Changelog

Changes made in this version not seen in first lecture:
- 12 September 2017: slide 28, 33: quote solution that uses z correctly

last time

Y86 — choices, encoding and decoding

shift operators
- `shr` assembly, `>>` in C
- right shift = towards least significant bit
- right shift = dividing by power of two

on the quizzes in general

yes, I know quizzes are hard

intention: quiz questions from slides/etc. + some serious thought

(and sometimes I miss the mark)

main purpose: review material other than before exams, warning sign for me
- why graded? because otherwise...

on the quiz (1)

RISC versus CISC: about simplifying hardware

variable-length encoding is less simple for HW

instructions chosen more based on what's simple for HW
    (e.g. push/pop not simple for HW)

more registers — simpler than adding more instructions
    compensates for seperate memory instructions

---

on the quiz (2)

instruction set — what the instructions do

size of memory address — operands to rmmovq, etc. mean what?

floating point support — do such instructions exist?

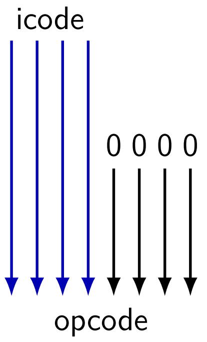
---

constructing instructions

```
typedef unsigned char byte;
byte make_simple_opcode(byte icode) {
    // function code is fixed as 0 for now
    return opcode * 16;
    // 16 = 1 0000 in binary
}
```

---

constructing instructions in hardware



reversed shift right?

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

~~`opcode >> (-4)`~~

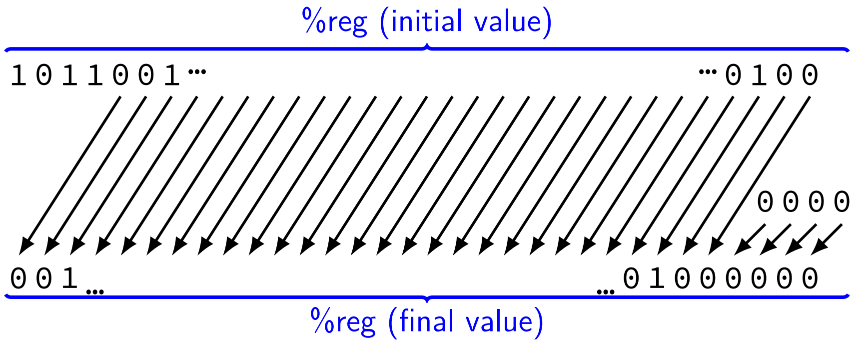
---

shift left

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

C: `value << amount`

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



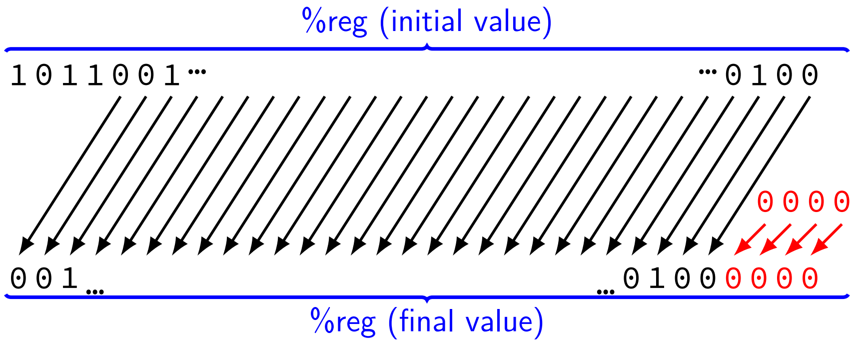
---

shift left

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

C: `value << amount`

`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

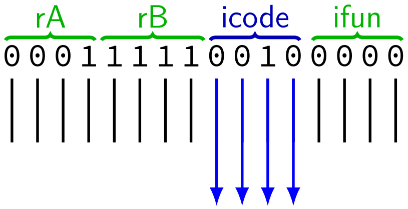
```
1 << 0 == 1             0000 0001
1 << 1 == 2             0000 0010
1 << 2 == 4             0000 0100

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

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

---

extracting icode from more

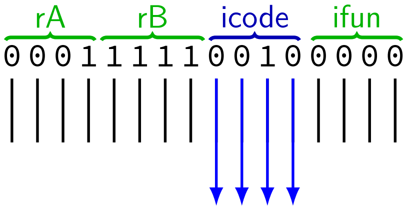


rA rB icode ifun

0 0 0 1 1 1 1 1 0 0 1 0 0 0 0 0

---

extracting icode from more



rA rB icode ifun

0 0 0 1 1 1 1 1 0 0 1 0 0 0 0 0

```
// % -- remainder
unsigned extract_opcode1(unsigned value) {
    return (value / 16) % 16;
}

unsigned extract_opcode2(unsigned value) {
    return (value % 256) / 16;
}
```

---

manipulating bits?

easy to manipulate individual bits in HW

how do we expose that to software?

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
```

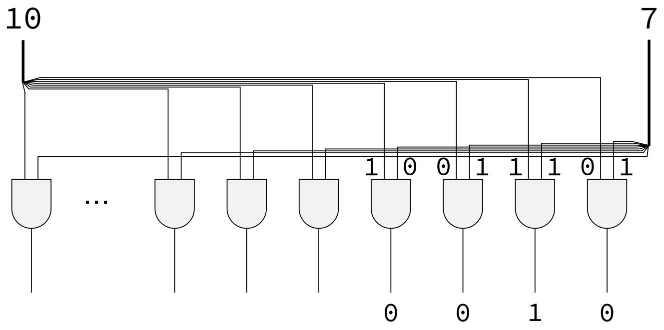
---

bitwise AND — C/assembly

x86: **and** `%reg, %reg`

C: `foo & bar`

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



---

extract opcode from larger

```
unsigned extract_opcode1_bitwise(unsigned value) {
    return (value >> 4) & 0xF; // 0xF: 00001111
    // like (value / 16) % 16
}

unsigned extract_opcode2_bitwise(unsigned value) {
    return (value & 0xF0) >> 4; // 0xF0: 11110000
    // like (value % 256) / 16;
}
```

---

extract opcode from larger

```
extract_opcode1_bitwise:
    movl %edi, %eax
    shrl $4, %eax
    andl $0xF, %eax
    ret

extract_opcode2_bitwise:
    movl %edi, %eax
    andl $0xF0, %eax
    shrl $4, %eax
    ret
```

---

more truth tables

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

&

conditionally clear bit
conditionally keep bit

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

|

conditionally set bit

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

^

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)

$$\begin{array}{r} \sim \overbrace{0\ 0\ \dots\ 0\ 0\ 0\ 0}^{\text{32 bits}} \\ \hline 1\ 1\ \dots\ 1\ 1\ 1\ 1 \end{array}$$

---

negation / not — ~

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

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF
```
(aka −1)

```
~2 == (int) 0xFFFFFFFD
```
(aka −3)

$$\begin{array}{r} \sim \overbrace{0\ 0\ \dots\ 0\ 0\ 0\ 0}^{\text{32 bits}} \\ \hline 1\ 1\ \dots\ 1\ 1\ 1\ 1 \end{array}$$

negation / not — ~

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

```
!0 == 1
!notZero == 0
~0 == (int) 0xFFFFFFFF (aka −1)
~2 == (int) 0xFFFFFFFD (aka −3)
```

$$\begin{array}{r} \sim\ \overbrace{0\ 0\ \ldots\ 0\ 0\ 0\ 0}^{\text{32 bits}} \\ \hline 1\ 1\ \ldots\ 1\ 1\ 1\ 1 \end{array}$$

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

---

note: ternary operator

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

---

one-bit ternary

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

constraint: everything is 0 or 1

now: reimplement in C without if/else/||/etc.
(assembly: no jumps probably)

---

one-bit ternary

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

constraint: everything is 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: everything is 0 or 1

`(x ? y : 0)`

---

one-bit ternary parts (1)

constraint: everything is 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: everything is 0 or 1 — but y, z is any integer

`(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) & 1)` only gets least sig. bit

`(x ? y : z)`

---

multibit ternary

constraint: x is 0 or 1

old solution `((x & y) | (~x) & 1)` 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`)

one idea: `x | (x << 1) | (x << 2) | ...`

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`)

one idea: `x | (x << 1) | (x << 2) | ...`

a trick: −`x`

---

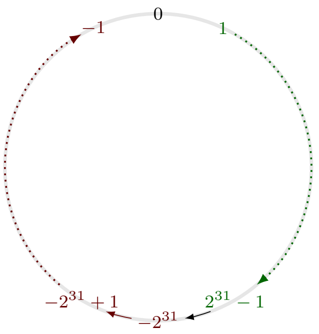
two’s complement refresher

$$\begin{array}{rccccccc} & -2^{31} & +2^{30} & +2^{29} & & +2^{2} & +2^{1} & +2^{0} \\ -1 = & 1 & 1 & 1 & \ldots & 1 & 1 & 1 \end{array}$$

---

two’s complement refresher

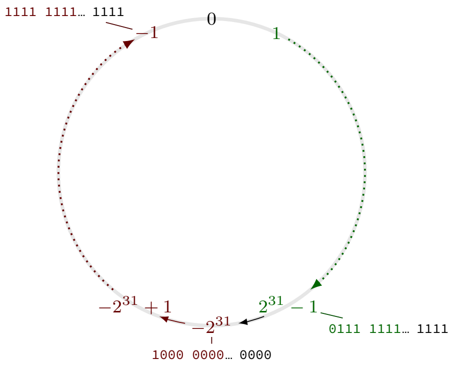
$$\begin{array}{rccccccc} & -2^{31} & +2^{30} & +2^{29} & & +2^{2} & +2^{1} & +2^{0} \\ -1 = & 1 & 1 & 1 & \ldots & 1 & 1 & 1 \end{array}$$



---

two’s complement refresher

$$\begin{array}{rccccccc} & -2^{31} & +2^{30} & +2^{29} & & +2^{2} & +2^{1} & +2^{0} \\ -1 = & 1 & 1 & 1 & \ldots & 1 & 1 & 1 \end{array}$$



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`)

one idea: `x | (x << 1) | (x << 2) | ...`

a trick: `−x`

`((-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) & 1)` 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 — ∼ 1 cycle
- integer multiply — ∼ 1-3 cycles
- integer divide — ∼ 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 — ~ 1 cycle
- integer multiply — ~ 1-3 cycles
- integer divide — ~ 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`

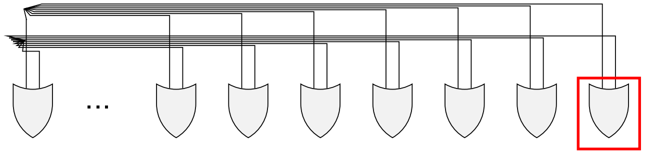
---

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



---

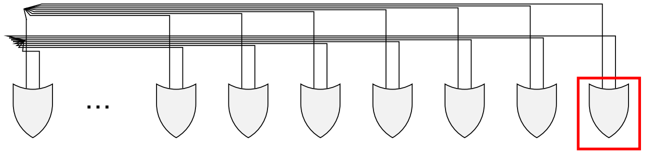
wasted work (2)

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

performing 3 bitwise ors

…each bitwise or does 4 OR operations

2/3 of bitwise ORs useless — don't use upper bits



any-bit: divide and conquer

four-bit input $x = x_1x_2x_3x_4$

`(x >> 1) | x` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1y_2y_3y_4$

---

any-bit: divide and conquer

four-bit input $x = x_1x_2x_3x_4$

`(x >> 1) | x` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1y_2y_3y_4$

$y_2 =$ any-of$(x_1x_2) = x_1|x_2$, $y_4 =$ any-of$(x_3x_4) = x_3|x_4$

---

any-bit: divide and conquer

four-bit input $x = x_1x_2x_3x_4$

`(x >> 1) | x` $= (x_1|0)(x_2|x_1)(x_3|x_2)(x_4|x_3) = y_1y_2y_3y_4$

$y_2 =$ any-of$(x_1x_2) = x_1|x_2$, $y_4 =$ any-of$(x_3x_4) = x_3|x_4$

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

---

parallelism

bitwise operations — each bit is seperate

parallelism

bitwise operations — each bit is seperate

same idea can apply to more interesting operations

$010 + 011 = 101;\ 001 + 010 = 011 \rightarrow$
$01000001 + 01100010 = 10100011$

---

parallelism

bitwise operations — each bit is seperate

same idea can apply to more interesting operations

$010 + 011 = 101;\ 001 + 010 = 011 \rightarrow$
$01000001 + 01100010 = 10100011$

sometimes specific HW support

- e.g. x86-64 has a “multiply four pairs of floats” instruction

---

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? ($abcdef$ becomes $abcdfe$)

```
/* version A */
    return ((x >> 1) & 1) | (x & (~1));

/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));

/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);

/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
```

version A

```
/* version A */
    return ((x >> 1) & 1) | (x & (~1));
    //     ^^^^^^^^^^^^^^
    //      abcdef --> 0abcde -> 00000e

    //                      ^^^^^^^^^^
    //      abcdef --> abcde0

    //     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //      00000e | abcde0 = abcdee
```

version B

```
/* version B */
    return ((x >> 1) & 1) | ((x << 1) & (~2)) | (x & (~3));
    //     ^^^^^^^^^^^^^^
    //      abcdef --> 0abcde --> 00000e

    //                       ^^^^^^^^^^^^^^^
    //      abcdef --> bcdef0 --> bcde00

    //                                          ^^^^^^^^^
    //      abcdef -->            abcd00
```

version C

```
/* version C */
    return (x & (~3)) | ((x & 1) << 1) | ((x >> 1) & 1);
    //     ^^^^^^^^^^
    //      abcdef -->            bcde00

    //                  ^^^^^^^^^^^^^^
    //      abcdef --> 00000f --> 0000f0

    //                                   ^^^^^^^^^^^^^
    //      abcdef --> 0abcde --> 00000e
```

version D

```
/* version D */
    return (((x & 1) << 1) | ((x & 3) >> 1)) ^ x;
    //     ^^^^^^^^^^^^^^^
    //      abcdef --> 00000f --> 0000f0

    //                            ^^^^^^^^^^^^^^
    //      abcdef --> 0000ef --> 00000e

    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //      0000fe ^ abcdef --> abcd(f XOR e)(e XOR f)
```

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

---

backup slides

right shift in math

```
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
```

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

---

non-power of two arithmetic

```
unsigned times130(unsigned x) {
    return x * 130;
}
```

---

non-power of two arithmetic

```
unsigned times130(unsigned x) {
    return x * 130;
}
unsigned times130(unsigned x) {
    return (x << 7) + (x << 1); // x * 128 + x * 2
}
```

---

non-power of two arithmetic

```
unsigned times130(unsigned x) {
    return x * 130;
}
unsigned times130(unsigned x) {
    return (x << 7) + (x << 1); // x * 128 + x * 2
}
```

```
times130:
    movl %edi, %eax
    shll $7, %eax
    leal (%rax, %rdi, 2), %eax
    ret
```