ISAs / Y86

last time

loops/switch to assembly

C types

pointer arithmetic

struct

printf

undefined behavior

logical right shift

arithmetic right shift and division

exercise

```
int shiftTwo(int x) {
    return x >> 2;
}
```

`shiftTwo(-6) = ???`

A. -4 B. -3 C. -2 D. -1 E. 0

F. some positive number G. something else

explanation

6 =	`000…00000110`
flip bits	`111…11111001`
add one	
-6 =	`111…11111010`
arithmetic shift by 2	**`11`**`111…1111111010`
	`111…111110` (-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 `1`s instead of `0`s
(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

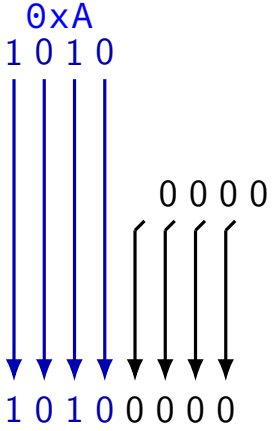
C division: rounds towards zero (truncate)

arithmetic shift: rounds towards negative infinity

solution: "bias" adjustments — described in textbook

```
divideBy8: // GCC generated code
    leal   7(%rdi), %eax // eax ← edi + 7
    testl  %edi, %edi    // set cond. codes based on %edi
    cmovns %edi, %eax    // if (SF = 0) eax ← edi
    sarl   $3, %eax      // arithmetic shift
```

multiplying by 16



0xA $\times$ 16 $=$ 0xA0

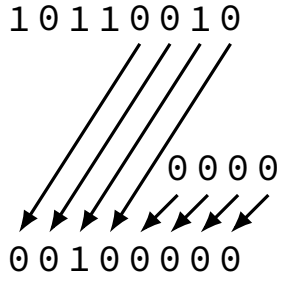
shift left

~~`shr $-4, %reg`~~

instead: `shl $4, %reg` ("**sh**ift **l**eft")

~~`value >> (-4)`~~

instead: `value << 4`



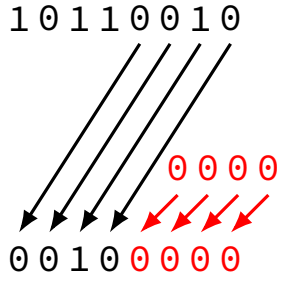
shift left

~~`shr $-4, %reg`~~

instead: `shl $4, %reg` ("**sh**ift **l**eft")

~~`value >> (-4)`~~

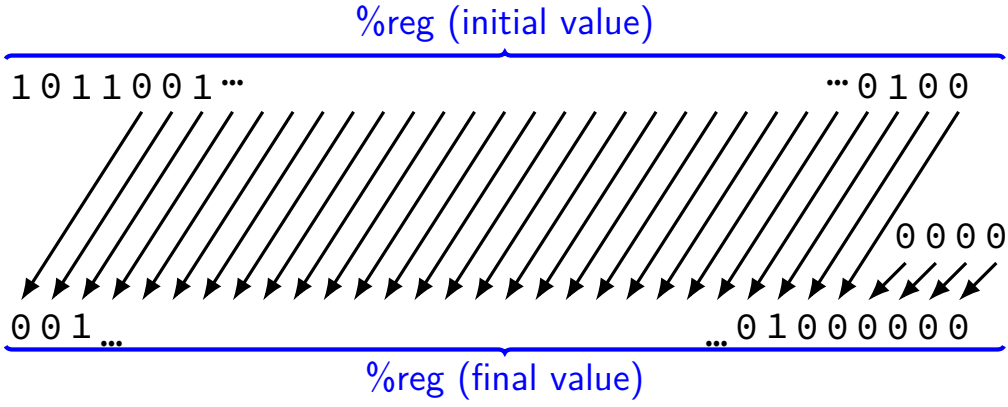
instead: `value << 4`



shift left

x86 instruction: `shl` — shift left

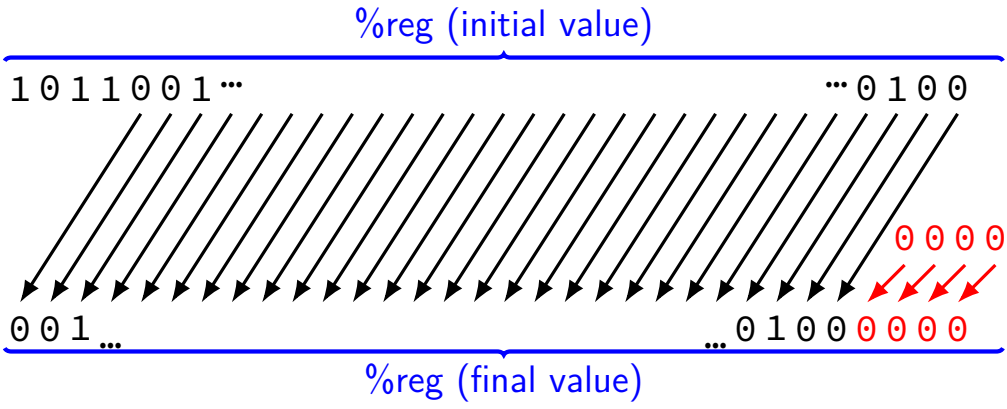
`shl` `$`*`amount`*`, %reg` (or variable: `shl %cl, %reg`)



shift left

x86 instruction: `shl` — shift left

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



left shift in math

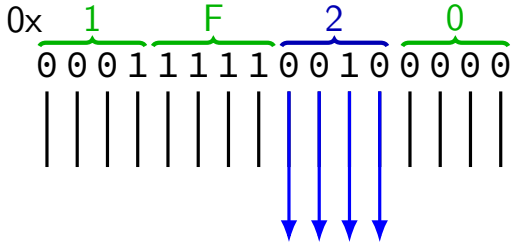
`1 << 0 == 1`	`0000 0001`
`1 << 1 == 2`	`0000 0010`
`1 << 2 == 4`	`0000 0100`
`10 << 0 == 10`	`0000 1010`
`10 << 1 == 20`	`0001 0100`
`10 << 2 == 40`	`0010 1000`
`-10 << 0 == -10`	`1111 0110`
`-10 << 1 == -20`	`1110 1100`
`-10 << 2 == -40`	`1101 1000`

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`
`−10 << 0 == −10`	`1111 0110`
`−10 << 1 == −20`	`1110 1100`
`−10 << 2 == −40`	`1101 1000`

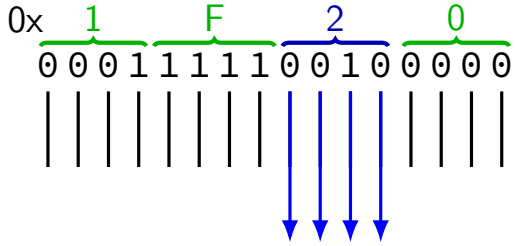
$$x \texttt{ << } y = x \times 2^y$$

extracting nibble from more




```
unsigned
extract_2nd(unsigned value) {
    return ???;
}
```

extracting nibble from more




```
unsigned
extract_2nd(unsigned value) {
    return ???;
}
```

```
// % -- remainder
unsigned extract_second_nibble(unsigned value) {
    return (value / 16) % 16;
}

unsigned extract_second_nibble(unsigned value) {
    return (value % 256) / 16;
}
```

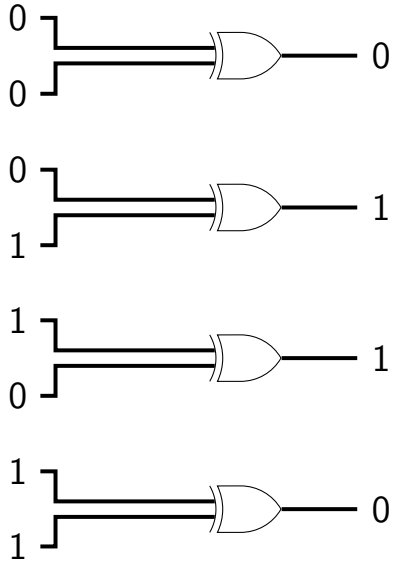
manipulating bits?

easy to manipulate individual bits in HW

- separate wire for each bit
- just ignore/select wires you care about

how do we expose that to software?

circuits: gates



interlude: a truth table

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

interlude: a truth table

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

AND with 1: keep a bit the same

interlude: a truth table

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

AND with 1: keep a bit the same

AND with 0: clear a bit

interlude: a truth table

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

AND with 1: keep a bit the same

AND with 0: clear a bit

method: construct "mask" of what to keep/remove

bitwise AND — &

Treat value as array of bits

```
1 & 1 == 1

1 & 0 == 0

0 & 0 == 0

2 & 4 == 0

10 & 7 == 2
```

bitwise AND — &

Treat value as array of bits

```
1 & 1 == 1

1 & 0 == 0

0 & 0 == 0

2 & 4 == 0

10 & 7 == 2
```

```
    … 0 0 1 0
&   … 0 1 0 0
  ---------
    … 0 0 0 0
```

bitwise AND — &

Treat value as array of bits

```
1 & 1 == 1

1 & 0 == 0

0 & 0 == 0

2 & 4 == 0

10 & 7 == 2
```

```
    … 0 0 1 0
&   … 0 1 0 0
  ---------
    … 0 0 0 0
```

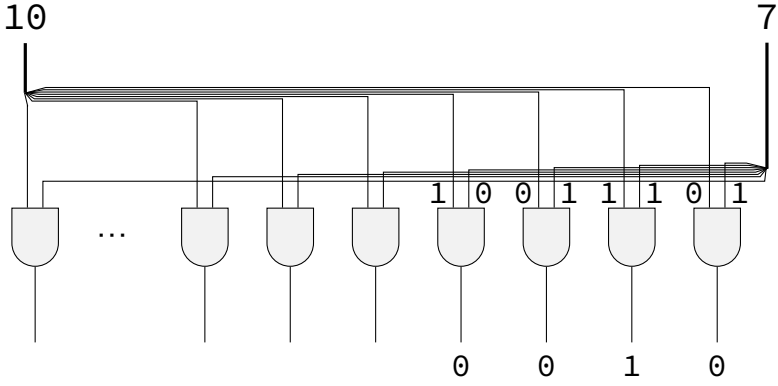
```
    … 1 0 1 0
&   … 0 1 1 1
  ---------
    … 0 0 1 0
```

bitwise AND — C/assembly

x86: `and %reg, %reg`

C: `foo & bar`

bitwise hardware (`10 & 7 == 2`)



extract 0x3 from 0x1234

```
unsigned get_second_nibble1(unsigned value) {
    return (value >> 4) & 0xF; // 0xF: 00001111
    // like (value / 16) % 16
}
```

aaaabbbbccccdddd → aaaabbbbcccc → 00000000cccc

```
unsigned get_second_nibble2(unsigned value) {
    return (value & 0xF0) >> 4; // 0xF0: 11110000
    // like (value % 256) / 16;
}
```

aaaabbbbccccdddd → 000000000cccc0000 → 00000000cccc

extract 0x3 from 0x1234

```
get_second_nibble1_bitwise:
    movl %edi, %eax
    shrl $4, %eax
    andl $0xF, %eax
    ret

get_second_nibble2_bitwise:
    movl %edi, %eax
    andl $0xF0, %eax
    shrl $4, %eax
    ret
```

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	**false**	**true**
false	false	true
true	true	true

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	**false**	**true**
false	false	true
true	true	true

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	**false**	**true**
false	false	true
true	true	true

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	**false**	**true**
false	false	true
true	true	true

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	**false**	**true**
false	false	true
true	true	true

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

AND	**false**	**true**
false	false	false
true	false	true

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	**false**	**true**
false	false	false
true	false	true

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	**false**	**true**
false	false	false
true	false	true

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	**false**	**true**
false	false	false
true	false	true

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

AND	**false**	**true**
false	false	false
true	false	true

and/or/xor

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

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

XOR	**0**	**1**
0	0	1
1	1	0

&

|

^

conditionally clear bit
conditionally keep bit

conditionally set bit

conditionally flip bit

bitwise OR — |

```
1 | 1 == 1

1 | 0 == 1

0 | 0 == 0

2 | 4 == 6

10 | 7 == 15
```

```
    … 1 0 1 0
|   … 0 1 1 1
-------------
    … 1 1 1 1
```

bitwise xor — ^

```
1 ^ 1 == 0

1 ^ 0 == 1

0 ^ 0 == 0

2 ^ 4 == 6

10 ^ 7 == 13
```

```
    … 1 0 1 0
^   … 0 1 1 1
  ------------
    … 1 1 0 1
```

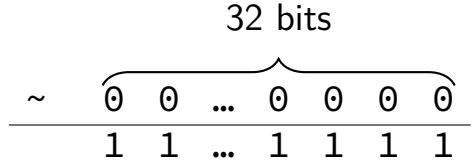
negation / not — ~

~ ('complement') is bitwise version of `!`:

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)
```



negation / not — ~

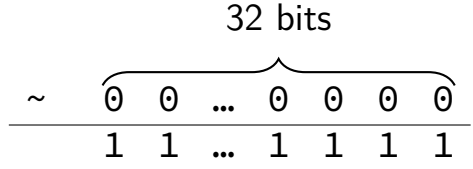
~ ('complement') is bitwise version of !:

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)

~2 == (int) 0xFFFFFFFD (aka −3)
```



negation / not — ~

~ ('complement') is bitwise version of `!`:

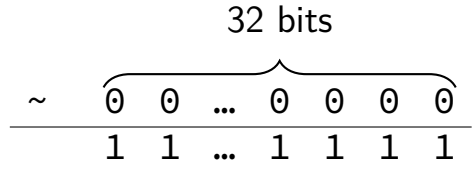
```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF (aka −1)

~2 == (int) 0xFFFFFFFD (aka −3)

~((unsigned) 2) == 0xFFFFFFFD
```



bit-puzzles

future assignment

bit manipulation puzzles

solve some problem with bitwise ops

- maybe that you could do with normal arithmetic, comparisons, etc.

why?

- good for thinking about HW design
- good for understanding bitwise ops
- unreasonably common interview question type

note: ternary operator

```
w = (x ? y : z)
```

```
if (x) { w = y; } else { w = z; }
```

one-bit ternary

`(x ? y : z)`

constraint: *x, y, and z are 0 or 1*

now: reimplement in C without if/else/||/etc.

(assembly: no jumps probably)

one-bit ternary

```
(x ? y : z)
```

constraint: *x, y, and z are 0 or 1*

now: reimplement in C without if/else/||/etc.

(assembly: no jumps probably)

divide-and-conquer:

```
(x ? y : 0)
(x ? 0 : z)
```

one-bit ternary parts (1)

constraint: *x, y, and z are 0 or 1*

```
(x ? y : 0)
```

one-bit ternary parts (1)

constraint: *x, y, and z are 0 or 1*

```
(x ? y : 0)
```

	y=0	**y=1**
x=0	0	0
x=1	0	1

→ `(x & y)`

one-bit ternary parts (2)

`(x ? y : 0)` = `(x & y)`

one-bit ternary parts (2)

`(x ? y : 0)` = `(x & y)`

`(x ? 0 : z)`

opposite `x`: `~x`

`((~x) & z)`

one-bit ternary

constraint: *x, y, and z are 0 or 1*

```
(x ? y : z)
```

```
(x ? y : 0) | (x ? 0 : z)
```

```
(x & y) | ((~x) & z)
```

multibit ternary

constraint: x *is 0 or 1*

old solution `((x & y) | (~x) & z)` only gets least sig. bit

`(x ? y : z)`

multibit ternary

constraint: x *is 0 or 1*

old solution `((x & y) | (~x) & z)` only gets least sig. bit

```
(x ? y : z)
```

```
(x ? y : 0) | (x ? 0 : z)
```

constructing masks

constraint: x *is 0 or 1*

`(x ? y : 0)`

if x = 1: want `1111111111…1` (keep y)

if x = 0: want `0000000000…0` (want `0`)

constructing masks

constraint: x *is 0 or 1*

`(x ? y : 0)`

if x = 1: want `1111111111…1` (keep y)

if x = 0: want `0000000000…0` (want `0`)

a trick: −x (`-1` is `1111…1`)

constructing masks

constraint: x *is 0 or 1*

`(x ? y : 0)`

if x = 1: want `1111111111…1` (keep y)

if x = 0: want `0000000000…0` (want `0`)

a trick: −x (`-1` is `1111…1`)

```
((-x) & y)
```

constructing other masks

constraint: x *is 0 or 1*

`(x ? 0 : z)`

if x = ~~1~~ 0: want `1111111111…1`

if x = ~~0~~ 1: want `0000000000…0`

mask: ~~>x~~

constructing other masks

constraint: x *is 0 or 1*

`(x ? 0 : z)`

if x = 1 0: want `1111111111…1`

if x = 0 1: want `0000000000…0`

mask: ~~x~~ `-(x^1)`

multibit ternary

constraint: x *is 0 or 1*

old solution `((x & y) | (~x) & z)` only gets least sig. bit

```
(x ? y : z)
```

```
(x ? y : 0) | (x ? 0 : z)
```

```
((−x) & y) | ((−(x ^ 1)) & z)
```

fully multibit

~~constraint: x is 0 or 1~~

```
(x ? y : z)
```

fully multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

easy C way: `!x` = 0 or 1, `!!x` = 0 or 1

- x86 assembly: `testq %rax, %rax` then `sete`/`setne` (copy from ZF)

fully multibit

~~constraint: x is 0 or 1~~

`(x ? y : z)`

easy C way: `!x` = 0 or 1, `!!x` = 0 or 1

- x86 assembly: `testq %rax, %rax` then `sete`/`setne` (copy from ZF)

`(x ? y : 0) | (x ? 0 : z)`

`((-!!x) & y) | ((-!x) & z)`

simple operation performance

typical modern desktop processor:

- bitwise and/or/xor, shift, add, subtract, compare — $\sim$ 1 cycle
- integer multiply — $\sim$ 1-3 cycles
- integer divide — $\sim$ 10-150 cycles

(smaller/simpler/lower-power processors are different)

simple operation performance

typical modern desktop processor:

- bitwise and/or/xor, shift, add, subtract, compare — $\sim$ 1 cycle
- integer multiply — $\sim$ 1-3 cycles
- integer divide — $\sim$ 10-150 cycles

(smaller/simpler/lower-power processors are different)

add/subtract/compare are more complicated in hardware!

but *much* more important for typical applications

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

how do we solve is x is two bits? four bits?

problem: any-bit

is any bit of x set?

goal: turn 0 into 0, not zero into 1

easy C solution: `!(!(x))`

another easy solution if you have − or + (lab exercise)

what if we don't have ! or − or +

how do we solve is x is two bits? four bits?

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

wasted work (1)

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

in general: `(x & 1) | (y & 1) == (x | y) & 1`

wasted work (1)

```
((x & 1) | ((x >> 1) & 1) | ((x >> 2) & 1) | ((x >> 3) & 1))
```

in general: `(x & 1) | (y & 1) == (x | y) & 1`

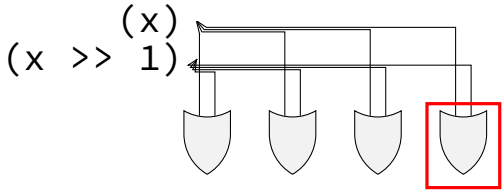
```
(x | (x >> 1) | (x >> 2) | (x >> 3)) & 1
```

wasted work (2)

4-bit any set: `(x | (x >> 1)| (x >> 2) | (x >> 3)) & 1`

performing 3 bitwise ors

...each bitwise or does 4 OR operations



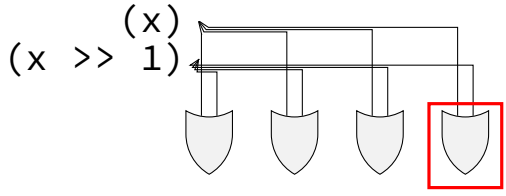
wasted work (2)

4-bit any set: `(x | (x >> 1)| (x >> 2) | (x >> 3)) & 1`

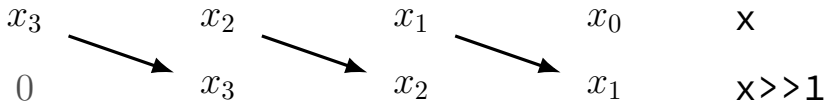
performing 3 bitwise ors

...each bitwise or does 4 OR operations

but only result of one of the 4!

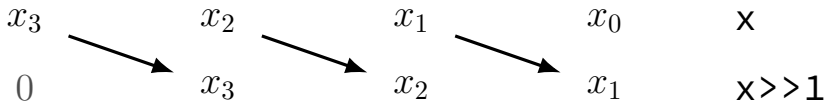


any-bit: looking at wasted work



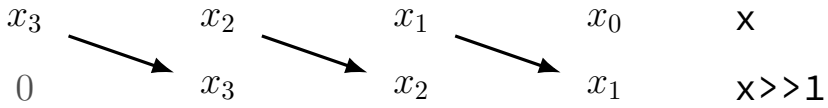

```
y=(x|x>>1)
```

any-bit: looking at wasted work



$(0|x_3)$ $(x_3|x_2)$ $(x_2|x_1)$ $(x_1|x_0)$ `y=(x|x>>1)`

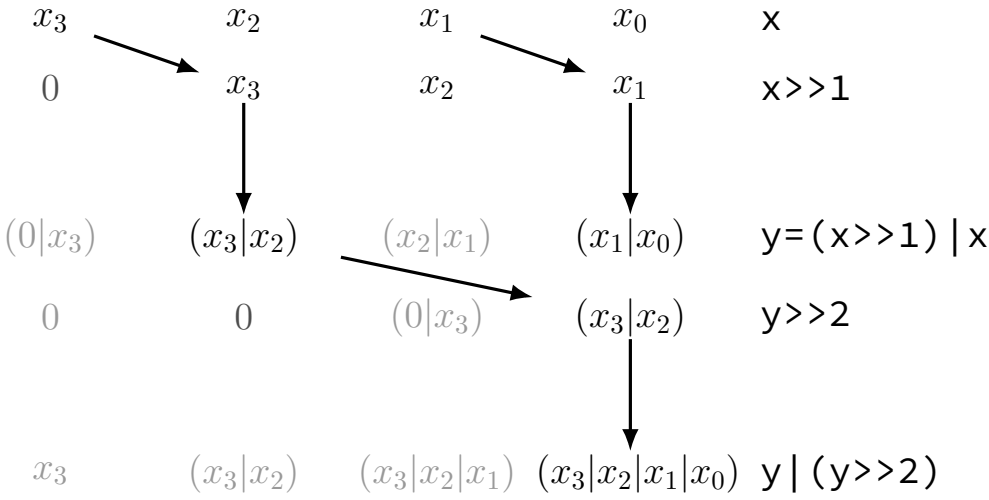
any-bit: looking at wasted work



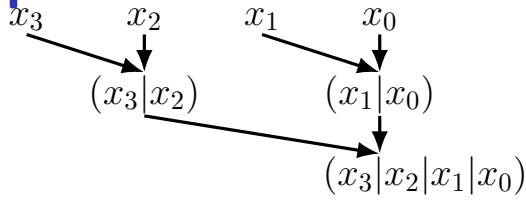
$(0|x_3)$ $(x_3|x_2)$ $(x_2|x_1)$ $(x_1|x_0)$ y=(x|x>>1)

final value wanted: $x_3|x_2|x_1|x_0$
previously:
compute x|(x>>1) for $x_1|x_0$;
(x>>2)|(x>>3) for $x_3|x_2$
observation: got both parts with just x|(x>>1)

any-bit: divide and conquer



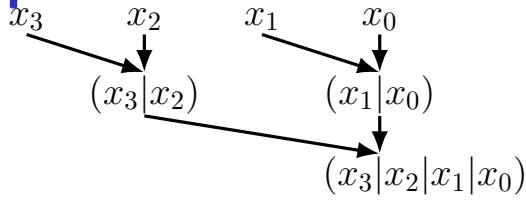
any-bit: divide and conquer



four-bit input $x = x_3x_2x_1x_0$

`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

any-bit: divide and conquer



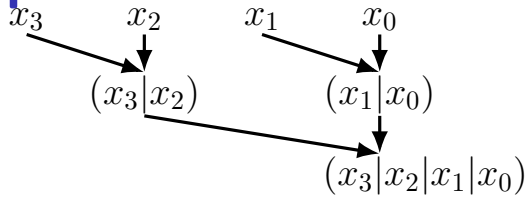
four-bit input $x = x_3x_2x_1x_0$

`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`y | (y >> 2)` $= (y_1|0)(y_2|0)(y_3|y_1)(y_4|y_2) = z_1z_2z_3z_4$

$z_4 = (y_4|y_2) = ((x_2|x_3)|(x_0|x_1)) = x_0|x_1|x_2|x_3$ "is any bit set?"

any-bit: divide and conquer



four-bit input $x = x_3x_2x_1x_0$

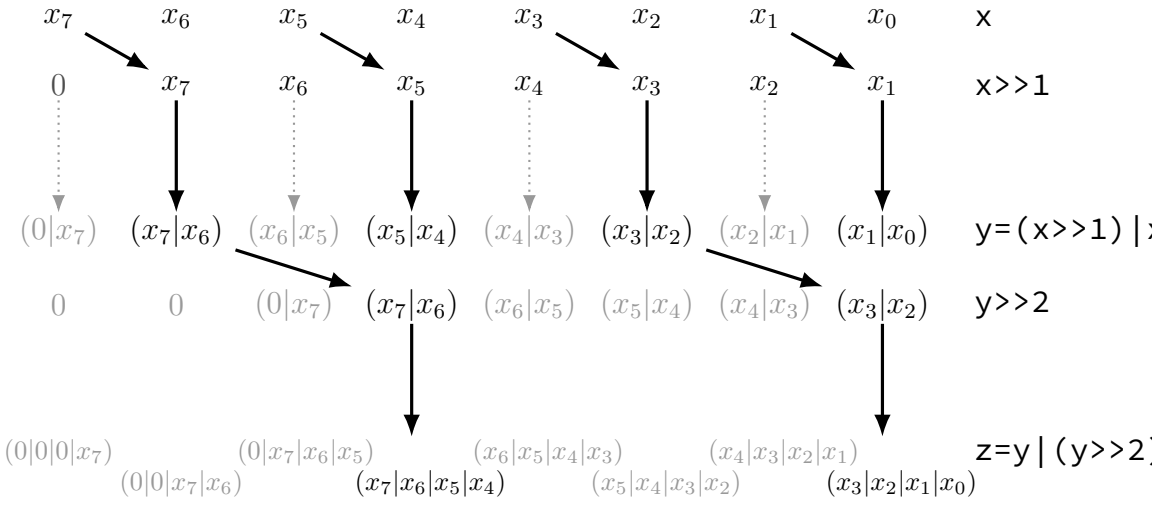
`x | (x >> 1)` $= (x_3|0)(x_2|x_3)(x_1|x_2)(x_0|x_1) = y_1y_2y_3y_4$

`y | (y >> 2)` $= (y_1|0)(y_2|0)(y_3|y_1)(y_4|y_2) = z_1z_2z_3z_4$

$z_4 = (y_4|y_2) = ((x_2|x_3)|(x_0|x_1)) = x_0|x_1|x_2|x_3$ "is any bit set?"

```
unsigned int any_of_four(unsigned int x) {
    int part_bits = (x >> 1) | x;
    return ((part_bits >> 2) | part_bits) & 1;
}
```

any-bit: divide and conquer



any-bit-set: 32 bits

```
unsigned int any(unsigned int x) {
    x = (x >> 1) | x;
    x = (x >> 2) | x;
    x = (x >> 4) | x;
    x = (x >> 8) | x;
    x = (x >> 16) | x;
    return x & 1;
}
```

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)
```

exercise

Which of these will swap last and second-to-last bit of an `unsigned int` x? (bits $uvwxyz$ become $uvwxzy$)

```
/* 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));
    //      ^^^^^^^^^^^^^^
    //       uvwxyz --> 0uvwxy -> 00000y

    //                          ^^^^^^^^^^
    //       uvwxyz --> uvwxy0

    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //       00000y | uvwxy0 = uvwxyy
```

version B

```
/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));
    //      ^^^^^^^^^^^^^^
    //       uvwxyz --> 0uvwxy --> 00000y

    //                          ^^^^^^^^^^^^^^^
    //       uvwxyz --> vwxyz0 --> vwxy00

    //                                          ^^^^^^^^^
    //       uvwxyz -->             uvwx00
```

version C

```
/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);
    //      ^^^^^^^^^^
    //       uvwxyz -->             uvwx00

    //                    ^^^^^^^^^^^^^^
    //       uvwxyz --> 00000z --> 0000z0

    //                                    ^^^^^^^^^^^^
    //       uvwxyz --> 0uvwxy --> 00000y
```

version D

```
/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
    //      ^^^^^^^^^^^^^^^
    //       uvwxyz --> 00000z --> 0000z0

    //                          ^^^^^^^^^^^^^^
    //       uvwxyz --> 0000yz --> 00000y

    //       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //       0000zy ^ uvwxyz --> uvwx(z XOR y)(y XOR z)
```

expanded code

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

ISAs being manufactured today

(ISA = instruction set architecture)

x86 — dominant in desktops, servers

ARM — dominant in mobile devices

POWER — Wii U, IBM supercomputers and some servers

MIPS — common in consumer wifi access points

SPARC — some Oracle servers, Fujitsu supercomputers

z/Architecture — IBM mainframes

Z80 — TI calculators

SHARC — some digital signal processors

RISC V — some embedded

...

microarchitecture v. instruction set

microarchitecture — design of the hardware

- "generations" of Intel's x86 chips
- different microarchitectures for very low-power versus laptop/desktop
- changes in performance/efficiency

instruction set — interface visible by software

- what matters for software compatibility
- many ways to implement (but some might be easier)

ISA variation

instruction set	instr. length	# normal registers	*approx.* # instrs.
x86-64	1–15 byte	16	1500
Y86-64	1–10 byte	15	18
ARMv7	4 byte*	16	400
POWER8	4 byte	32	1400
MIPS32	4 byte	31	200
Itanium	41 bits*	128	300
Z80	1–4 byte	7	40
VAX	1–14 byte	8	150
z/Architecture	2–6 byte	16	1000
RISC V	4 byte*	31	500*

other choices: condition codes?

instead of:

```
cmpq %r11, %r12
je somewhere
```

could do:

```
/* _B_ranch if _EQ_ual */
beq  %r11, %r12, somewhere
```

other choices: addressing modes

ways of specifying operands. examples:

x86-64: `10(%r11,%r12,4)`

ARM: `%r11 << 3` (shift register value by constant)

VAX: `((%r11))` (register value is pointer to pointer)

other choices: number of operands

`add src1, src2, dest`

ARM, POWER, MIPS, SPARC, ...

`add src2, src1=dest`

x86, AVR, Z80, ...

VAX: both

other choices: instruction complexity

instructions that write multiple values?

- ARM: `ldm` (load multiple), `stm` (store multiple), `push`/`pop`, ...
- x86-64: `push`/`pop`, `movs` (move string to string), `stos` (store string), ...

more?

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC and RISC

RISC — Reduced Instruction Set Computer

reduced from what?

CISC — Complex Instruction Set Computer

some VAX instructions

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

`POLY` *x*, *coefficientsLen*, *coefficientsPtr*
Evaluate the polynomial whose coefficients are pointed to by *coefficientPtr* at the value *x*.

`EDITPC` *sourceLen*, *sourcePtr*, *patternLen*, *patternPtr*
Edit the string pointed to by *sourcePtr* using the pattern string specified by *patternPtr*.

microcode

`MATCHC` *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

loop in hardware???

typically: lookup sequence of microinstructions (“microcode”)

secret simpler instruction set

Why RISC?

complex instructions were usually not faster

complex instructions were harder to implement

compilers, not hand-written assembly

Why RISC?

complex instructions were usually not faster

complex instructions were harder to implement

compilers, not hand-written assembly

assumption: okay to require compiler modifications

typical RISC ISA properties

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

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

- choose instructions with particular assembly language in mind?
- expose logical program operations to hardware for optimization?
- ...but more resources spent on making hardware correct?
- easier to specialize for particular applications
- less work for compilers

RISC-like (easier to make hardware, harder to use assembly)

- choose instructions with particular HW implementation in mind?
- expose individual hardware operations to compiler for optimization?
- simpler to build/test hardware
- ...so more resources spent on making hardware fast?
- more work for compilers

ISAs: who does the work?

CISC-like (harder to make hardware, easier to use assembly)

- choose instructions with particular assembly language in mind?
- expose logical program operations to hardware for optimization?
- ...but more resources spent on making hardware correct?
- easier to specialize for particular applications
- less work for compilers

RISC-like (easier to make hardware, harder to use assembly)

- choose instructions with particular HW implementation in mind?
- expose individual hardware operations to compiler for optimization?
- simpler to build/test hardware
- ...so more resources spent on making hardware fast?
- more work for compilers

ISAs: who does the work?

CISC-like

less work for assembly-writers

more work for hardware

choose assembly, design instructions?

harder to build/test CPU

design new instrs for target apps?

RISC-like

more work for assembly-writers

less work for hardware

design for particular kind of HW?

easier to build/test CPU

spend more time optimizing HW?

is CISC the winner? (1)

well, can't get rid of x86 features

- backwards compatibility matters

now more application-specific instructions

- but usually exposing specialized hardware — not "loops in hardware"

but...compilers tend to use more RISC-like subset of instructions

modern x86: often convert to RISC-like "microinstructions"

- sounds really expensive, but ...
- lots of instruction preprocessing used in 'fast' CPU designs (even for RISC ISAs)

is CISC the winner? (2)

some early RISC designs: compiler change with CPU version

- instruction set included hardware limitations

in practice: we expect old code to continue working

in practice: new CPU must be faster with old code

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64 instruction set

based on x86

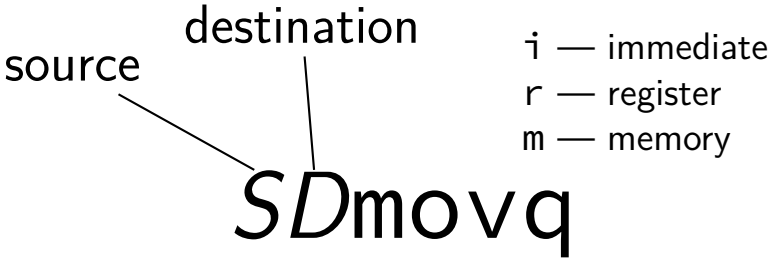
omits most of the 1000+ instructions

leaves

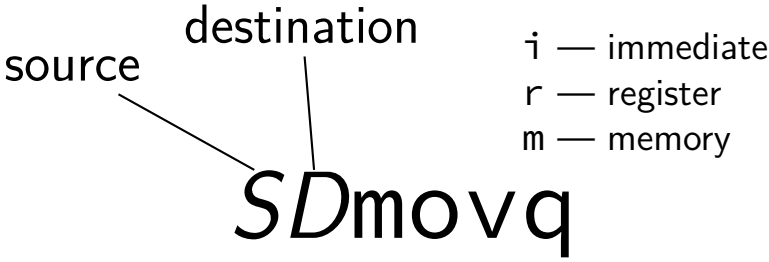
`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64: movq



Y86-64: movq

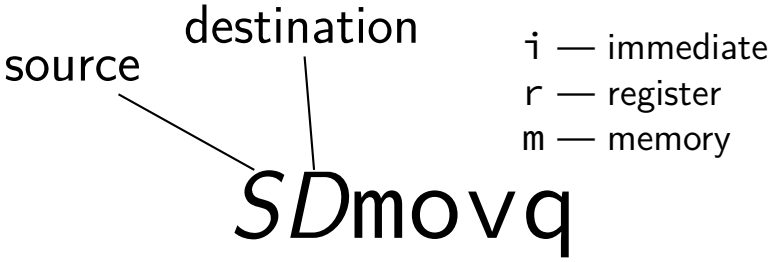


irmovq ~~immovq~~ ~~iimovq~~

rrmovq rmmovq ~~rimovq~~

mrmovq ~~mmmovq~~ ~~mimovq~~

Y86-64: movq



irmovq

~~immovq~~

rrmovq

rmmovq

mrmovq

~~mmmovq~~

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

cmovCC

conditional move

exist on x86-64 (but you probably didn't see them)

Y86-64: register-to-register only

instead of:

```
    jle skip_move
    rrmovq %rax, %rbx
skip_move:
    // ...
```

can do:

```
    cmovg %rax, %rbx
```

halt

(x86-64 instruction called `hlt`)

Y86-64 instruction `halt`

stops the processor

otherwise — something's in memory "after" program!

real processors: reserved for OS

Y86-64: specifying addresses

Valid: `rmmovq %r11, 10(%r12)`

Y86-64: specifying addresses

Valid: `rmmovq %r11, 10(%r12)`

Invalid: `rmmovq %r11, 10(%r12,%r13)`

Invalid: `rmmovq %r11, 10(,%r12,4)`

Invalid: `rmmovq %r11, 10(%r12,%r13,4)`

Y86-64: accessing memory (1)

r12 $\leftarrow$ memory[10 + r11] + r12

Invalid: ~~`addq 10(%r11), %r12`~~

Y86-64: accessing memory (1)

r12 $\leftarrow$ memory[10 + r11] + r12

Invalid: ~~`addq 10(%r11), %r12`~~

Instead:

```
mrmovq 10(%r11), %r11
/* overwrites %r11 */

addq %r11, %r12
```

Y86-64: accessing memory (2)

r12 ← memory[10 + 8 * r11] + r12

Invalid: ~~`addq 10(,%r11,8), %r12`~~

Y86-64: accessing memory (2)

r12 ← memory[10 + 8 * r11] + r12

Invalid: ~~`addq 10(,%r11,8), %r12`~~

Instead:

```
/* replace %r11 with 8*%r11 */
addq %r11, %r11
addq %r11, %r11
addq %r11, %r11

mrmovq 10(%r11), %r11
addq %r11, %r12
```

Y86-64 constants (1)

```
irmovq $100, %r11
```

only instruction with non-address constant operand

Y86-64 constants (2)

r12 $\leftarrow$ r12 + 1

Invalid: ~~`addq $1, %r12`~~

Y86-64 constants (2)

r12 $\leftarrow$ r12 + 1

Invalid: ~~`addq $1, %r12`~~

Instead, need an extra register:

```
irmovq $1, %r11
addq %r11, %r12
```

Y86-64: operand uniqueness

only one kind of value for each operand

instruction name tells you the kind

(why `movq` was 'split' into four names)

Y86-64: condition codes

ZF — value was zero?

SF — sign bit was set? i.e. value was negative?

this course: no OF, CF (to simplify assignments)

set by `addq`, `subq`, `andq`, `xorq`

not set by anything else

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

missing `OF` (overflow flag); `CF` (carry flag)

Y86-64: conditionals (1)

~~`cmp`~~, ~~`test`~~

Y86-64: conditionals (1)

~~`cmp`~~, ~~`test`~~

instead: use side effect of normal arithmetic

Y86-64: conditionals (1)

~~`cmp`~~, ~~`test`~~

instead: use side effect of normal arithmetic

instead of

```
cmpq %r11, %r12
jle somewhere
```

maybe:

```
subq %r11, %r12
jle
```

(but changes `%r12`)

push/pop

`pushq %rbx`

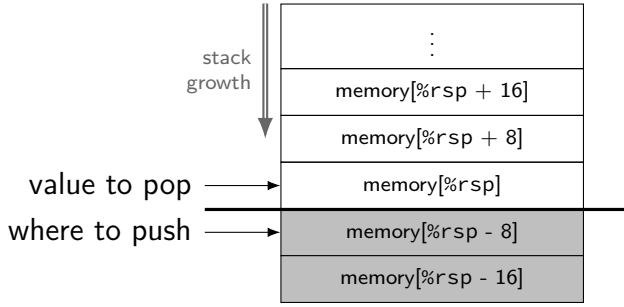
`%rsp` $\leftarrow$ `%rsp` $- 8$

memory[`%rsp`] $\leftarrow$ `%rbx`

`popq %rbx`

`%rbx` $\leftarrow$ memory[`%rsp`]

`%rsp` $\leftarrow$ `%rsp` $+ 8$



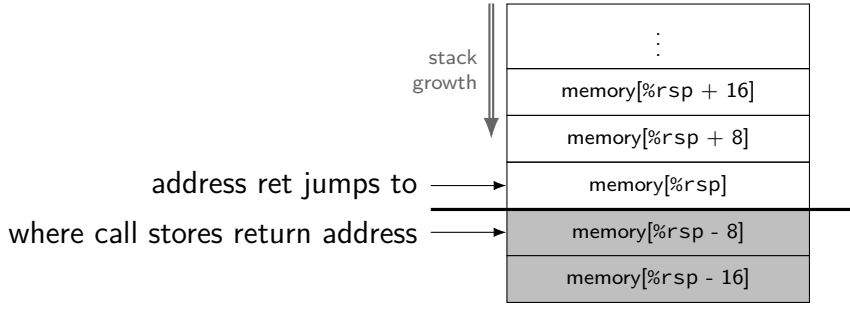
call/ret

`call` LABEL

push PC (next instruction address) on stack
jmp to LABEL address

`ret`

pop address from stack
jmp to that address



Y86-64 state

%r*XX* — 15 registers

- ~~%r15~~ missing
- smaller parts of registers missing

ZF (zero), SF (sign), ~~OF (overflow)~~

- book has OF, we'll not use it
- ~~CF~~ (carry) missing

Stat — processor status — halted?

PC — program counter (AKA instruction pointer)

main memory

typical RISC ISA properties

fewer, simpler instructions

seperate instructions to access memory

~~fixed-length instructions~~

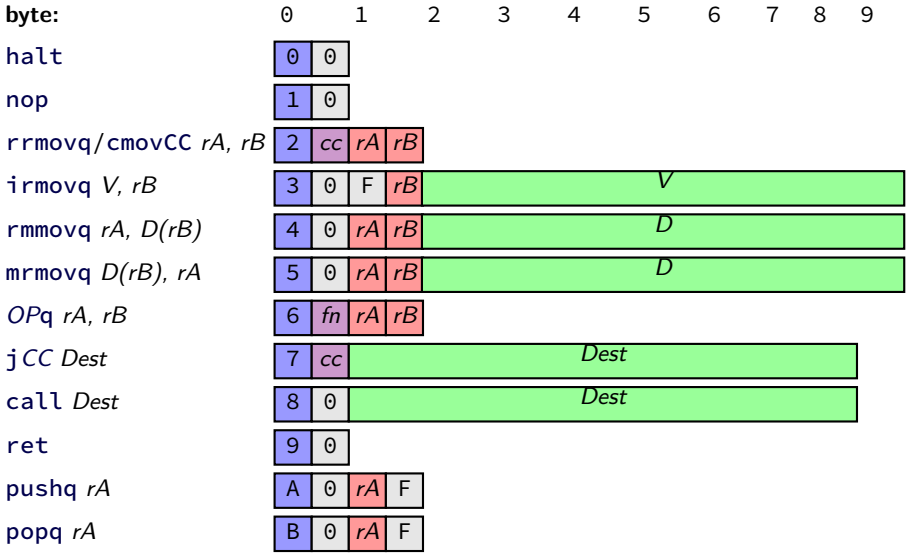
more registers

no “loops” within single instructions

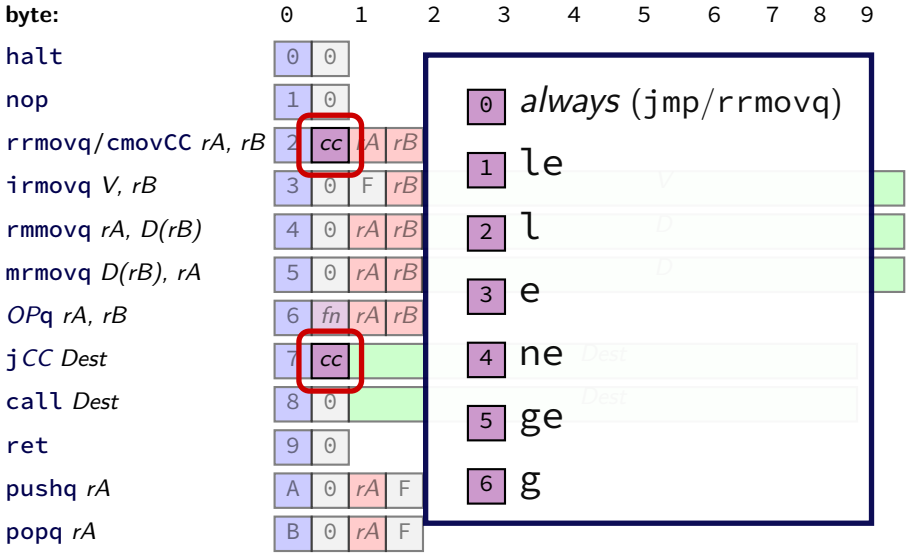
no instructions with two memory operands

few addressing modes

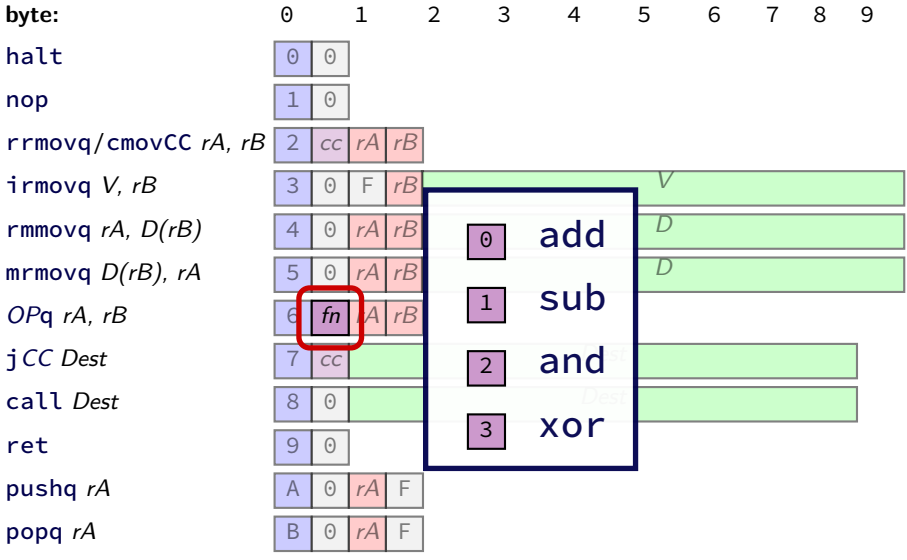
Y86-64 instruction formats



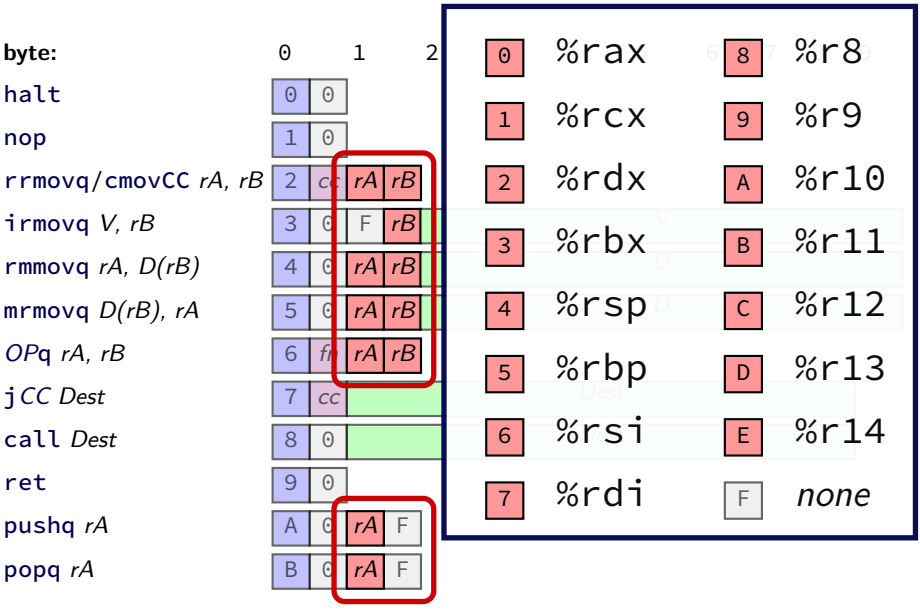
Secondary opcodes: `cmov`*cc*`/j`*cc*



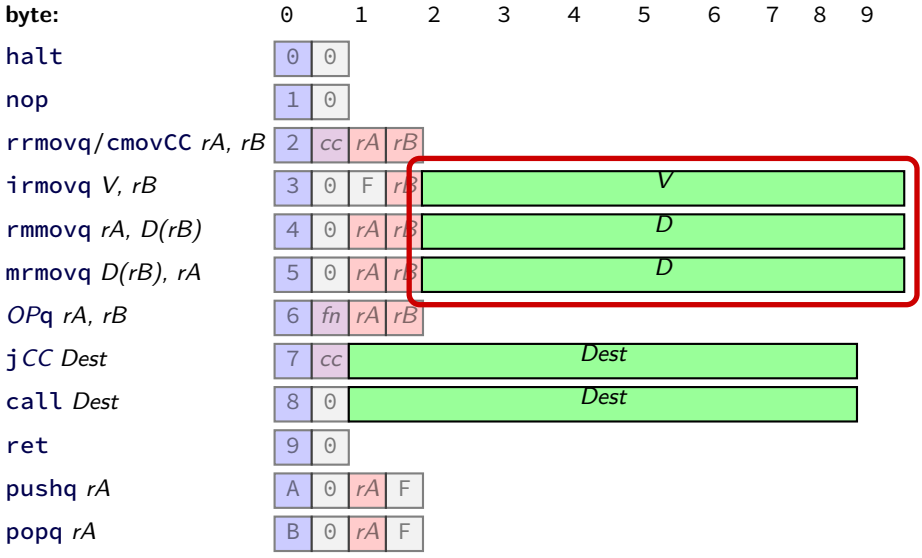
Secondary opcodes: *OP*q



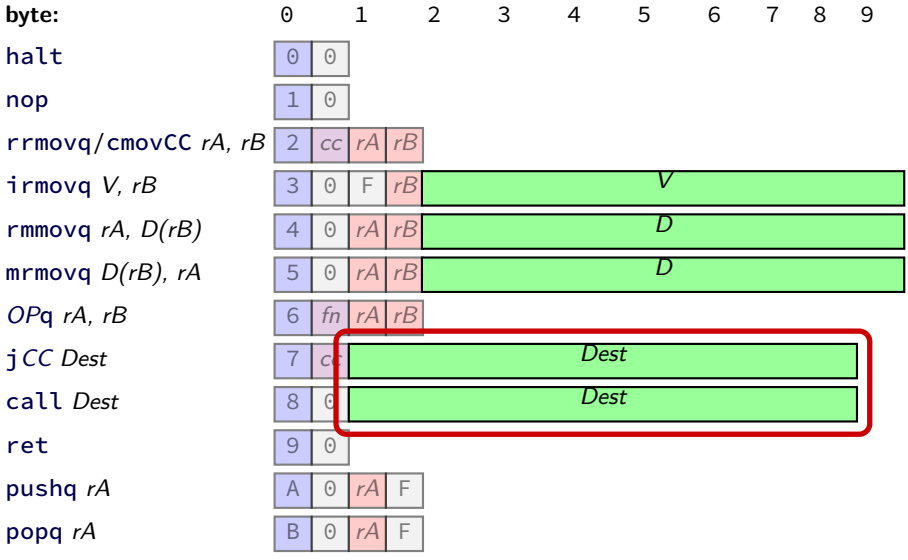
Registers: *rA, rB*



Immediates: *V, D, Dest*



Immediates: *V, D, Dest*



Y86-64 encoding (1)

```
long addOne(long x) {
    return x + 1;
}
```

x86-64:

```
    movq %rdi, %rax
    addq $1, %rax
    ret
```

Y86-64:

Y86-64 encoding (1)

```
long addOne(long x) {
    return x + 1;
}
```

x86-64:

```
    movq %rdi, %rax
    addq $1, %rax
    ret
```

Y86-64:

```
    irmovq  $1,   %rax
    addq    %rdi, %rax
    ret
```

Y86-64 encoding (2)

```
addOne:
  irmovq   $1,    %rax
  addq     %rdi, %rax
  ret
```

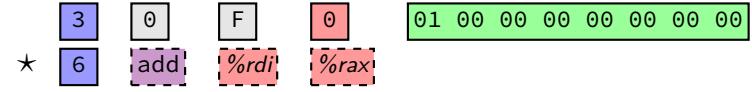
Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```

★ 3 | 0 | F | 0 | 01 00 00 00 00 00 00 00

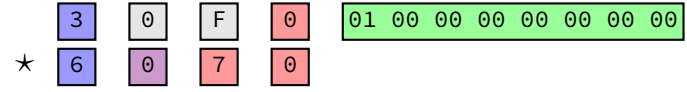
Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```



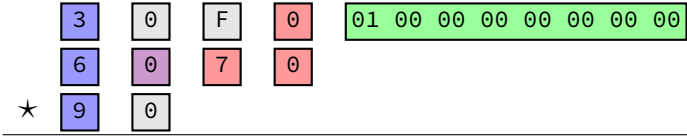
Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```



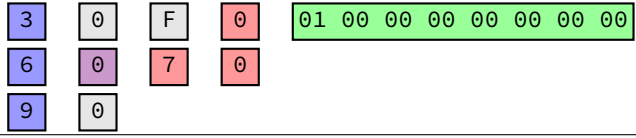
Y86-64 encoding (2)

```
addOne:
   irmovq   $1,    %rax
   addq     %rdi, %rax
   ret
```



Y86-64 encoding (2)

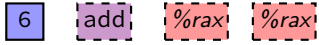
```
addOne:
   irmovq   $1,    %rax
   addq     %rdi, %rax
   ret
```




```
30 F0 01 00 00 00 00 00 00 00 60 70 90
```

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



Y86-64 encoding (3)

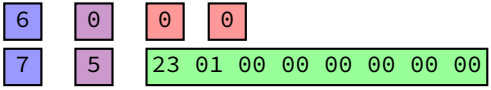
```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

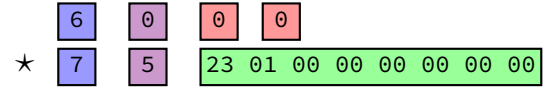
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



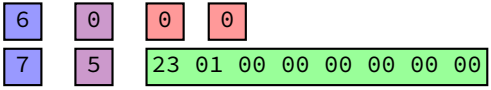
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



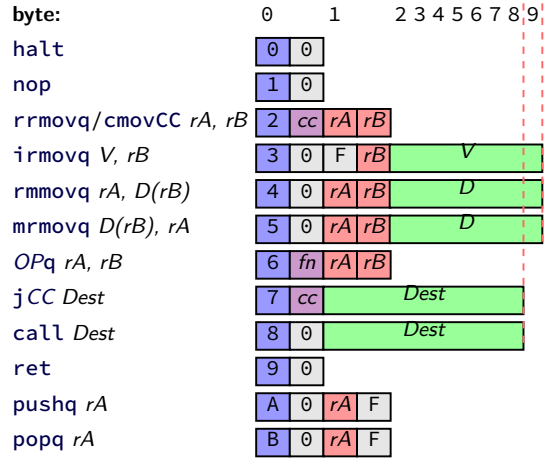
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



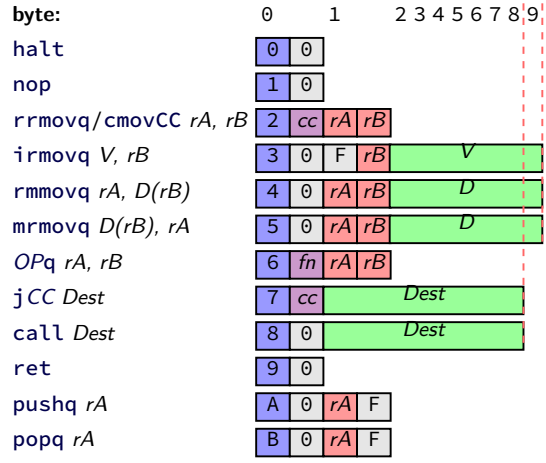
Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```



Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

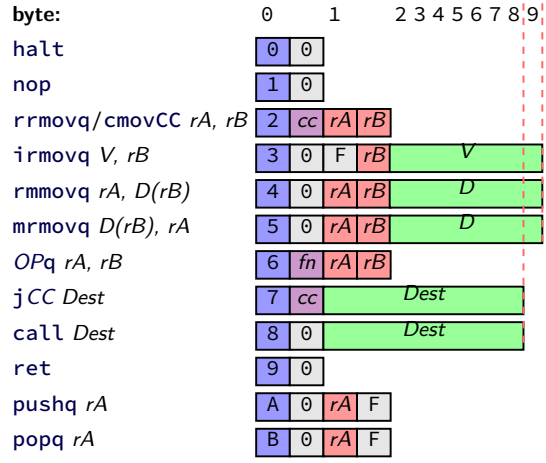


Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

`rrmovq %rcx, %rax`

- 0 as cc: always
- 1 as reg: `%rcx`
- 0 as reg: `%rax`



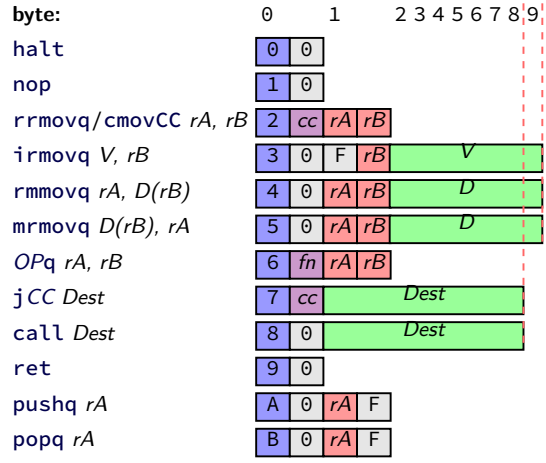
Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

```
rrmovq %rcx, %rax

addq   %rdx, %rax
subq   %rbx, %rdi
```

- 0 as fn: add
- 1 as fn: sub



Y86-64 decoding

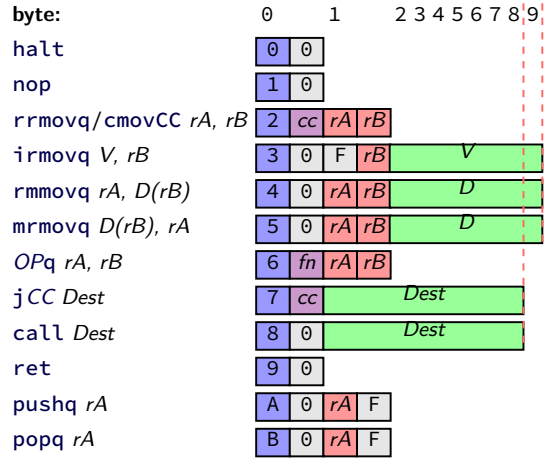
```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

```
rrmovq %rcx, %rax

addq   %rdx, %rax
subq   %rbx, %rdi

jl     0x84
```

- 2 as cc: `l` (less than)
- hex 84 00… as little endian *Dest: 0x84*



Y86-64 decoding

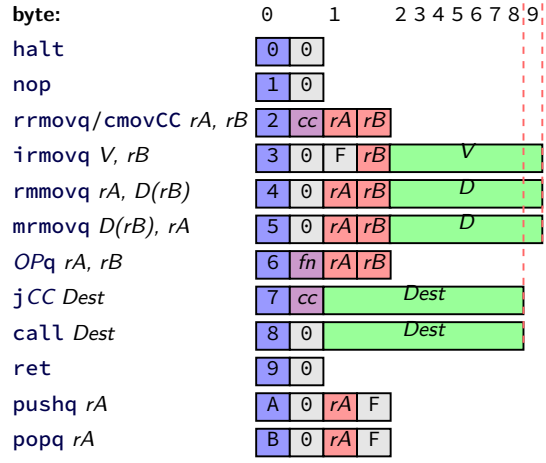
```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

```
rrmovq %rcx, %rax

addq   %rdx, %rax
subq   %rbx, %rdi

jl     0x84

rrmovq %rcx, %rdx
rrmovq %rax, %rcx
jmp    0x68
```



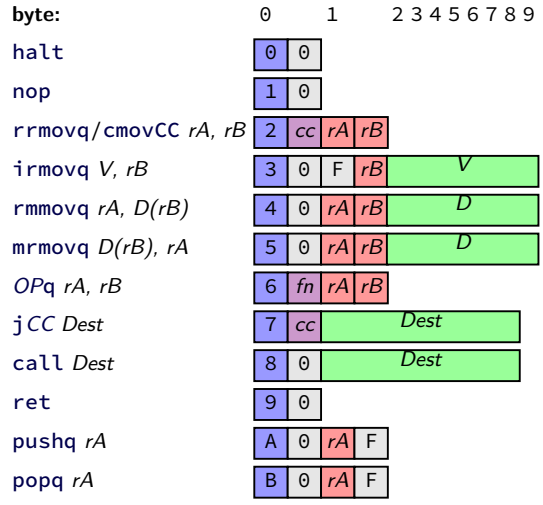
Y86-64: convenience for hardware

4 bits to decode instruction size/layout

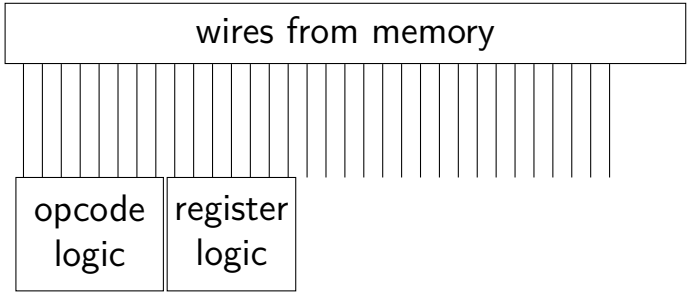
(mostly) uniform placement of operands

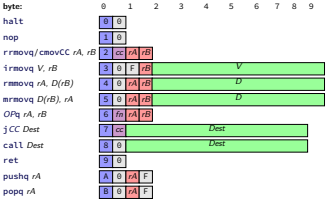
jumping to zeroes (uninitialized?) by accident halts

no attempt to fit (parts of) multiple instructions in a byte



on uniform placement



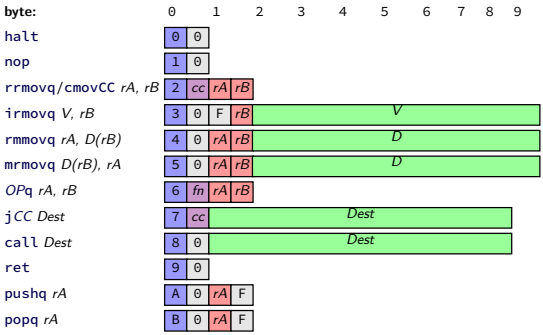


simpler hardware: directly wire memory bits to where needed

disadvantage: "wasted" space in instructions

- e.g. 0s and Fs in Y86
- and more space with fixed-length instruction sets

on variable size



future: want to run instructions in parallel

problem with Y86: need to read one instruction to even find next

but at least only need to process 4 bits

Y86-64

Y86-64: simplified, more RISC-y version of X86-64

minimal set of arithmetic

only movs touch memory

only jumps, calls, and movs take immediates

simple variable-length encoding

next time: implementing with circuits

backup slides

miscellaneous bit manipulation

common bit manipulation instructions are not in C:

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

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

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

byte swap: (x86: `bswap`)