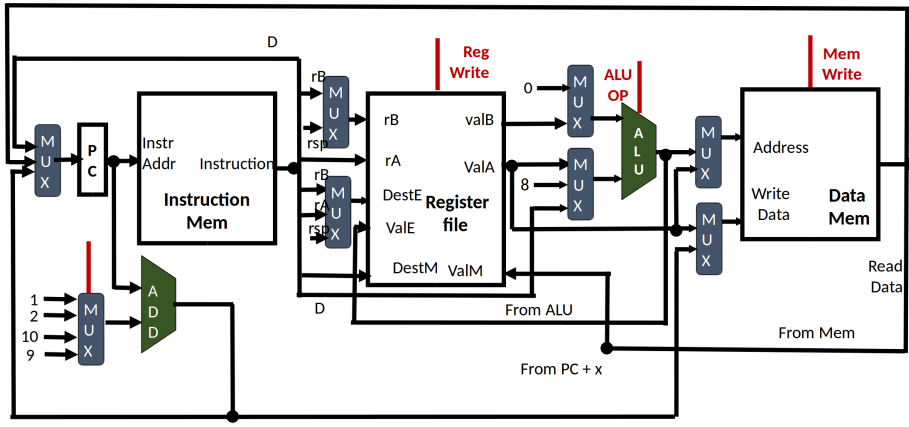
For the single-cycle design, **order** when these events happen pushq %rax instruction:

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



31

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

32

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

32

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

32

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

32

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

32

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

33

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

33

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()	OR	0	1
> 1		0	1
one()		1	1
> 1			

33

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()	OR	0	1
> 1		0	1
one()		1	1
> 1			

33

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()	OR	0	1
> 1		0	1
one()		1	1
> 1			

33

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

34

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
```

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

34

Right shift

Undefined: ~~$x \lll 1$~~ Instead: $x \gg 1$

```
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
```

35

Right shift

Undefined: ~~$x \lll 1$~~ Instead: $x \gg 1$

```
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
```

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

35

Shifts and negative numbers

$-10 \gg 1 == ???$ ($-10 = 1111 \dots 1111 0110$)

binary ?111 ... 1111 1011

36

Shifts and negative numbers

$-10 \gg 1 == ???$ ($-10 = 1111 \dots 1111 0110$)

binary **?**111 ... 1111 1011

Option 1: binary **1**111 ... 1011 =

$$-5 = -10 \times 2^{-k}$$

copy sign bit

Option 2: binary **0**111 ... 1011 = $2^{31} - 5$

always use zero

36

Shifts and negative numbers

$-10 \gg 1 == ???$ ($-10 = 1111 \dots 1111 0110$)

binary **?**111 ... 1111 1011

Option 1: binary **1**111 ... 1011 =

$$-5 = -10 \times 2^{-k}$$

copy sign bit

arithmetic

Option 2: binary **0**111 ... 1011 = $2^{31} - 5$

always use zero

logical

36

Shifts and negative numbers

$-10 \gg 1 == ???$ ($-10 = 1111 \dots 1111 0110$)

binary **?**111 ... 1111 1011

Option 1: binary **1**111 ... 1011 =

$$-5 = -10 \times 2^{-k}$$

copy sign bit

arithmetic

Option 2: binary **0**111 ... 1011 = $2^{31} - 5$

always use zero

logical

36

Typical RISC ISA properties

theme: simpler to implement

fewer, simpler instructions

separate instructions to access memory

fixed-length instructions

more registers

no “loops” within single instructions

no instructions with two memory operands

few addressing modes

37

Y86-64: Simple condition codes (1)

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

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

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

SF = 1 (negative)

ZF = 0 (not zero)

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

r10 becomes 1

SF = 0 (non-negative)

ZF = 0 (not zero)

38

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 \leq 0
l	SF = 1	value $<$ 0
e	ZF = 1	value = 0
ne	ZF = 0	value \neq 0
ge	SF = 0	value \geq 0
g	SF = 0 and ZF = 0	value $>$ 0

39