Representing Data

Dov Kruger

Department of Electrical and Computer Engineering
Rutgers University

November 7, 2024

Digital Representation of Information

- This module describes how to represent data as bits
- By the end you will be able to
  - Represent unsigned and signed Integers
  - Convert binary to/from decimal
  - Convert binary to/from hexadecimal
  - Compute twos complement numbers
  - Identify when integers overflow
  - Identify how to store text on computers
  - Identify how bits are interpreted as floating point

It's all Bits

- With binary data, it's all context
- It is not obvious what bits represent
- In this module you will lean how the computer represents
- whole numbers, fractions, and text

Bits and Bytes

- Bit: Smallest unit of data (0,1)
- Byte: Group of 8 bits
- Type of data completely depends on context

Combinations

- 2 bits: 00, 01, 10, 11
- 3 bits: 000, 001, 010, 011, 100, 101, 110, 111
- n bits $= 2^n$ combinations
- 8 bits $= 2^8 = 256$ combinations
- 16 bits $= 2^{16} = 65536$ combinations
- 32 bits $= 2^{32} = 4294967296$ combinations

Different Binary Data Representations

- Unsigned Integers
- Signed Integers
- ASCII text
- Unicode
- Fixed Point Fractions
- Floating Point

- Integers come in different sizes (8, 16, 32, 64 bits)
- Signed/Unsigned
- Little-endian/Big-endian

Representing 3 bit Integers

bits	unsigned	signed
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

What happens if we add 1 to 7?

What happens if we subtract 1 from 0?

Twos Complement Arithmetic

- Consider just 8 bit number for simplicity
- First bit is sign 0 = positive, 1 = negative
- To negate a number
  - invert all bits
  - add 1
  - the resulting number is the negative of the original
  - Example: $5 = 00000101 \rightarrow 11111010 \rightarrow 11111011 = -5$
  - Example: $17 = 00010001 \rightarrow 11101110 \rightarrow 11101111 = -17$
  - Example: $-11 = 11110101 \rightarrow 00001010 \rightarrow 00001011 = 11$
  - Example: $-128 = 10000000 \rightarrow 01111111 \rightarrow 00000000 = -128$

Overflow and Underflow

- Overflow is when the result of a computation is too large to fit
- Underflow is the same in the negative direction
- Example: given 3-bit unsigned
  - $3 + 5 = 1000 = 8 = 000 = 0$
  - $4 + 6 = 1010 = 10 = 010 = 2$
  - $4 - 5 = 111 = 7$
- Example: given 3-bit signed
  - $3 + 2 = 101 = -2$
  - $2 - 3 = 111 = -1$
  - $3 + 1 = 100 = -4$

Overflow and Underflow

- When a result is too large, store only the low n bits
- Example: 3 bits
    - $3 + 3 = 6$ (no overflow)
    - $4 + 4 = 8$ (too big) $= 0$ (overflow)
    - $3 - 2 = 1$ (no overflow)
    - $3 - 4 = -1 = 7$ (underflow)

Integer Data Types

bits		minval	maxval
8	signed	-128	127
8	unsigned	0	255
16	signed	-32768	32767
16	unsigned	0	65535
32	signed	-2147483648	2147483647
32	unsigned	0	4294967295
64	signed	-9223372036854775808	9223372036854775807
64	unsigned	0	18446744073709551615

Bases in General

A number in a base b is represented as a sum of powers of b

- Base 10 (digits 0-9): $123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$
- Base 2 (digits 0-1):
  $1111111 = 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$
- Base 8 (digits 0-7):
  $123 = 1 * 8^2 + 2 * 8^1 + 3 * 8^0 = 64 + 16 + 3 = 83$
- Base 16 (digits 0-9, A-F): $123 = 7 * 16^1 + 11 * 16^0$

Examples of Base numbers

- What is $(543)_8$ in decimal?
- What is $(101101)_2$ in decimal?
- What is $(3AB)_{16}$ in decimal?
- 78651 cannot be base 8, why not?

Every digit in octal represents 3 bits

Oct	bits	—	Oct	bits
0	000		4	100
1	001		5	101
2	010		6	110
3	011		7	111

$(123)_8 = 1 * 8^2 + 2 * 8^1 + 3 * 8^0 = 64 + 16 + 3 = 83$

$(456)_8 = 100101110$

Base 16: Hexadecimal

Hex	bits	—	Hex	bits
0	0000		8	1000
1	0001		9	1001
2	0010		A	1010
3	0011		B	1011
4	0100		C	1100
5	0101		D	1101
6	0110		E	1110
7	0111		F	1111

Encoding a byte in Hexadecimal

- D9 = 11011001
- AF = 10101111
- 8C = 10001100

Converting Between Binary and Decimal

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

- `Method 1: sum powers of 2`
- $10010010 = 128 + 16 + 2 = 146$
- `Method 2: start from left`
  - `Start with 1`
  - `For each digit, multiply by 2`
  - `If the digit is 1, add 1`
  - `Example:` $(((((1 * 2) * 2) * 2) + 1) * 2 * 2 * 2 + 1) * 2 = 146$

Integer Operations Overview

- Integer operations are fundamental in digital systems
- We'll cover arithmetic, logical, and shift operations
- Each operation will be demonstrated with 8-bit examples
- Understanding these operations is crucial for digital design

Addition

- Addition is performed bit by bit, carrying over when necessary
- Example (8-bit):

$$\begin{array}{r} \texttt{01101001} \\ + \ \texttt{00110110} \\ \hline \texttt{10011111} \end{array}$$

- Note: Overflow can occur if the result exceeds 8 bits

Subtraction

- Subtraction is often implemented as addition with two's complement
- Example (8-bit):

$$\begin{array}{r} 1\ \ 11 \\ 01101001 \\ 00110110 \\ \hline 00010011 \end{array}$$

- Two's complement of 00110110 is 11001010

Bitwise AND

- AND operation: 1 if both bits are 1, otherwise 0
- Example (8-bit):

```
  01101001
& 00110110
----------
  00110000
```

- Often used for masking specific bits

Bitwise OR

- OR operation: 1 if either bit is 1, otherwise 0
- Example (8-bit):

$$
\begin{array}{r}
01101001 \\
\mid\ 00110110 \\
\hline
01111111
\end{array}
$$

- Useful for setting specific bits

Bitwise XOR

- XOR operation: 1 if bits are different, 0 if same
- Example (8-bit):

$$\begin{array}{r} 01101001 \\ \hat{0}0110110 \\ \hline 01011111 \end{array}$$

- Often used in cryptography and error detection

Bitwise NOT

- Inverts all bits: 1 becomes 0, 0 becomes 1
- Example (8-bit):

$$\begin{array}{r} \sim\ 01101001 \\ \hline 10010110 \end{array}$$

- Useful for flipping specific bits

Logical Left Shift

- Shifts all bits to the left, filling rightmost with 0
- Example (8-bit, shift by 2):

$$\frac{01101001 \;\; << \;\; 2}{10100100}$$

- Equivalent to multiplication by 2^n for n shifts

Logical Right Shift

- Shifts all bits to the right, filling leftmost with 0
- Example (8-bit, shift by 2):

$$\begin{array}{r} \texttt{01101001} \ \ll \ \texttt{2} \\ \hline \texttt{10100100} \quad\quad\quad \end{array}$$

- Equivalent to multiplication by 2^n for n shifts (unsigned)

Rotate Left

- All bits move to the left
- Bits that drop off the end come in on right
- Example (8-bit, rotate by 2):

```
 01101001 ROL 2
----------------
 10100101
```

- Preserves all bits, useful in cryptography

Rotate Right

- Shifts all bits to the right, wrapping around
- Example (8-bit, rotate by 2):

```
 01101001 ROR 2
----------------
 10100101
```

- Example (8-bit, rotate by 2):
- Also preserves all bits, used in various algorithms

Arithmetic Left Shift

- Shifts bits to left just like logical shift
- If sign changes, it's an overflow
- Example (8-bit, shift by 2):

$$\frac{\texttt{01101001} \;\; <<< \;\; \texttt{1}}{\texttt{11010010}}$$

- Equivalent to multiplication by 2^n for n shifts
- Can cause overflow if significant bits are shifted out

Arithmetic Right Shift

- Shift bits right preserving sign
- Leftmost bits are filled with the sign bit
- Example (8-bit, shift by 2):

$$\frac{\texttt{11101001} \ >>> \ \texttt{1}}{\texttt{11110100}}$$

- For positive numbers, equivalent to division by 2^n for n shifts
- For negative numbers, rounds towards negative infinity
- Preserves the sign of the number

Bitwise Operations in Verilog

- Verilog provides operators for various bitwise operations
- These operations are fundamental in digital design
- Examples (assuming 8-bit variables a and b):

```
c = a & b;                      // Bitwise AND
d = a | b;                      // Bitwise OR
e = a ^ b;                      // Bitwise XOR
f = ~a;                         // Bitwise NOT
g = a << 2;                     // Logical left shift
h = a >> 2;                     // Logical right shift
i = $signed(a) >>> 2;           // Arithmetic right shift
j = {a[5:0], a[7:6]};           // Rotate left
k = {a[1:0], a[7:2]};           // Rotate right
```

ASCII Encoding

- ASCII: American Standard Code for Information Interchange
- Maps characters to binary values
- ASCII table overview `https://www.ascii-code.com/`
- Example: 'A' $= 01000001 = 65$
- Example: 'B' $= 01000010 = 66$
- Example: 'a' $= 01100001 = 97$

Unicode UTF-8 Encoding

- Unicode is a standard for representing text of all languages
- Originally fit into 16 bits
- Now requires 19 bits because of emoji
- UTF-8 is a variable length encoding
- Useful when most of the text is English
- `https://symbl.cc/en/unicode-table/`
- `https://r12a.github.io/app-conversion/`

1st byte	2nd byte	3rd byte	4th byte	19-bit value
0xxxxxxx				000000000000000xxxxxx
110yyyyy	10xxxxxx			0000000000yyyyyxxxxxx
1110zzzz	10yyzzzz	10yyyyyy	10xxxxxx	00000zzzzyyyyyyxxxxxx
11110uuu	10uuzzzz	10yyyyyy	10xxxxxx	uuuuuzzzzyyyyyyxxxxxx

Unicode UTF-8 Encoding

Let's decode the following bytes

```
00000000: 74 65 73 74 0a ce b5 cf 85 cf 87 ce b1 cf 81 ce b
00000010: cf 83 cf 84 cf 8e 0a E5 A4 A7
```

- 74 = 01110100 starts with 0, so look up ASCII
- same through 0A (ASCII for linefeed)
- CE B5 = **110**01110 **10**110101 starts with 110, so it's a 2-byte char
- bits = 01110 110101 = 011 1011 0101 = 3B1
- Look up 3B1 in Unicode online (it's a Greek letter)
- E5 A4 A7 = **1110**0101 **10**100100 **10**100111 starts with 1110, so it's a 3-byte char
- bits = 0101 100100 100111 = 5 9 2 7
- Look up 0x5927 in Unicode online (Chinese)

Fixed Point Fractions

$2^3 \mid 2^2 \mid 2^1 \mid 2^0 \mid . \mid 2^{-1} \mid 2^{-2}$

- Fractions in binary are represented as negtive powers
- $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$
- Examples

$$
\begin{aligned}
101.1 &= 4 + 1 + \tfrac{1}{2} = 5.5 \\
1.01 &= 1 + \tfrac{1}{4} = 1.25 \\
110.11 &= 4 + 2 + \tfrac{1}{2} + \tfrac{1}{4} \\
1001.001 &= 8 + 1 + \tfrac{1}{8}
\end{aligned}
$$

Floating Point

- Fixed point represents fractions, but only a single size
- Floating point can represent values wildly different values
- IEEE-754

Single precision	32 bits
Double precision	64 bits
Quad Precision	128 bits (not yet in hardware)
Half Precision	16 bits (GPUs)
fp8	8 bits (GPUs)

float (32-bit) Representation

value	seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm	hex
1.0	00011111100000000000000000000000	3f800000
2.0	01000000000000000000000000000000	40000000
1.5	00011111110000000000000000000000	3fc00000
0.1	00111101110011001100110011001101	3dcccccd
1234567.8	01001001100101101011010000111110	4996b43e
1.2345678e+10	01010000001101111111011100000110	5037f706
6.023e+23	01100110111111110001010101111111	66ff157f
6.674e-11	00101110100100101100001101001000	2e92c348
-1234567.8	11001001100101101011010000111110	c996b43e
1.234e-30	00001101110010000011101001011011	0dc83a5b
NaN	01111111110000000000000000000000	7fc00000
Infinity	01111111100000000000000000000000	7f800000
-Infinity	11111111100000000000000000000000	ff800000

Infinity

- Infinity is a special value
- Too large to be countable
- From calculus: $\lim_{x \to 0} \frac{1}{x} = \infty$
- Example: $1.0/0.0 = \infty$
- Not countable: $1.0/0.0 = 2.0/0.0$
- Negative infinity: $-1.0/0.0 = -\infty$

NaN: Not a Number

- NaN is a special value meaning the answer is unknown
- When two opposing infinities fight, we don't know the answer
- Example $1.0/0.0 - 1.0/0.0 =$ NaN
- Example: $0.0/0.0 =$ NaN

NaN: Not a Number

For each of the following, determine if the result is NaN, Infinity, or a number

$x = 5.0/0.0$	
$y = 1.0/x$	
$z = \sqrt{x}$	
$w = \sin(x)$	
$v = -3.5/0.0$	
$u = \sqrt{v}$	