Fixed Length & Signed Magnitude Representation

Textbook Section 2.2

Fixed Length Binary Numbers

- Previously, binary numbers of unlimited length
- Unrealistic in a computer due to fixed length of memory locations
- Instead use fixed length binary numbers
    - Common sizes are 8, 16, 32, and 64 bits

Diagram: 8-bit Unsigned Binary Number

00000000

Bit Position 7
Most Significant Bit (MSBit)

Bit Position 0
Least Significant Bit (LSBit)

	Binary	Octal	Hex	Decimal
Minimum Value	00000000	000_{8}	00_{16}	0_{10}
Maximum value	11111111	377_{8}	FF_{16}	255_{10}

Fixed Length Binary Numbers

For an n-bit fixed length number

- Can represent 2^n values
- Range of values represented is 0 to 2^n-1
    - The **-1** is because 0 is included in the range

# of Bits	Range of Values
16	0 to 65,535
32	0 to 4,294,967,295
64	0 to 14,446,744,073,709,551,615

Fixed Length Binary Numbers

What happens with a fixed length number if we:

- increment the largest value?
- decrement the smallest value?
- add two values whose result exceeds the largest?
- subtract a larger number from a smaller number?
  - The carry out from the MSBit goes to the magic bitbucket
  - A carry in to the MSBit comes from the magic bitbucket
  - Wrap program on INS in 2655/examples

Fixed Length Binary Numbers

- All bits in a memory location have a value
- Values are 0 or 1
- Therefore no undefined bit values
- Like an odometer:
  - Exceed the highest value causes the results to wrap around

Signed Binary Numbers

- Signed numbers must be representable
    - Two programming courses have shown this
- How could negative (signed) numbers be represented with the same N bits?
- Three main methods were developed for representing signed numbers
- All methods use a <u>known fixed length</u>

Signed-Magnitude

- To represent negative arbitrary-length base-10 numbers you just put a negative sign in front of the highest digit
    - e.g.   -10
    - Clear, easy to notate, works with arbitrary length numbers
- On a computer there's no way to represent a negative sign
    - A sign has only two possibilities + or –, only need **one** bit!
- Usually the Most Significant Bit (MSBit) is sacrificed as the sign bit, where:
        - Zero means a positive number
        - One means a negative number

Signed-Magnitude

Conversion – Base-10 to binary of fixed length N

- Convert the absolute value of the original number to binary
- if the binary # is < N bits then store it in a fixed length of N-1
  - this ensure it fits into N-1 bits
  - add leading zeros if needed to get to N-1 bits
- else can’t be represented
- if the original number is negative then set MSBit = 1
- else set MSBit = 0 (number has to be positive)

Signed-Magnitude

Convert 12_{10} to 8-bit signed magnitude binary (SM)

1. $\text{Abs}(12_{10}) = 12_{10} = 8 + 4 = 1100_2$
2. 12_{10} requires 4 bits in binary which is <= 7, so can be represented
   - Since 4 < 7 need to add 3 leading zeros to get 0001100_2
3. The original number is positive so the sign bit (bit 7) is 0
4. Combining the sign bit and value (magnitude) the answer = 00001100_2 SM

Signed-Magnitude

Convert -12_{10} to 8-bit signed magnitude binary (SM)

1. $Abs(-12_{10}) = 12_{10} = 8 + 4 = 1100_2$
2. 12_{10} requires 4 bits in binary which is <= 7, so can be represented
   - Since 4 < 7 need to add 3 leading zeros to get 0001100_2
3. The original number is negative so the sign bit (bit 7) is 1
4. Combining the sign bit and value (magnitude) the answer = 10001100_2 SM

Signed-Magnitude - Practice

Convert 36_{10} to an 8-bit signed magnitude binary number

1. Abs(36_{10}) = 36_{10} = 100100_2
2. 36_{10} requires 6 bits in binary which is <= 7, so can be represented
   - Since 6 < 7 need to add 1 leading zero to get 0100100_2.
3. The original number is positive so the sign bit (bit 7) is 0
4. Combining the sign bit and value (magnitude) and answer = 00100100_2 SM

Signed-Magnitude - Practice

Convert -75 to an 8-bit signed magnitude binary number

1. Abs(-75_{10}) = 75_{10} = 64 + 8 + 2 + 1 = 1001011_2
2. 75_{10} requires 7 bits in binary which is <= 7, so can be represented
   - 7 bits already used, no leading zeros added
3. The original number is negative so the sign bit (bit 7) is 1
4. Combining the sign bit and value (magnitude) the answer = 11001011_2 SM

Signed-Magnitude

Convert 149 to an 8-bit signed magnitude binary number

1. Abs(149_{10}) = 149 = 128 + 16 + 4 + 1 = 10010101_2
2. 149_{10} requires 8 bits, 8 > 7
   - 149_{10} cannot be represented as an 8-bit signed magnitude number

Signed-Magnitude

Conversion – signed magnitude binary of fixed length N to Decimal

1. Split binary number into MSBit and N-1 bit positive binary number
2. answer = N-1 bit number converted to decimal
3. if MSBit(number) = 1 then
   answer = -1 * answer

Signed-Magnitude

Convert 10111010_2 SM to decimal

Divide into sign & magnitude → $1\ 0111010_2$

Convert the magnitude $0111010_2 = 32 + 16 + 8 + 2 = 58_{10}$

The MSBit is 1 so the number is negative

therefore answer = -58_{10}

Signed-Magnitude - Practice

Convert 11000100_2 SM to Decimal

1. Sign-bit = 1, Number = 1000100_2
2. 1000100_2 = 64 + 4 = 68_{10}
3. Since sign-bit = 1

   Answer = -1 * 68_{10} = -68_{10}

Signed-Magnitude - Practice

Convert 00110100_2 SM to Decimal

1. Sign-bit = 0, Number $=0110100_2$
2. $0110100_2 = 32 + 16 + 4 = 52_{10}$
3. Since sign-bit = 0

   Answer = 52_{10}

Signed-Magnitude

- A signed fixed-length number using signed-magnitude from
    - **$-(2^{(N-1)} - 1)$ to $2^{(N-1)} - 1$**
- Out of the **2^N** values half are positive and half are negative
    - Hence the $2^{(N-1)}$ in the above range $2^N/2 = 2^{(N-1)}$
- What do the following 8-bit binary SM values represent
    - `00000000 = +0`
    - `10000000 = -0`
    - This is the reason the **-1** in both parts of the range

Signed-Magnitude - Math

Examples in base-10

```
      +8    -8    -253
+    +3+   -3+   +762
```

When doing the addition do you add the signs? **NO**

Same is true when doing math on Sign Magnitude values

Signed-Magnitude - Addition Rules

Separate the numbers into sign and magnitude

- **IF** the signs are equal **THEN**
  - add the two magnitudes and the sign remains the same
  - if the result is > N-1 bits then an invalid result
- **ELSE**
  - subtract the smaller magnitude from the larger magnitude
  - the sign of the result is the sign of the larger magnitude
- For *Subtraction* think through the rules you know and use

Signed Magnitude - Addition Example

Add the following 8-bit SM binary values

```
  00000010
+ 10000101
  0 0000010
  1 0000101     note: different signs, subtract smaller value from
                      larger value (needs comparison)

    0000101
-   0000010
    0000011     result sign is from "larger" value
```

10000011_2 SM

Signed-Magnitude

Pros:

✓ easy for humans to understand

Cons:

× arithmetic circuits must handle multiple “sign” cases, so more complex

× arithmetic circuits must handle both “zero” cases, so more complex

× requires adder, subtractor, comparator