cs107e

§1 Baremetal systems programming

§2 Building a personal computer

- Graphics

- Keyboard

§3 Topics in systems programming

Systems Programming

Computer arithmetic (today)

Caches; debugging discussion

New systems prog lang: Rust and Go
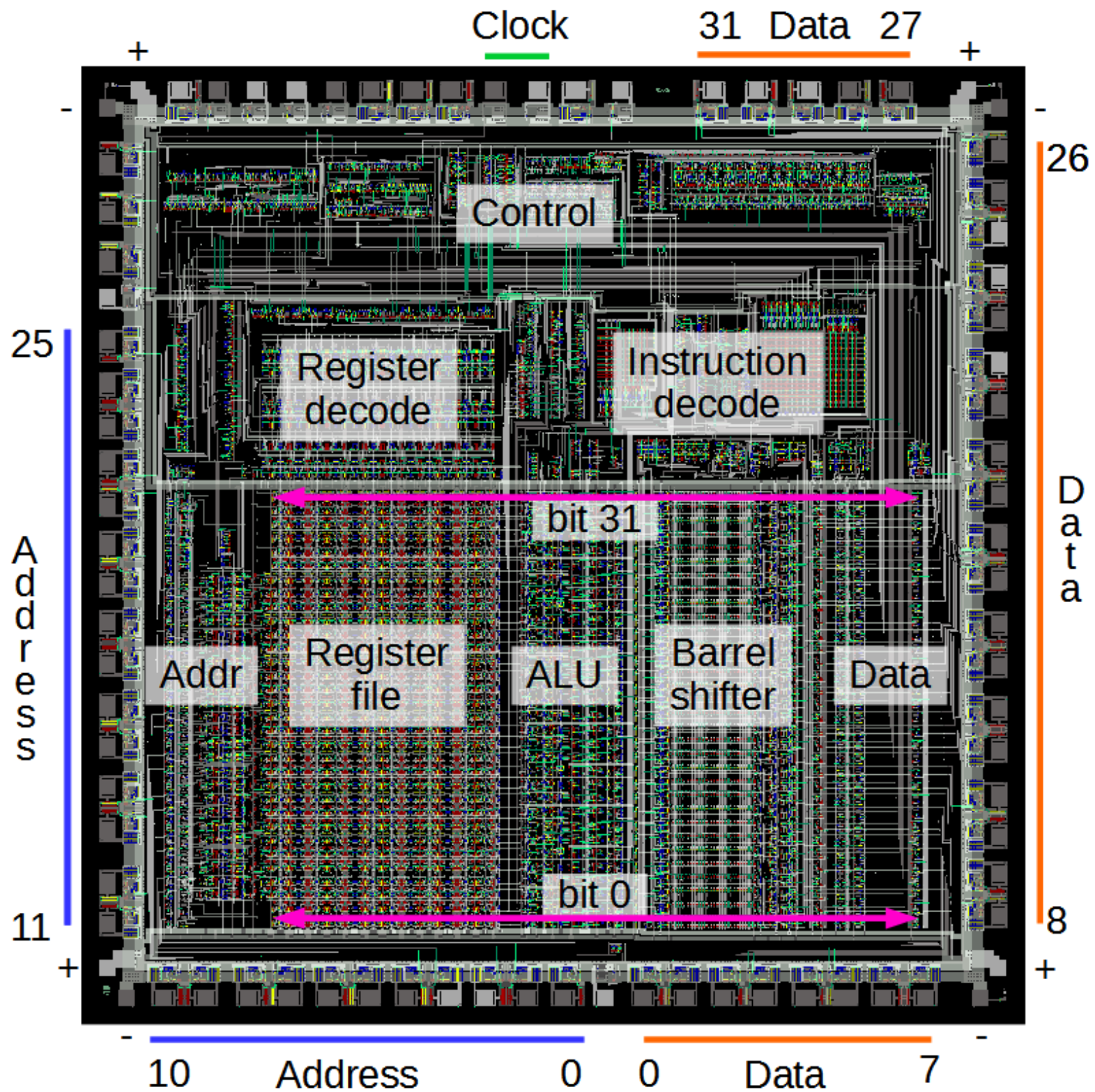
Veronica, Quinn Dunki

To Linux and beyond

Final project demonstrations

Debugging discussion (due Mon)

Send me email with the answers to the following questions. Think of a memorable bug you encountered this quarter.

1. What was the bug?
2. What were the symptoms?
3. How did you figure out what the problem was?
4. How did you fix it?
5. What lessons did you learn from this experience?

Send the answers to hanrahan@cs.stanford.edu by 12 noon on Mon.

Clock   31   Data   27

26

Control

25

Register decode   Instruction decode

bit 31

Address   Data

Addr   Register file   ALU   Barrel shifter   Data

bit 0

11   8

10   Address   0   0   Data   7

Learning Goals

Processor flags: Z, N, C, V

Addition and carry, subtraction and borrow

2's complement representation of negative numbers

Signed and unsigned conversions (C craziness)

Arithmetic-logic units (ALUs)

Addition

High School Addition

```
 247
   9
 ---
```

High School Addition

```
   1
 247
   9
 ---
   6
```

High School Addition

```
    1
  247
    9
  ---
   56
```

High School Addition

```
   1
 247
  09
 ---
 256
```

High School Addition (Hexadecimal)

```
 F7
 09
---
```

High School Addition (Hexadecimal)

```
   1
  F7
  09
 ---
   0
```

High School Addition (Hexadecimal)

```
 11
  F7
  09
 ---
  00
```

High School Addition (Hexadecimal)

```
 11
  F7
  09
 ---
 100
```

High School Addition (Binary)

```
 11110111
 00001001
----------
```

High School Addition (Binary)

```
         1
  11110111
  00001001
 ----------
         0
```

High School Addition (Binary)

```
        11
  11110111
  00001001
----------
        00
```

High School Addition (Binary)

```
      111
 11110111
 00001001
---------
      000
```

High School Addition (Binary)

```
     1111
  11110111
  00001001
----------
      0000
```

High School Addition (Binary)

```
    11111
 11110111
 00001001
----------
    00000
```

High School Addition (Binary)

```
   111111
  11110111
  00001001
----------
   000000
```

High School Addition (Binary)

```
 1111111
 11110111
 00001001
----------
 0000000
```

High School Addition (Binary)

```
 11111111
  11110111
  00001001
 ----------

  00000000
```

High School Addition (Binary)

```
 11111111
  11110111
  00001001
 ----------
  00000000
```

Result: 00000000 (only room for 8-bits)
Carry (C): 1 (extra bit)

To hold the result of adding two n-bit numbers requires n+1 bits

CPSR
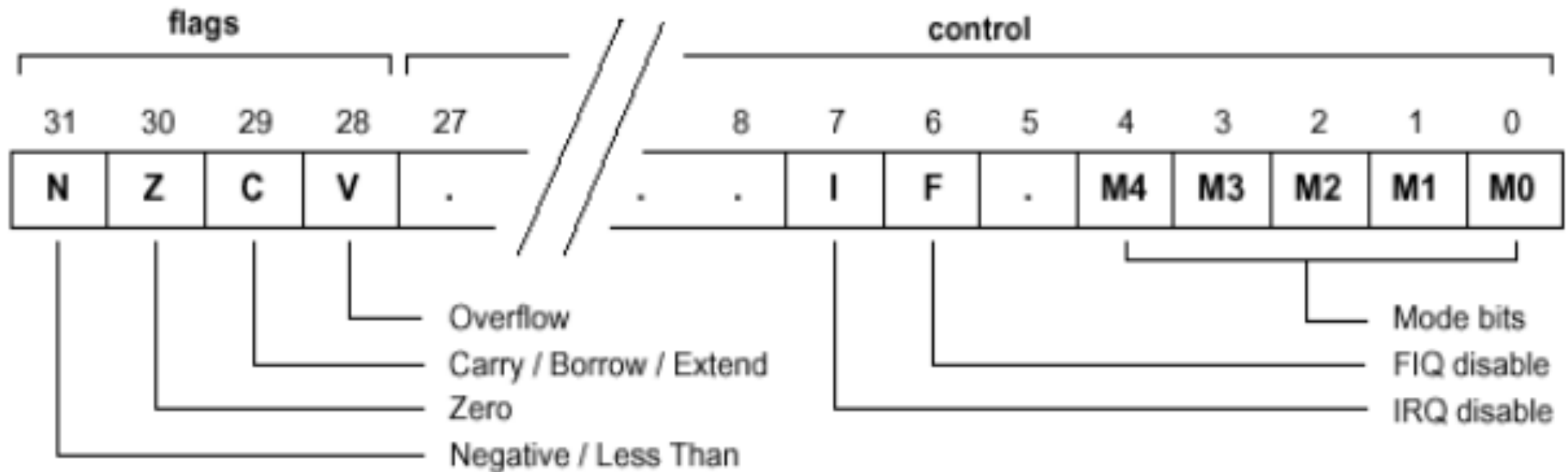
Arithmetic instructions set Z, N, C, V
Logic instructions set Z, N

We will cover V soon

cpsr.c : uadd32

```
// Multiple precision addition
// https://gcc.godbolt.org/

int64_t add64(int64_t a, int64_t b)
{
    return a + b;
}


add64(int64_t, int64_t):
        adds    r2, r2, r0 // adds, not add
        adc     r3, r3, r1 // add w/ carry
        mov     r0, r2
        mov     r1, r3
        bx      lr
```

Negative Numbers

Up to now, all binary numbers have been positive (unsigned)

How would you define negative numbers?

A clever way of defining -1 is to say that it is the number that when added to 1, results in 0 mod 256

0xFF + 0x01 = 0x100 = 0x00

0xFF can be *interpreted* as -1

In this system, adding unsigned numbers is exactly the same as adding signed numbers. Cool!

How do we negate a number?

Subtract the number from 0!

```
  11111111
   00000000
   00000001
  ----------
   11111111
```

11111111 (0xFF) is -1

Note the *borrows*.

```
0x00 =   0
0xFF = -1
0xFE = -2
…
0x80 = -128 (could interpret as 128)
0x7F =   127
…
0x01 =   1
```

if we choose to *interpret* 0x80=-128,
then the most-significant bit of the number
indicates that it is negative (N) - sign bit

8-bit signed number range from -128 to 127
32-bit numbers from -2147483648 to 2147483647

But, 100000000 is more than 8-bits!

Rewrite as 100000000 = 11111111 + 1
-x = (11111111-x)+1

```
  11111111
 -00000001
 ----------
  11111110 = ~00000001


  11111110
 +00000001
 ----------
  11111111
```

Negating

-x = ~x + 1

Some nomenclature:

~x : One's complement (~0-x)

~x + 1: Two's complement (2^{32}-x)

Subtraction is negation and addition

$a - b = a + (-b) = a + \sim b + 1$

```
0x01 - 0x00 = 0x01+0xFF+0x01 = 0x01 + C
0x01 - 0x01 = 0x01+0xFE+0x01 = 0x00 + C
0x01 - 0x02 = 0x01+0xFD+0x01 = 0xFF
```

Note that carry is set if a >= b

borrow = !carry

cpsr.c : usub32

```
// Multiple precision subtraction
// https://gcc.godbolt.org/

int64_t sub64(int64_t a, int64_t b)
{
    return a - b;
}


sub64(int64_t, int64_t):
        subs      r2, r2, r0 // subs, not sub
        sbc       r3, r3, r1 // sub w/ carry
        mov       r0, r2
        mov       r1, r3
        bx        lr
```

Signed subtraction - overflow (sub32)

```
80000000+00000001=80000001 : z=0,n=1,c=0,uge=0
80000000-00000001=7fffffff : z=0,n=0,c=1,uge=1
7fffffff+00000001=80000000 : z=0,n=1,c=0,uge=0
7fffffff-00000001=7ffffffe : z=0,n=0,c=1,uge=1
-2147483648+1=-2147483647 : z=0,n=1,c=0,v=0,ge=0
-2147483648-1= 2147483647 : z=0,n=0,c=1,v=1,ge=0
 2147483647+1=-2147483648 : z=0,n=1,c=0,v=1,ge=1
 2147483647-1= 2147483646 : z=0,n=0,c=1,v=0,ge=1
```

"Overflow" happens when adding two numbers of
the same sign, and getting a result
with a different sign.

V = (n1 == n2) & (n != n1)

Comparison

```
unsigned comparison (usub32)

00000000-00000000=00000000 : z=1,n=0,c=1,uge=1
00000000-00000001=ffffffff : z=0,n=1,c=0,uge=0
00000001-00000000=00000001 : z=0,n=0,c=1,uge=1
00000001-00000001=00000000 : z=1,n=0,c=1,uge=1
00000001-ffffffff=00000002 : z=0,n=0,c=0,uge=0
ffffffff-00000001=fffffffe : z=0,n=1,c=1,uge=1
ffffffff-ffffffff=00000000 : z=1,n=0,c=1,uge=1

bcs: branch carry set (greater than or equal)
bcc: branch carry clear (less than)
```

Signed comparison (sub32)

```
00000000-00000000=00000000 : z=1,n=0,c=1,uge=1
00000000-00000001=ffffffff : z=0,n=1,c=0,uge=0
00000001-00000000=00000001 : z=0,n=0,c=1,uge=1
00000001-00000001=00000000 : z=1,n=0,c=1,uge=1
00000001-ffffffff=00000002 : z=0,n=0,c=0,uge=0
ffffffff-00000001=fffffffe : z=0,n=1,c=1,uge=1
ffffffff-ffffffff=00000000 : z=1,n=0,c=1,uge=1
```

bge: signed greater than or equal (n == v)
blt: signed less than (n != v)

int ge() { return !v ? n : !n }

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Signed and Unsigned Types in C

Ninja Job Interview Questions

Type Hierarchy

uint32   int32

uint16   int16

uint8   int8
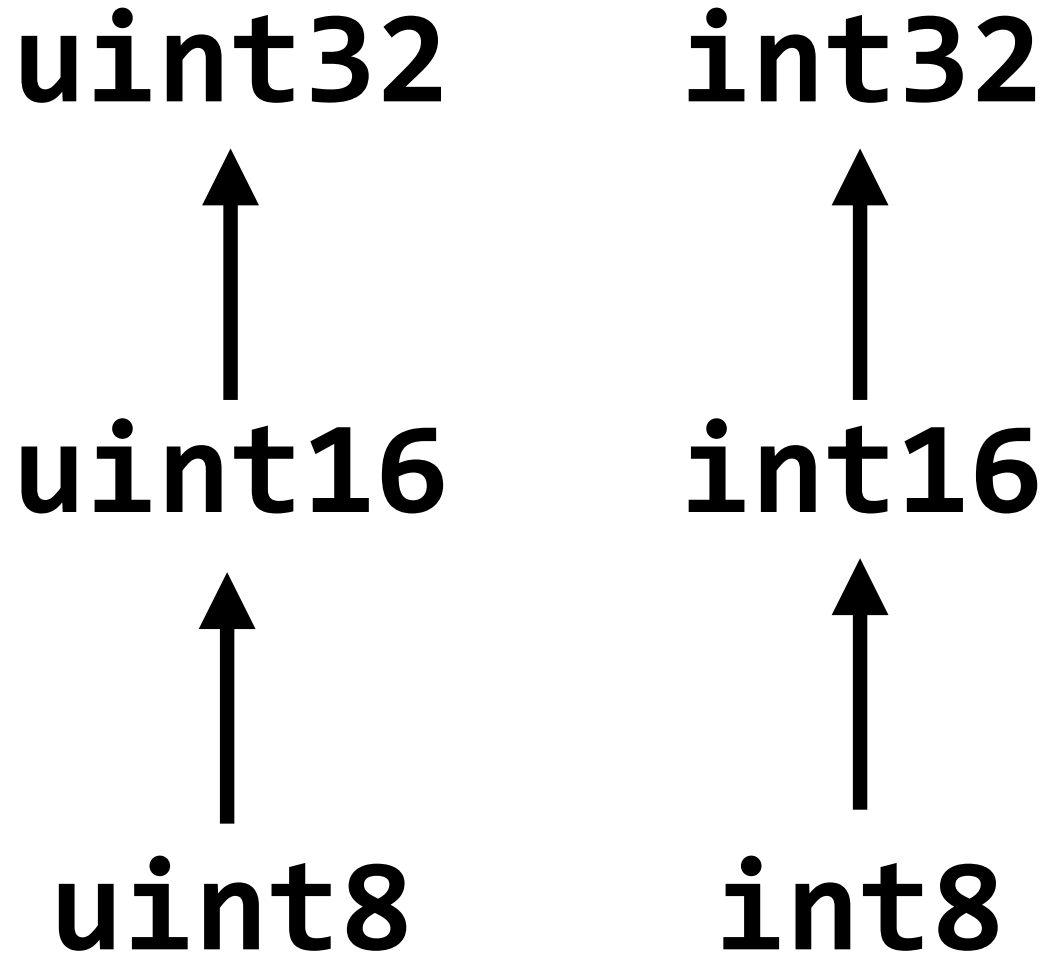
Types are sets of allowed values
Arrow indicate subsets: uint8 ⊂ uint16

```
// Sign extension:extend signrightwards
// code/sign.c

int8 0xFE -> int32 0xFFFFFFFE
int8 0x7E -> int32 0x0000007E


// Assembly language
LSL r0,r0,#24
0xFE -> 0xFE000000
ASR r0,r0,#24
0xFE000000 -> 0xFFFFFFFE
```

uint32   int32

↓   ↓

uint16   int16

↓   ↓

uint8   int8

Truncate (throw away the msbs)

uint32 ← int32
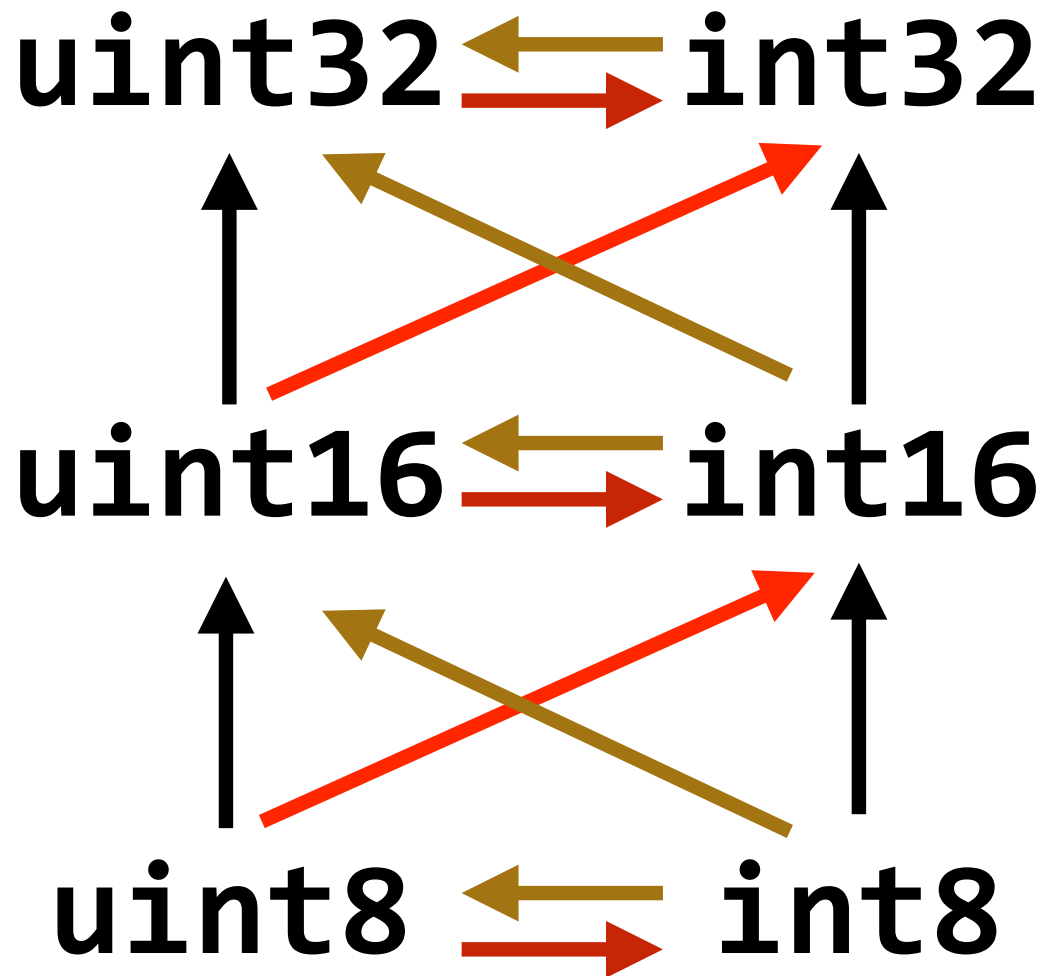
uint16 ← int16

uint8 ← int8

Ok?

uint32 ⟶ int32

uint16 ⟶ int16

uint8 ⟶ int8

Ok?

uint32　　int32

uint16　　int16

uint8　　int8

Ok?

uint32   int32

$\nwarrow$

uint16   int16

$\nwarrow$

uint8   int8

Ok?
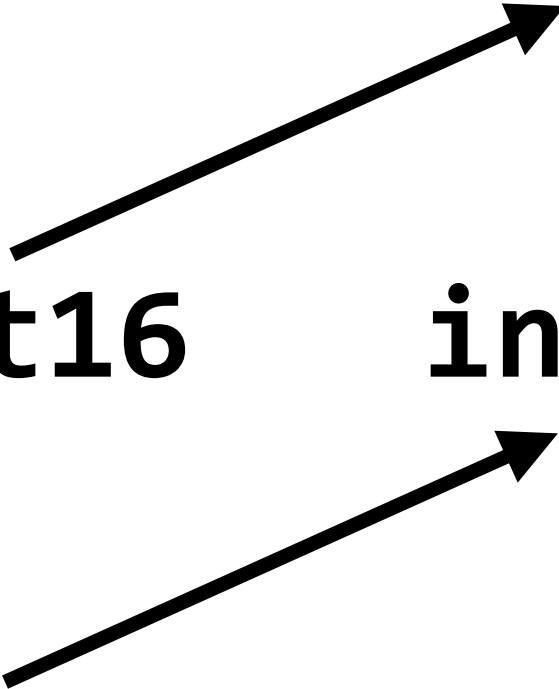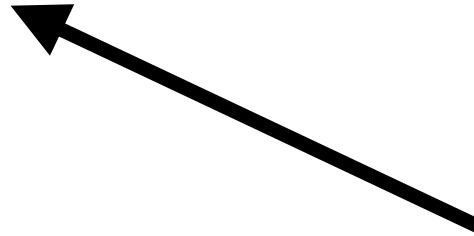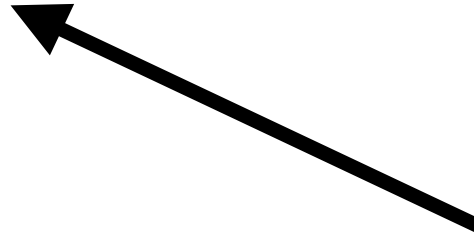
6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than _Bool, if the value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

3 Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Operations involving signed and unsigned

int32 + uint32, int32 < uint32

What is the type of the result?

Why is this the right thing to do?

Operations involving signed and unsigned

int32 + uint32?

What is the type of the result?
  Signed integers are changed to
  unsigned!

Why is this the right thing to do?
  Unsigned conversion is well-defined

This leads to unexpected behavior!

```c
#include <stdio.h>

int main(void)
{
    unsigned int a = 6;
    int b = -20;
    (b>a) ?
        puts("-20 > 6") :
        puts("-20 <= 6") ;
}
```

6.3.1.8 Usual arithmetic conversions

1 If both operands have the same type, then no further conversion is needed.

2 Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

3 Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

4 Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

5 Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.
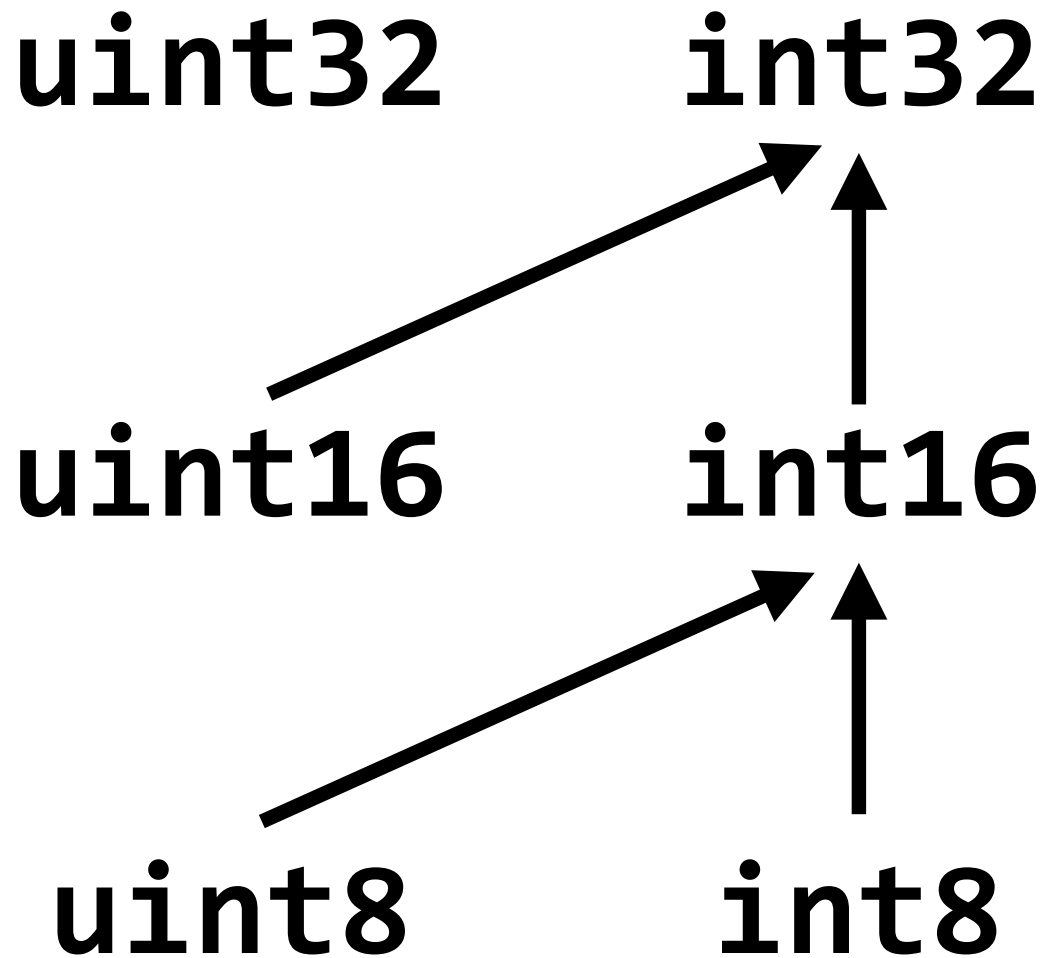
Alternative: Promote to greatest common type

"Whenever you mix signed and
unsigned numbers you get in trouble"

Bjarne Stroustrup
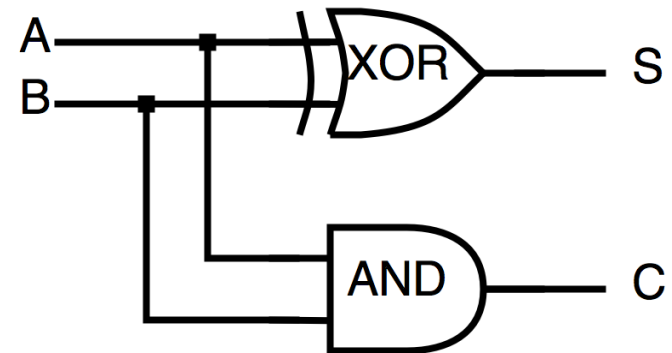
1-Bit Adder

```
Add 2 1-bit numbers

a b sum
0 0  00
0 1  01
1 0  01
1 1  10
```

Add 2 1-bit numbers (Half Adder)

```
a b sum
0 0  00
0 1  01
1 0  01
1 1  10
```

bit 0 of sum: S = a^b

bit 1 of sum: C = a&b

Have reduced addition to bitwise,
logical operations!

Add 3 1-bit numbers
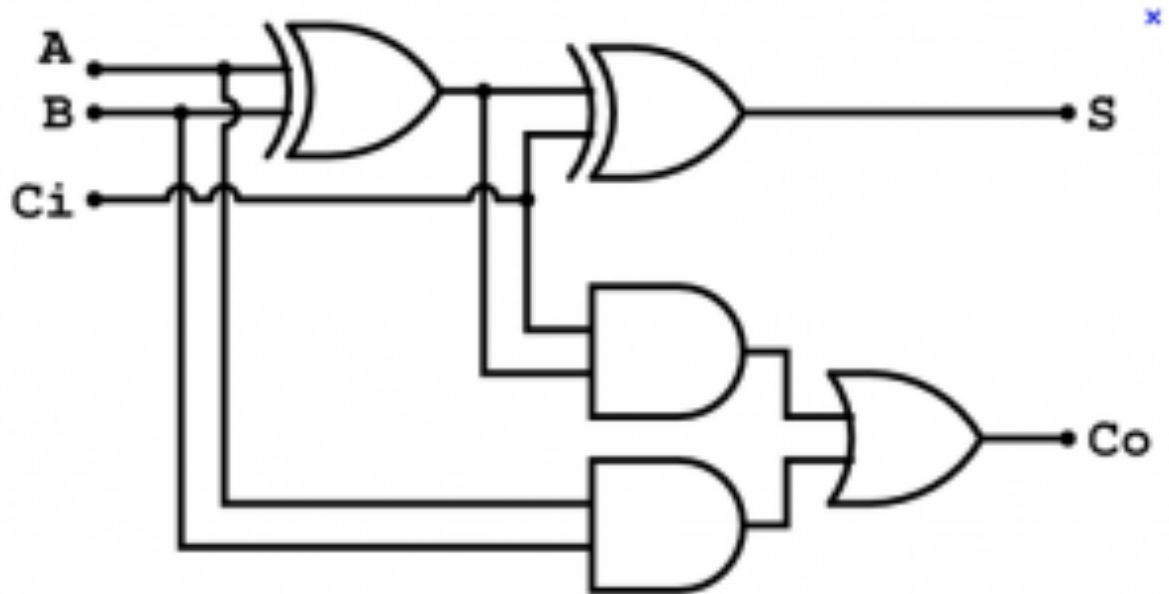
```
a b c = s c
0 0 0   0 0
0 1 0   0 1
1 0 0   0 1
1 1 0   1 0
0 0 1   0 1
0 1 1   1 0
1 0 1   1 0
1 1 1   1 1
```

Add 3 1-bit numbers

```
a b ci = s co
0 0 0     0 0
0 1 0     0 1
1 0 0     0 1
1 1 0     1 0
0 0 1     0 1
0 1 1     1 0
1 0 1     1 0
1 1 1     1 1
```

Full Adder

8-bit adder