CSC2/452 Computer Organization
Integer Arithmetic, Real Numbers

Sreepathi Pai

URCS

September 14, 2022

Outline

Outline

Homeworks

- Homework due today in class (in box)
- Homework #2 is out today (due next Wednesday)
- Assignment #1, on bitwise operations will be out by end of week

Outline

What we have in our digital universe so far

- Two bits: 0 and 1
- Lots (but finite) of bitwise operations: AND, OR, NOT, XOR, ...
- Bit-level data structures
  - Bitsets
  - Bitfields
- Integers
  - A bitfield consisting of “sign” and ”magnitude”
  - Many representations (sign-magnitude, one’s complement, two’s complement, ...)

Outline

Integer operations

- So far we've looked at bitwise operations
- But for integers, we need familiar arithmetic operations
  - Addition, Subtraction, Multiplication, Division
  - Comparison operations (greater than, less than)
- We will investigate:
  - How to implement these operations using bitwise operations
  - How working with a finite number of bits can lead to surprises and non-mathematical behaviour

Comparisons: Decimal Numbers

- Which is greater?

N1	N2	N1 $>$ N2?
13789	00123	
13789	14000	
-3459	-0200	
+3400	-3450	

- What algorithm did you use (in your head)?

Comparisons: Binary Numbers

- Which is greater?
  - Assume 5 bit numbers
  - Highest bit is sign
  - Magnitude (4 bits) uses two's complement

N1	N2	N1 > N2?
01001	00101	
01001	01010	
11010	10001	
01000	11000	

- How did you handle cases where:
  - both numbers were positive?
  - both numbers were negative?
  - numbers with different signs?

Comparing two binary numbers a and b

- Assume two's complement representation
- Compare sign bits
- If equal, with sign bit $= 0$ (positive)
  - Search for first high bit that differs
  - If such a bit exists, $a > b$ if a has a 1 bit in that position
- If equal, with sign bit $= 1$ (negative)
  - Search for first high bit that differs
  - If such a bit exists, $a > b$ if a has a 1 bit in that position
- If not equal,
  - $a > b$ if a's sign bit is 0

How to compare bits

a	b	Output
0	0	1
0	1	0
1	0	0
1	1	1

- In C code, what are OP1 and OP2?

```
equal_bits = OP1(a OP2 b)
```

Pseudocode for Compare

```
bitequal(bit a, bit b) {
   return ~(a ^ b);
}

compare(num a, num b) {
   if(bitequal(signbit(a), signbit(b)) {
     tmp = a ^ b;  // clears all bits that are the same to 0

     if(tmp == 0)
        printf("a == b");
     else {
        pos = highest_one_bit(tmp);
        if(bitequal(signbit(a), signbit(b)))
            // a and b have the same sign
            if(bitequal(getbit(a, pos), 1))
               print("a > b"); // a has highest 1 bit
            else
               print("b > a"); // b has highest 1 bit
        else
           if(bitequal(signbit(a), 0))
               print("a > b");   // a is +ve, b is -ve
           else
               print("b > a");   // a is -ve, b is +ve
   }
}
```

In hardware

- In hardware this is implemented as a boolean circuit
- Collection of boolean gates
- Take ECE112 to learn more
- We will treat these as boxes (examples ahead)
- But we want to know how to do it in software
  - Even if it is slower

Binary Addition

- Note both Sum and Carry are outputs

a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- In C code, what are OP1 and OP2?

```
sum = a OP1 b;
carry = a OP2 b;
```

Binary Addition with Incoming Carry

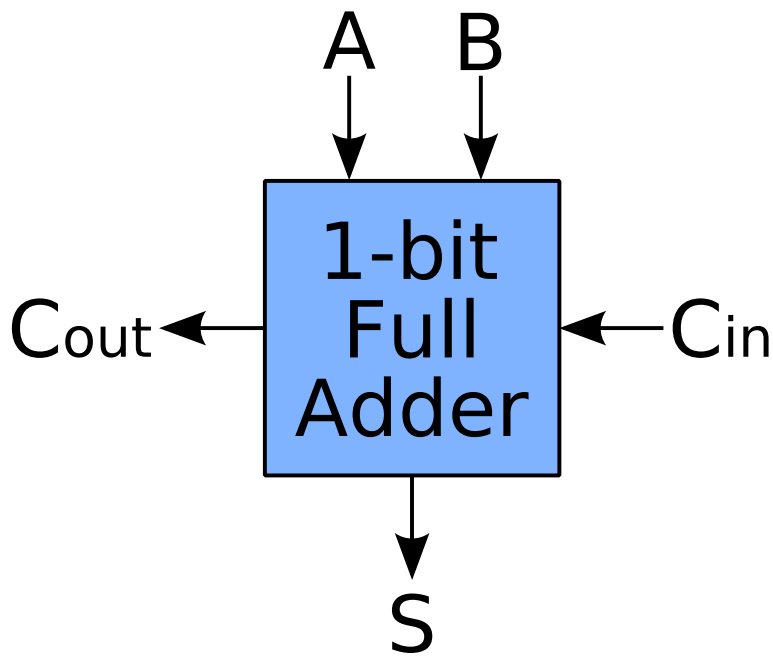
- Note both Sum and Carry are outputs

Carry_{in}	a	b	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- In C code:

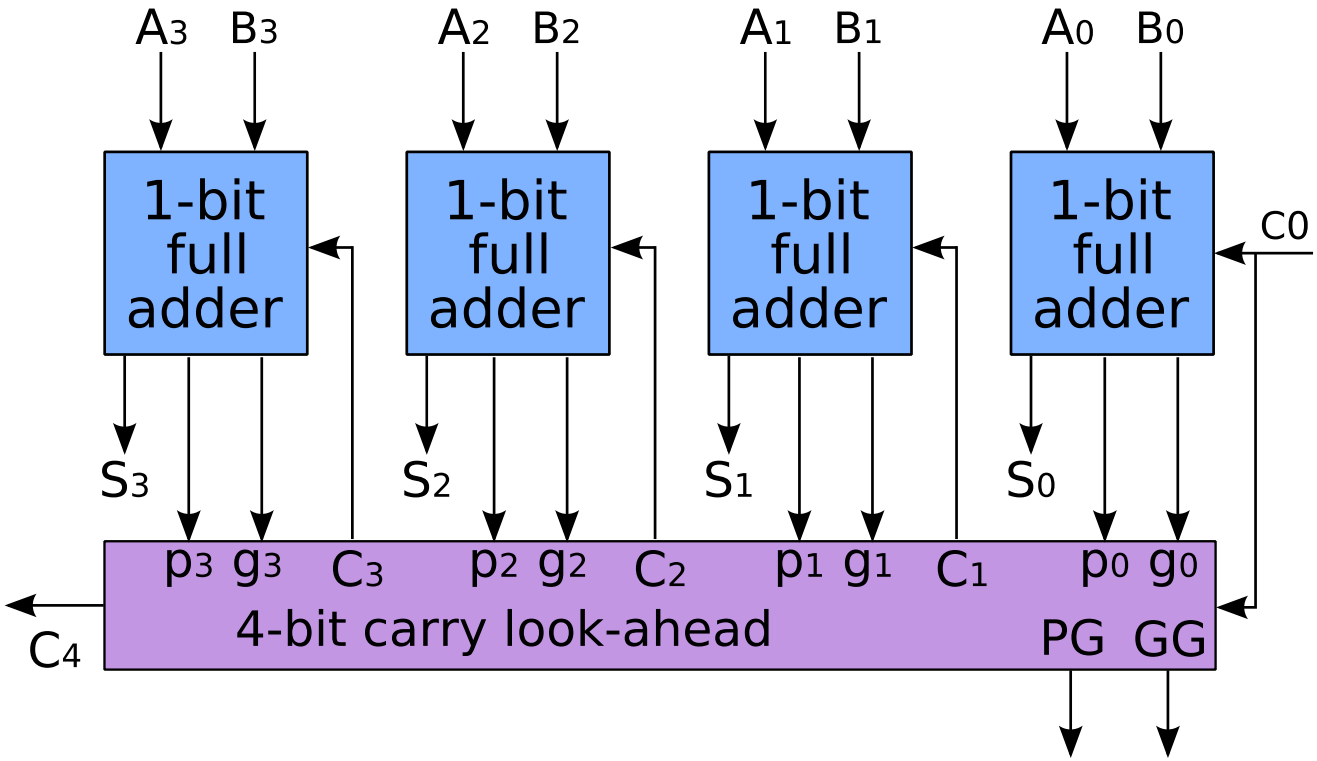
```
sum =
carry =
```

The Full Adder in Hardware (kinda)



- A, B, pairs of input bits
- C_{in} is carry in bit
- C_{out} is carry out bit
- S is sum bit
- Image Source: English Wikipedia, user:Cburnett, CC-BY-SA 3.0

Chaining the Full Adder



- g_n indicate this sum generates a carry (e.g. $1 + 1$)
- p_n indicates if this sum will propagate an incoming carry (e.g. $A = 1$ and $B = 0$)
- Image Source: English Wikipedia, user:Cburnett, CC-BY-SA 3.0

Buying a Full Adder

- You can get a hardware full adder as an integrated circuit for a few dollars
- Implements digital logic using electricity
  - 5V means 1
  - 0V means 0
- Link to Fairchild 7483 Datasheet

Binary Subtraction

$$9_{10} - 3_{10}$$

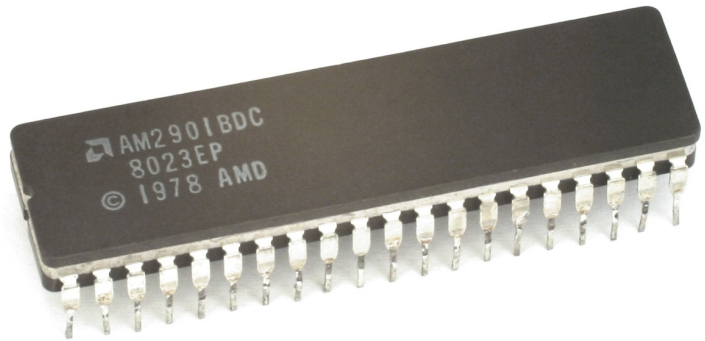
- Assume 5-bit two's complement:
  - 9 is 01001
  - 3 is 00011
  - two's complement of -3 is: $3 + x = 2^4$, so $x = 13$: 11101
- Now *add* 01001 and 11101
  - What do you get?

Multiplication and Division

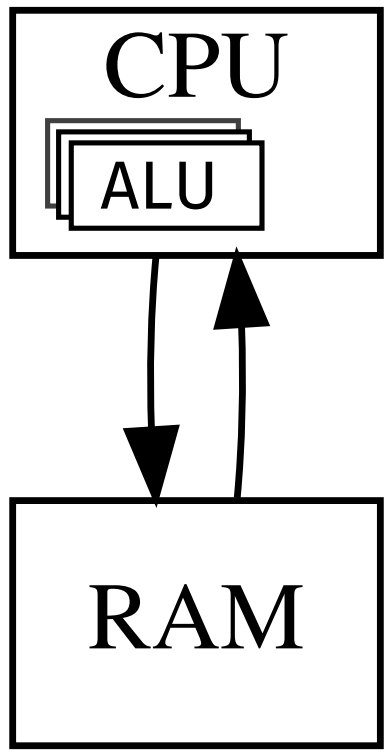
- Roughly speaking:
  - Multiplication is repeated addition
  - Division is repeated subtraction
- Can be optimized for certain cases
  - Multiplication replaced by left shift
  - Division replaced by right shift

Arithmetic Logic Unit

- Arithmetic Logic Units
  - Arithmetic operations
  - Logical Operations (bitwise)
- ALUs are the fundamental building block of computers
- Image at the right is the AMD 2901 4-bit ALU
  - Image source: By Konstantin Lanzet (with permission) - CPU collection Konstantin Lanzet, CC BY-SA 3.0



Revisiting our Box Diagram



- Multiple ALUs inside the CPU

Outline

Adding two integers of different widths

```
int8_t x = -5;
int16_t y = 10;

y = y + x;
```

- Shrink `y` to 8 bits?
  - What should we do to the top 8 bits?
- Expand `x` to 16 bits?
  - What should the new top 8 bits of `x` be?

C standard rules for integer conversion

- Terminology:
  - Promotion: creating a bigger int from a smaller int
  - Truncation: creating a smaller int from a bigger int
- By default, C always promotes `char` to `int` when operating on `char`.
  - Truncates the `int` back to `char` when storing it.
- For other conversions, C tries not to lose precision using notion of "rank":
  - Roughly, types with higher ranks have at least as much precision as types with lower ranks
  - Rankwise: `long int` > `int` > `short int` > `char`
  - Definition in the C standard is more exhaustive
- Conversions
  - Implicitly occur whenever operands of different types are operated upon
  - See rules on next slide

Rules for Integer Conversion of Two Operands[1]

- both same type, no further conversion is needed.
- both same integer type (signed or unsigned), lesser rank converted to greater rank
- if unsigned operand has higher or equal rank, signed operand is converted.
  - can change values if signed operand had negative value (which cannot be represented in unsigned)
- if signed operand can represent all of the values of the unsigned operand, unsigned operand is converted
- Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.
  - can change values

[1] https://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules

Revisiting our example

```
int8_t x = -5;
int16_t y = 10;

y = y + x;
```

- `x` is 8-bit, so integer promoted to `int`
- promoted `x` is same size as `y`
- operands are both of same type now, so no more conversions needed

What about new bits in `x`?

- Promotion of `x` creates 8 new high bits
- Two methods to initialize these new bits:
  - Zero extension: initialize them to zero
  - Sign extension: initialize them to value of the existing sign bit
- C uses sign extension when promoting to signed, zero extension otherwise
  - Note: all operands could be unsigned!
- Use a C cast to control conversions
  - e.g. `y + (int16_t) x`, explicitly casts `x` to `int16_t`

Shifting a signed integer to the right

```
int8_t x = -72;

x = x >> 3;
```

- What should be the value of high bits in the result?
- This right-shift operation on signed values is called shift *arithmetic* right

Unsigned integer overflow and underflow

```
/* overflow */
uint16_t x = 65535;
x = x + 1;

/* underflow */
uint16_t x = 0;
x = x - 1;
```

What happens with unsigned integers[2]

The C standard says:

A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

- "reduced modulo the number ...", for `uint16_t`, this number is 65536.
- So, `x = x + 1` is actually `x = (x + 1) % 65536` (where % represents modulo)
  - So `x = 0` ultimately
- For underflow, `x = x - 1`, a different rule is used
  - Add or subtract 65536 until it is representable by `uint16_t`
  - -1 is not representable, but $-1 + 65536$ is, so `x = 65535` ultimately
- Easiest to understand as wrapping around ($0 \rightarrow 65535 \rightarrow 0$)

[2] https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap

Shifting a signed integer to the right

```
int16_t x = -72;

x = x >> 17;
```

- What should `x` contain?

Signed integer overflow

```
/* overflow */
int16_t x = INT16_MAX;

x = x + 1;

/* underflow */
int16_t y = INT16_MIN;
y = y - 1;
```

- What should `x` overflow?
- What should `y` underflow to?

Undefined Behaviour

- For signed variables:
  - Shifting right by too many bits
  - Shifting left by too many bits
  - Overflowing
  - Underflowing
- Are all examples of *undefined behaviour*
  - The standard *refuses* to say what happens
- Note: these operations are defined at machine level
  - on Intel x86 processors, they wrap around, just like unsigned ints
  - but a C program cannot assume it is running on an x86

Undefined Behaviour can lead to unsound conclusions

- How to prove $1 = 2$
  - $a = 1$
  - $a = a^2$
  - $a - 1 = a^2 - 1$
  - $a - 1 = (a - 1)(a + 1)$
  - $1 = (a + 1)$ (dividing both sides by $a - 1$)
  - $1 = 2$ (substituting $a = 1$)

The quest for fast code

- Compilers want to generate as fast code as possible
- If your program uses undefined behaviour, its meaning is no longer defined
- The compiler is then free to do *whatever* it wants:
  - Remove code
  - Replace code
  - Move code
  - Burn your laptop
  - etc.
- Many security holes take advantage of undefined behaviour
  - Read the SEI CERT C Coding Standard
- Take CSC255 to find out more about compilers

Outline

Real Numbers

- $\mathbb{R}$
  - infinite (just like integers)
  - but they are different infinity (uncountable)
- There are infinite real numbers between any two real numbers
- How do we represent these using a finite, fixed number of bits?
  - Say, 32 bits

Representing Real Numbers

- We cannot represent real numbers accurately using a finite, fixed number of bits
  - But do we need infinite accuracy?
- How many (decimal) digits of precision do we use?
  - In our bank accounts (before and after the decimal point?)
  - In engineering?
  - In science?

On magnitudes

- Smallest length
  - Planck length, on the order of 10^{-35} (would require 35 decimal digits)
- Smallest time
  - Planck time, on the order of 10^{-44}
- Width of visible universe
  - On the order of 10^{24}
  - Lower bound on radius of universe: 10^{27}

On precision

- Avogadro's number: $6.02214076 \times 10^{23}$
  - So, actually: 602214076000000000000000
- $\pi = 3.1415... \times 10^{0}$
  - NASA requires about 16 decimal digits of π[3]
  - We know about a trillion

[3] https://blogs.scientificamerican.com/observations/how-much-pi-do-you-need/

Scientific notation for numbers

- The scientific notation allows us to represent real numbers as:

$$\text{significand} \times \text{base}^{\text{exponent}}$$

- For Avogadro's number:
  - Significand: 6.02214076
  - Significand is scaled so always only one digit before the decimal point
  - Base: 10
  - Exponent: 23

Binary Scientific Notation

- We can use scientific notation for binary numbers too:

$$1.011 \times 2^3$$

- Here, the number is:
  - $(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^3$
  - $(1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) = 11_{10}$
- Components:
  - Significand: 1.011
  - Base: 2
  - Exponent: 3

Binary Scientific Notation: Example #2

- Now with a negative exponent:

$$1.011 \times 2^{-3}$$

- Here, the number is:
  - $(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3}$
  - $(1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6})$
  - $(0.125_{10} + 0 + 0.0625_{10} + 0.03125_{10}) = 0.171875$
- Components:
  - Significand: 1.011
  - Base: 2
  - Exponent: -3

Some design notes

- Significand contains a radix point (i.e. decimal point or binary point)
  - But it's position is fixed: only one digit before the radix point
  - In binary scientific notation, this is always 1 (why?)
  - We don't need to store the radix point
  - So significand can be treated as an integer with an implicit radix point
- Base is always 2 for binary numbers
  - No need to store this
- Exponent is also an integer
  - Could be negative or positive or zero

Design notes (continued)

- So (binary) real numbers can be expressed as a combination of two fields:
  - significand (possibly a large number, say upto 10 decimal digits)
  - exponent (possibly a smallish number, say upto 44_{10})
  - would allow us to store numbers with at least 10 decimal digits of precision, upto 44 decimal digits long
- We'll also need to store sign information for the significand and the exponent
- How many bits?
  - for 10 significant decimal digits? e.g. 9,999,999,999
  - for max. exponent 50_{10}?
  - plus two bits for sign (one for significand, one for exponent)

Design notes (continued)

- How many bits?
  - for 10 significant decimal digits? e.g. 9,999,999,999: about 34 bits
  - for max. exponent 50? about 6 bits
  - plus two bits for sign (one for significand, one for exponent)
- Total: $34 + 6 + 2 = 42$ bits
  - **Could be implemented as a bitfield**
  - But 42 is between 32 and 64, not efficient to manipulate
- What format should we use to store negative significands and exponents?
  - sign/magnitude
  - one's complement
  - two's complement
  - other?

Bitfield Design Constraints

- Ideally should fit sign, significand and exponent in 32 bits or 64 bits
  - Easier to manipulate on modern systems
- Arithmetic operations should be fast and "easy"
- Comparison operations should be fast and "easy"
  - e.g. should not need to extract fields and compare separately
- Should satisfy application requirements
  - esp. with accuracy, precision and rounding
  - should probably be constraint #1