Lecture Topics

- Today: Integer Arithmetic
  (H&H 5.1-5.2)
- Next: Floating Point Arithmetic
  (H&H 5.3)

Announcements

- Self-study Module #4
- Project #4 (due Friday, 5/26)
- Exam #1 (Wednesday, 5/31)
Exam #1

- Wednesday, 5/31 during lecture
- 20% of course grade
- Bring MSU ID
- One 8.5x11 note sheet (both sides) allowed
- Study suggestions on course website

Fundamental Data Types

Computing systems can directly process:

- Boolean data
- Character data
- Unsigned integer data
- Signed integer data
- Floating point data

First four processed in integer circuits, last one processed in floating point circuits
Signed Integers

Several different systems developed:

- Sign and magnitude (signed magnitude)
- One’s complement
- Two’s complement

Half of the bit patterns are used to represent negative numbers

Two’s Complement

Negative representation formed by applying two’s complement operation to all bits in positive representation (flip all bits, add 1)

Example (assume 8 bits):

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>+25</td>
<td>00011001</td>
</tr>
<tr>
<td>-25</td>
<td>11100111</td>
</tr>
</tbody>
</table>
Two’s Complement

Range (assume 8 bits):

max 01111111 +\(2^7 - 1\)
min 10000000 -(2^7)

One representations of zero (assume 8 bits):

+0 00000000
-0 00000000

Example: -22 base 10

Internal representation (8 bits):

+22: 00010110
flip bits: 11101011
add 1: 1
--------
-22: 11101010
Summary: Signed Integers

Two’s complement representation used on most microprocessors

In C/C++, type "int" is typically 32 bits wide

Internal representation

~cse320/Examples/example04.pdf

Operations on Integers

Arithmetic operations:

- NEG (negation)
- ADD (addition)
- SUB (subtraction)
- MUL (multiplication)
- DIV (division)

Negation is a unary operation (one operand)
Operations on Integers

Bitwise operations:

- NOT
- AND
- OR
- XOR

NOT is a unary operation (one operand)

Shift operations:

- SLL (shift left logical)
- SRL (shift right logical)
- SRA (shift right arithmetic)

No need for “shift left arithmetic” – same as “shift left logical”
Integer Circuits

Math unit:
  addition and subtraction
  bitwise operations

Shift unit:
  shift operations

Multiply and Divide unit:
  multiplication and division

Arithmetic Operations

Assume two's complement for signed integers
  - Negation: flip all bits, add 1
  - Addition: ripple-carry adder (or faster alternative)
  - Subtraction: variation on addition
  - Multiplication: sequential logic
  - Division: sequential logic
Two’s Complement Addition

Addition of two’s complement values is the same as addition of unsigned values.

One approach: ripple carry addition:

\[
\begin{align*}
00011100 & \quad \text{carry bits} \\
00111101 \\
+ 00010110 \\
\hline
00110011
\end{align*}
\]

Full Adder

<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
<th>Cout</th>
<th>S(um)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Ripple-Carry Adder (4 bits)

```
<table>
<thead>
<tr>
<th></th>
<th>A3</th>
<th>B3</th>
<th>A2</th>
<th>B2</th>
<th>A1</th>
<th>B1</th>
<th>A0</th>
<th>B0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry</td>
<td>C</td>
<td>FA</td>
<td>Cin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|       | ---
```

Example: (+5) + (+6)

```
(+5)    00000100
+ (+6)  + 00000101
----    + 00000110
(+11)  00001011
```
Example: (+5) + (-6)

\[
\begin{array}{ccc}
(+5) & 00000000 \\
+ (-6) & + 0000101 \\
---- \\
(-1) & 11111111
\end{array}
\]

Example: (-5) + (-6)

\[
\begin{array}{ccc}
(-5) & 11111010 \\
+ (-6) & + 11110111 \\
---- \\
(-11) & 11110101
\end{array}
\]
Example: \((-5) + (+6)\)

\[
\begin{array}{c}
11111110 \\
(\text{-}5) \quad 11111011 \\
+ \quad (+6) \quad + \quad 00000110 \\
---- \\
(+1) \quad 00000001
\end{array}
\]

Summary

\[
\begin{array}{ccc}
00000100 & 11111010 \\
(+5) & 00000101 & (-5) \quad 11111011 \\
(+6) & + & 00000110 & (-6) \quad + \quad 11111010 \\
---- & ------ & ---- & ------ \\
(+11) & 00001011 & (-11) \quad 11110101
\end{array}
\]

\[
\begin{array}{ccc}
00000000 & 11111110 \\
(+5) & 00000101 & (-5) \quad 11111011 \\
(-6) & + & 11111010 & (+6) \quad + \quad 00000110 \\
---- & ------ & ---- & ------ \\
(-1) & 11111111 & (+1) \quad 00000001
\end{array}
\]
Overflow

Overflow occurs when the result is too large.

\[
\begin{array}{c}
01111110 \\
01111111 \\
+ 00000010 \\
\hline
10000001
\end{array}
\quad
\begin{array}{c}
10001110 \\
11011111 \\
+ 10000010 \\
\hline
01010001
\end{array}
\]

Result would be OK if there were more bits available to store the sum.

Detecting Overflow

Human: sign of result is different than sign of operands (overflow cannot occur if signs of operands are different)

Machine: carry into most significant bit (sign bit) is different than carry out of MSB

Test: \( C_{n-2} \text{ XOR } C_{n-1} \)
Examples:

\[
\begin{array}{c}
01111110 \\
01111111 \\
+ 00000010 \\
\hline \\
10000001
\end{array}
\] overflow

\[
\begin{array}{c}
10001110 \\
11001111 \\
+ 10000010 \\
\hline \\
01010001
\end{array}
\] overflow

Examples:

\[
\begin{array}{c}
00000100 \\
(+5) 00000101 \\
(+6) + 00000110 \\
\hline \\
(+1) 00001011
\end{array}
\]

\[
\begin{array}{c}
11111011 \\
(-5) 11111011 \\
(+6) + 00000010 \\
\hline \\
(+11) 00001011
\end{array}
\]

\[
\begin{array}{c}
00000000 \\
(+5) 00000101 \\
(-6) + 11111010 \\
\hline \\
(-1) 11111111
\end{array}
\]

\[
\begin{array}{c}
11111010 \\
(-5) 11111011 \\
(+6) + 00000110 \\
\hline \\
(+11) 00001011
\end{array}
\]
Two’s Complement Subtraction

One approach: circuit to perform ripple borrow subtraction

Alternative: use existing ripple carry adder

\[
A - B = A + (-B) = A + (~B + 1)
\]

Negate B by flipping all the bits, adding 1

Ex: \((+5) - (+6) = (+5) + (~6+1)\)

\[
\begin{array}{c}
1 \\
(+5) \\
+ (~6) \\
\hline
(-1)
\end{array} \quad \begin{array}{c}
00000011 \\
0000101 \\
1111001 \\
\hline
11111111
\end{array}
\]
Ex: \((-5) - (+6) = (-5) + (~6) + 1\)

\[
\begin{array}{cccc}
1 & 11111011
\hline
(-5) & 11111011
+ (~6) & + 1111001
----- & -------
(-11) & 11110101
\end{array}
\]

Combined Adder/Subtractor (4 bits)

SUB control signal: 0 means Add, 1 means Subtract
Status Bits

Useful to track status with Adder/Subtractor

NZCV bits:

- N: result was negative
- Z: result was zero
- C: copy of carry out
- V: overflow occurred

Determining NZCV bits:

- N: copy of $S_{n-1}$ (most significant bit of result)
- Z: NOR of all bits in result
- C: copy of $C_{n-1}$
- V: XOR of $C_{n-2}$ and $C_{n-1}$ (last two carrys)
Operations on Integers

Bitwise operations:

- **NOT**
- **AND**
- **OR**
- **XOR**

NOT is a unary operation (one operand)
Application: process ASCII chars

ASCII uses 7 bits – when stored in a byte of memory, room for 1 parity bit.

Even parity: total number of 1’s in byte is an even number (force with parity bit).

Examples:

'B' = 0x42  P1000010 (set P to 0)
'C' = 0x43  P1000011 (set P to 1)

Remove parity bit:

\[
\begin{align*}
\text{PXXXXXXX} & \quad \text{(generic ASCII char)} \\
\text{AND} & \quad 01111111 \quad \text{(mask)} \\
\hline \\
\text{0XXXXXXX} & \quad \text{(char without parity bit)}
\end{align*}
\]

Example:

\[
11000001 \text{ AND } 01111111 = 01000001 \ ('A')
\]
Application: process ASCII chars

Convert case (lower to upper):

\[ 011xxxxx \quad \text{(lower case ASCII char)} \]
\[ \text{AND} \quad 01011111 \quad \text{(mask)} \]
\[ \text{--------} \]
\[ 0x0xxxxx \quad \text{(upper case ASCII char)} \]

Example:
\[ 01100010 \quad \text{AND} \quad 01011111 = 01000010 \ ('B') \]

Application: process ASCII chars

Convert case (upper to lower):

\[ 010xxxxx \quad \text{(upper case ASCII char)} \]
\[ \text{OR} \quad 00100000 \quad \text{(mask)} \]
\[ \text{--------} \]
\[ 011xxxxx \quad \text{(lower case ASCII char)} \]

Example:
\[ 01000011 \quad \text{OR} \quad 00100000 = 01100011 \ ('c') \]
Bitwise Operators in C/C++

C/C++ bitwise operations:

- **NOT**  ~
- **AND**  &
- **OR**   |
- **XOR**  ^

Examples:

- `~cse320/Examples/example05.pdf`
- `~cse320/Examples/example06.pdf`

Bitwise Circuits

Example: 8-bit AND
Status Bits

Determining NZCV bits:

- **N**: copy of \( S_{n-1} \) (most significant bit of result)
- **Z**: NOR of all bits in result
- **C**: 0 (not meaningful)
- **V**: 0 (not meaningful)

Shift Operations

Shift operations move bit patterns to the left or the right within a register.

Example (assuming 8 bits):

\[
\begin{align*}
\text{00110011} \\
\text{SLL 2:} & \quad \text{11001100}
\end{align*}
\]

Shift count: 0 to \( N-1 \) (where \( N \) is number of bits)
Shift Left Logical (SLL)

Move bit pattern to the left, fill with zeroes on the right.

Examples (assuming 8 bits):

\[
\begin{align*}
00110011 & \quad 11001010 \\
\text{SLL 1: } & \quad 01100110 \\
\text{SLL 5: } & \quad 01000000 \\
\end{align*}
\]

Shift Right Logical (SRL)

Move bit pattern to the right, fill with zeroes on the left.

Examples (assuming 8 bits):

\[
\begin{align*}
00110011 & \quad 11001010 \\
\text{SRL 1: } & \quad 0011001 \\
\text{SRL 5: } & \quad 000000110 \\
\end{align*}
\]
Shift Right Arithmetic (SRA)

Move bit pattern to the right, fill with copies of the sign bit on the left.

Examples (assuming 8 bits):

00110011            11001010

SRA 1: 00011001       SRA 5: 11111110

SRA preserves the original sign of the value.

Application: count bits

Use masking and shifting to count number of bits which are 1 in an ASCII character:

\[
\text{total} = 0 \\
\text{loop 7 times:} \\
\quad \text{mask off least significant bit} \\
\quad \text{if bit is a 1, increment total} \\
\quad \text{shift char right by 1 position}
\]
Bitwise Operators in C/C++

C/C++ shift operations:

- **SLL**  `<<`  (when value is unsigned)
- **SRL**  `>>`  (when value is unsigned)
- **SRA**  `>>`  (when value is signed)

Example:

~cse320/Examples/example07.pdf

Shift Circuits

Logarithmic barrel shifter: for N bit register, use \( \log_2 N \) levels of multiplexers

Example: 8 bit register ==> 3 levels

1\(^{st}\) level – shift 4 bits
2\(^{nd}\) level – shift 2 bits
3\(^{rd}\) level – shift 1 bit
8-bit right logical shifter

Combined shift circuits

Control signal (2 bits):

- No shift
- Shift left logical
- Shift right logical
- Shift right arithmetic
Multiplication and Division

Require several steps

Simple versions use sequential logic

Multiply: \( N \text{ bits} \times N \text{ bits} \Rightarrow 2N \text{ bits} \)

Divide: \( 2N \text{ bits} / N \text{ bits} \Rightarrow N \text{ bits} \)

Unsigned Multiplication

Example: 4 bits \( \times \) 4 bits \( \Rightarrow \) 8 bits

\[
\begin{array}{c}
\text{1101} & \text{Multiplicand} \\
\text{MUL 1011} & \text{Multiplier} \\
\hline
\text{1101} & \text{Partial products} \\
\text{1101} \\
\text{0000} \\
\text{1101} \\
\hline
\text{10001111} & \text{Product}
\end{array}
\]
Unsigned Serial Multiplication

Unsigned Serial Division
Summary

Serial multiply (and divide) require sequence of steps

Signed multiply (and divide) must account for sign bits

More complex circuits can reduce number of steps

Programming in C

C is the parent of C++ (and thus a subset)

C++ features which are not part of C:
- Classes (and thus no I/O classes)
- Templates (and thus no STL)

One other minor difference: C does not support local variables in "for" statements

```c
for (int i=0; i<10; i++) // illegal
```
I/O in C

No classes, so no stream class library. Thus, all I/O must be done via functions.

Example:

```c
printf( "Sum: %d\n", total );
```

Rich set of functions in the standard library; most programmers only use a few.

---

Standard Streams

stdin – standard input stream
stdout – standard output stream
File Streams

Function `fopen()` used to create a file object, interact with the operating system’s file system.

Character I/O

Returns one character (one byte, converted to four bytes) from the standard input stream. Returns -1 when end-of-file is true.

```
    int getchar();
```

Sends one character (four bytes, converted to one byte) into the standard output stream. Returns a copy of the character (-1, if an error occurs).

```
    int putchar( int );
```
#include <stdio.h>

int main()
{
    int input;

    for (;;) {
        input = getchar();

        if (input == EOF) break;

        putchar( input );
    }

    return 0;
}

Translate (compile and link) the program. Then, execute it and supply input from the keyboard.

<2 north:~/Examples > gcc -Wall example08.c

<3 north:~/Examples > a.out
This is the first line.
This is the first line.
And this is the last line.
And this is the last line.
^D
Execute it again and use input redirection to supply input from a file.

```bash
<4 north:/~Examples > a.out < example08.c
```

```c
/*********************************************************
Example #8 -- Demo the use of "getchar" and "putchar"
************************************************************************/

#include <stdio.h>

int main()
{
    int input;
    ...

(effire contents of the file, character-by-character)
```

---

**I/O Buffering (input stream)**

Function **`getchar()`** waits until the buffer is not empty, fetches the next character and updates the pointer to the current item. OS routines fill the buffer when it is empty.
I/O Buffering (input stream)

The only difference with the standard input stream: the OS routines echo print the characters as they are copied into the buffer. Copying a newline signals "buffer ready".

I/O Buffering (output stream)

Function `putchar()` waits until there is room in the buffer, then stores the next character and updates the pointer to the current item. OS routines empty the buffer when it is full.
I/O Buffering (output stream)

Executing program \texttt{putchar()} \rightarrow \texttt{OS} \rightarrow \texttt{monitor}

No significant difference with the standard output stream (some output functions flush the buffer themselves, such as the \texttt{endl} manipulator in C++).

Formatted I/O

Reads characters from the standard input stream, converts substrings according to the format string. Returns the number of successful conversions.

\begin{verbatim}
int scanf( const char *form, ... );
\end{verbatim}

Converts items into characters strings according to the format string, sends them to the standard output stream. Returns the number of characters sent.

\begin{verbatim}
int printf( const char *form, ... );
\end{verbatim}
int main()
{
    const char A = '?'
    const short B = 19
    const int C = 28

    printf( "A: %2c %2d %2x\n\n", A, A, A);
    printf( "B: %2d %2x %04x\n\n", B, B, B);
    printf( "C: %2d %2X %08X\n\n", C, C, C);
}

A:  ? 63 3f
B: 19 13 0013
C: 28 1C 0000001C

int main()
{
    const float D = 23.5
    const double E = 16.25
    const char F[] = "CSE 320"

    printf( "D: %f %5.2f %e %6.4e\n\n", D, D, D, D);
    printf( "E: %f %5.2f %E %6.4E\n\n", E, E, E, E);
    printf( "F: >%s< >%10s< >%-10s\n\n", F, F, F);
}

D: 23.500000 23.50 2.350000e+01 2.3500e+01
E: 16.250000 16.25 1.625000E+01 1.6250E+01
F: >CSE 320< > CSE 320< >CSE 320 <
int main()
{
    int Count, X;
double Y;
char Z;

    printf("Enter an integer, a real and a char: ");
    Count = scanf("%d %lf %c", &X, &Y, &Z);

    if (Count > 0)
    {
        printf("X:  %d  Y:  %g  Z:  %c
\n", X, Y, Z);
    }
}

Enter an integer, a real and a char:  125 7.5e-3  A
X:  125  Y:  0.0075  Z:  A

Function \texttt{printf()} converts items from internal representation into character strings (based on the formatting specs) and copies them into the output buffer.

\%c – put character
\%s – put character(s), stop on null byte
\%d – put decimal digit(s)
\%x – put hex digit(s)

Note: \texttt{printf()} is converting internal representation to external representation.
Example:

```c
short int N = 67;

printf( "N: %d %x %c\n", N, N, N );
```

N in RAM: 0000000001000011

Output buffer:

```
  . . \n N : 6 7 4 3 C \n .
```

Function `scanf()` extracts characters from the input buffer, converts them based on the formatting specs and stores them in memory at the pointers.

- `%c` – get character
- `%s` – get character(s), stop on whitespace
- `%d` – get decimal digit(s), stop on anything else
- `%x` – get hex digit(s), stop on anything else

Function skips leading whitespace; handles special cases gracefully (ex: formatting spec is `%d`, next character is not a decimal digit).
Example:

\[ \text{Count} = \text{scanf}( "\%d \%x \%c", \&X, \&Y, \&Z ); \]

Input buffer:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>7</th>
<th>a</th>
<th>f</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td></td>
<td>125</td>
<td></td>
<td>7af</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%d – extracts "125", converts to two’s complement
%x – extracts "7af", converts to two’s complement
%c – extracts "A" (no conversion necessary)

Buffer pointer now is positioned on "B"

Conversion: external base 10 to internal

"125" \(\Rightarrow\) 0000000001111101 (16 bits)

Calculation:

\[ 125 = 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 \]

Processing characters:

\[ ("1"-"0") \times 10^2 + ("2"-"0") \times 10^1 + ("5"-"0") \times 10^0 \]

Note: "5"-"0" = 53 - 48 = 5
Conversion: external base 16 to internal

"7af" ==> 0000011110101111 (16 bits)

Calculation:

\[ 7af = 7 \times 16^2 + a \times 16^1 + f \times 16^0 \]

Processing characters:

\[ \left(\text{“7”-“0”}\right) \times 16^2 + \left(\text{“a”-“a”+10}\right) \times 16^1 + \left(\text{“f”-“a”+10}\right) \]

Note: \( \text{“f”-“a”+10} = 102 - 97 + 10 = 15 \)

Nested calculation:

\[ 125 = 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 \]

\[ = (((((1) * 10) + 2) * 10) + 5) \]

\[ 7af = 7 \times 16^2 + a \times 16^1 + f \times 16^0 \]

\[ = (((((7) * 16) + a) * 16) + f) \]
Algorithm:

answer = 0
iterate over digits in original number
    answer = answer * base + current digit

Programming considerations:

- digits must be converted from char to int
- all values in two’s complement inside circuits

Conversion: internal to external base 10

0000000001000011 (16 bits) ==> "67"

Calculation:

67 / 10 = 6 R 7
6 / 10 = 0 R 6 (halt when quotient is 0)

Process digits from bottom to top, convert each digit to the corresponding character.

Note: 6 + “0” = “6"
Conversion: internal to external base 16

0000000001000011 (16 bits) ==> "43"

Calculation:

\[
\begin{align*}
67 / 16 &= 4 \text{ R } 3 \\
4 / 16 &= 0 \text{ R } 4 \quad \text{(halt when quotient is 0)}
\end{align*}
\]

Process digits from bottom to top, convert each digit to the corresponding character.

Note: 4 + “0” = “4”

Algorithm:

\[
\begin{align*}
\text{value} &= \text{original number} \\
\text{loop until value} &= 0 \\
\quad \text{current digit} &= \text{value} \% \text{base} \\
\quad \text{value} &= \text{value} / \text{base}
\end{align*}
\]

Programming considerations:

- digits generated in reverse order
- digits must be converted from int to char
- all values in two’s complement inside circuits
Complete examples:

~cse320/Examples/example08.pdf

~cse320/Examples/example09.pdf