Memory and Data Structures

Arrays, Stacks, Queues

(Ch 10 & 16)
Memory

• This is the “RAM” in a system
• We have seen labels and addresses point to pieces of memory storing:
  • words
  • bytes
  • strings
  • numbers
• Memory is just a collection of bits
• We could use it to represent integers
• Or as an arbitrary set of bits
Treat memory as a giant array

• Compiler or programmer decides what use to make of it.
• The element numbering starts at 0
• The element number is an address
• In “C” to allocate some memory:

```c
char m[size_of_array];
```
Storage of Data

- LC-3 architecture is “word addressable” meaning that all addresses are “word” addresses.
- This means the smallest unit of memory we can allocate is 16-bits, a word.
- Use ‘LD’ (load) and ‘ST’ (store) to access this unit (or LDR & STR).
Storage of bytes

Example

mychar
newline

...  .BLKW 1  .FILL xA

...  
LD  R1, newline
GETC
ST  R0, mychar
JSR  Sub ; R2=R1-R0
BRz  found_newline

...  
found_newline  ...
The data is placed in memory like this at start up (assuming data section starts at address 1). The “mychar” variable will change to the value of the character entered by the user once stored.
Pointers and Arrays

We've seen examples of both of these in our LC-3 programs, let's see how these work in "C"

```
Test:VAL
.stringz
.printf
```

**Pointer**

- Address of a variable in memory
- Allows us to indirectly access variables
  - in other words, we can talk about its *address* rather than its *value*

**Array**

- A list of values arranged sequentially in memory
- Example: a list of telephone numbers
- Expression `a[4]` refers to the 5th element of the array `a`
Arrays

Array implementation is very important

- Most assembly languages have only basic concept of arrays (BLKW)
- From an array, any other data structure we might want can be built
Properties of arrays:
- Each element is the same size
- Elements are stored contiguously
- First element at the smallest memory address

In assembly language we must
- Allocate correct amount of space for an array
- Map array addresses to memory addresses
LC-3 declarations of arrays within memory

To allocate a portion of memory (more than a single variable’s worth)

```
variableName .BLKW numElements
```

numElements is just that, numbering starts at 0 (as in C)
Array of Integers

Calculating the address of an array element

```c
int myarray[7]    /* C */
myarray .BLKW 7    ; LC-3
```

- If base address of “myarray” is 25

Which is base address + distance from the first element
How do you get the address of myarray?

- Use the "load effective address" instruction, "LEA"
- Keep clear the difference between an address and the contents of an address.
Addressing Byte Arrays

To get address of `myarray[4]` in LC-3, write the code...

```
LEA   R0, myarray
ADD   R1, R0, 4
```

Now, if we wanted to increment element number 5 by 1...

```
LDR   R4, R1, 0
ADD   R4, R4, 1
STR   R4, R1, 0
```
Address vs. Value

Sometimes we want to deal with the address of a memory location, rather than the value it contains.

Recall example from Chapter 6: adding a column of numbers.
- \( \text{R2} \) contains address of first location.
- Read value, add to sum, and increment \( \text{R2} \) until all numbers have been processed.

\( \text{R2} \) is a pointer -- it contains the address of data we’re interested in.
2-Dimensional Arrays

2-Dimensional arrays are more complicated in assembly

- Memory is a 1-D array
- Must map 2-D array to 1-D array
- Arrays have rows and columns
  - $r \times c$ array
  - $r =$ rows
  - $c =$ columns
2-Dimensional Arrays

$$[[r]]$$

Two sensible ways to map 2-D to 1-D

Row major form:
(rows are all together)

| 0,0 |
| 0,1 |
| 1,0 |
| 1,1 |
| 2,0 |
| 2,1 |
| 3,0 |
| 3,1 |

Column major form:
(columns are all together)

| 0,0 |
| 0,1 |
| 1,0 |
| 1,1 |
| 2,0 |
| 2,1 |
| 3,0 |
| 3,1 |
2-Dimensional Arrays

\[
Cr[j][c]
\]

How do you calculate addresses in a 2-D array?

- **Row Major:**
  \[
  \text{Address } (r_i, c_i) = \text{Base Address} + (((r_i \times \text{Number of Cols}) + c_i) \times \text{Element size})
  \]

- **Column Major:**
  \[
  \text{Address } (r_i, c_i) = \text{Base Address} + (((c_i \times \text{Number of Rows}) + r_i) \times \text{Element size})
  \]
\[ A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \]

\[ A[0][0] = ((0 \cdot 3) + 0) \cdot 1 - 0 \]

\[ A[0][2] = ((0 \cdot 3) + 2) = 2 \]

\[ A[2][1] = ((2 \cdot 3) + 1) = 7 \]
\[ L = 0.2 \]

\[ \text{short} \ [8] = 8 \]

\[ \text{inh} + 32 \ [8] = 32 \]
Summary of 2D arrays

- Row/Column major (storage order)
- Base address
- Size of elements
- Dimensions of the array

How about 3-D arrays?
Bounds Checking

- Many HLL’s have bounds checking (not C!!!)
- Assembly languages have no implied bounds checking
- Your program is in total control of memory
- With a 5 x 3 array, what does the following address?

```
array .BLKW 15
```

```
LEA R1, array
ADD R1, R1, 15
LD R0, R1, 0
```

- Bounds checking is often a good idea!!
- Most C development environments include optional bounds checking.
Stacks

A LIFO (last-in first-out) storage structure.

- The first thing you put in is the last thing you take out.
- The last thing you put in is the first thing you take out.

This means of access is what defines a stack, not the specific implementation.

Two main operations:

- **PUSH**: add an item to the stack
- **POP**: remove an item from the stack
A Physical Stack

Coin rest in the arm of an automobile

Initial State

After One Push

After Three More Pushes

After One Pop

First quarter out is the last quarter in.
A Hardware Implementation

Data items move between registers

Initial State: Empty: Yes

After One Push: Empty: No

After Three More Pushes: Empty: No

After Two Pops: Empty: No
A Software Implementation

Data items don't move in memory, just our idea about where the TOP of the stack is.

Initial State

After One Push

After Three More Pushes

After Two Pops

By convention, R6 holds the Top of Stack (TOS) pointer.
Basic Push and Pop Code

For our implementation, stack grows downward (when item added, TOS moves closer to 0)

Push

```
ADD R6, R6, #-1 ; decrement stack ptr
STR  R0, R6, #0 ; store data (R0)
```

Pop

```
LDR  R0, R6, #0 ; load data from TOS
ADD  R6, R6, #1 ; increment stack ptr
```
Pop with Underflow Detection

If we try to pop too many items off the stack, an underflow condition occurs.

- Check for underflow by checking TOS before removing data.
- Return status code in R5 (0 for success, 1 for underflow)

```
POP
LD R1, EMPTY ; EMPTY = -x4000
ADD R2, R6, R1 ; Compare stack pointer
BRz FAIL ; to x4000 to see if empty
LDR R0, R6, #0
ADD R6, R6, #1
AND R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0 ; FAIL: R5 = 1
ADD R5, R5, #1
RET
EMPTY .FILL xC000 ; 2SC rep of -x4000
```
Push with Overflow Detection

If we try to push too many items onto the stack, an overflow condition occurs. This example assumes stack has room for 5 items.

- Check for overflow by checking TOS before adding data.
- Return status code in R5 (0 for success, 1 for overflow)

```
PUSH  LD  R1, MAX          ; MAX = -x3FFB
ADD  R2, R6, R1           ; Compare stack pointer
BRZ FAIL                  ; top address to see if full
ADD  R6, R6, #-1
STR  R0, R6, #0
AND  R5, R5, #0           ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0       ; FAIL: R5 = 1
ADD  R5, R5, #1
RET
MAX .FILL xC005           ; 2SC of -x3FFB
```
Stack Example

- Printing out a positive integer, character by character
- Push LSB to MSB
- Pop MSB to LSB (LIFO)

```java
int integer = 1024

if (integer == 0) {
    push '0'
} else {
    while (integer != 0) {
        int digit = integer % base;
        char = digit + 48;
        push char onto stack
        integer = integer div base
    }
    while stack is not empty {
        pop char
        print char
    }
```
RPN

Arithmetic Using a Stack

Instead of registers, some ISA’s use a stack for source and destination operations: a zero-address machine.

- Example:
  ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

Evaluating \((A+B)\cdot(C+D)\) using a stack:

1. push A
2. push B
3. ADD
4. push C
5. push D
6. ADD
7. MULTIPLY
8. pop result

Why use a stack?
- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using LIFO data structure.
Example: OpAdd

POP two values, ADD, then PUSH result.
Example: OpAdd

OpAdd

<table>
<thead>
<tr>
<th>JSR POP</th>
<th>; Get first operand.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R5,R5,#0</td>
<td>; Check for POP success.</td>
</tr>
<tr>
<td>BRp Exit</td>
<td>; If error, bail.</td>
</tr>
<tr>
<td>ADD R1,R0,#0</td>
<td>; Make room for second.</td>
</tr>
<tr>
<td>JSR POP</td>
<td>; Get second operand.</td>
</tr>
<tr>
<td>ADD R5,R5,#0</td>
<td>; Check for POP success.</td>
</tr>
<tr>
<td>BRp Restore1</td>
<td>; If err, restore &amp; bail.</td>
</tr>
<tr>
<td>ADD R0,R0,R1</td>
<td>; Compute sum.</td>
</tr>
<tr>
<td>JSR RangeCheck</td>
<td>; Check size.</td>
</tr>
<tr>
<td>BRp Restore2</td>
<td>; If err, restore &amp; bail.</td>
</tr>
<tr>
<td>JSR PUSH</td>
<td>; Push sum onto stack.</td>
</tr>
<tr>
<td>RET</td>
<td></td>
</tr>
</tbody>
</table>

| Restore2           | ADD R6,R6,#-1       | ; Decr stack ptr (undo POP) |
| Restore1           | ADD R6,R6,#-1       | ; Decr stack ptr |
| Exit               | RET                 |                      |
Queues

A **queue** is a FIFO (First In, First Out).
- The classic analogy of a queue is a line.
  - Person gets on the end of the line (the **Tail**),
  - Waits,
  - Gets off at the front of the line (the **Head**).
- Getting into the queue is an operation called **enqueue**
- Taking something off the queue is an operation called **dequeue**.
- It takes 2 pointers to keep track of the data structure,
  - Head (let’s use R5)
  - Tail always points to empty element (R6)
Initial state:

After 1 enqueue operation:

After another enqueue operation:
After a dequeue operation:

Like stacks, when an item is removed from the data structure, it is physically still present, but correct use of the structure cannot access it.
Implementation of a queue

Storage:
- `queue .BLKW infinity` ; assume infinite for now
- `LEA R5, queue` ; head
- `LEA R6, queue` ; tail

Enqueue (item):
- `STR R0, R6, #0` ; R0 has data to store
- `ADD R6, R6, #1`

Dequeue (item):
- `JSR SUB` ; R0 = R5-R6
- `BRz queue_empty`
- `LDR R1, R5, #0` ; put data in R1
- `ADD R5, R5, #1`
Circular Queues

- To avoid infinite array, wrap around from end to beginning.
- Head == Tail means empty
- Head points to first item (for next dequeue)
- Tail point to empty location (for next enqueue)

Example of an 8 element circular queue
Circular Queues

After “enqueue’ing” one element

After “enqueue’ing” another element
Circular Queues

T = 10
T = 1

8 - 1 = 7

After "dequeue'ing" an element

Head

Tail

T = 7
T = 0
8 - 8 = 0
Circular Queues

Storage and initialization:

```
.queue
.queue_end
LEA R5, queue ; head
LEA R6, queue ; tail
.BLKW queue_size
.BLKW 1
```

Enqueue (item)

```
STR R0, R6, #0 ; data to enqueue is in R0
ADD R6, R6, #1
LEA R1, queue_end
JSR continue1
SUB ; R1 = R1 - R6
LEA R6, queue ; wrap around
```

end = end - Tail
Dequeue (item):

JSR    SUB        ; R1 = R5 – R6
BRz    queue_empty
LDR    R0, R5, #0
ADD    R5, R5, #1
[LEA    R1, queue_end
    JSR    SUB        ; R1 = R5 – R1
    BRn    continue2
    LEA    R5, queue   ; wrap around
    continue2]
Summary of data structures

- All data structures are based on the simple array.
- 2D Arrays, Stacks, Queues.
- It is all about the implementation.
- Bounds checking is important.
- If not documented can become confusing.
Floating Point Numbers
• Registers for real numbers usually contain 32 or 64 bits, allowing $2^{32}$ or $2^{64}$ numbers to be represented.
• Which reals to represent? There are an infinite number between 2 adjacent integers. (or two reals!!)
• Which bit patterns for reals selected?
• Answer: use scientific notation
### Floating Point Numbers

Consider: $A \times 10^B$, where $A$ is one digit

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td></td>
<td>$A \times 10^B$</td>
</tr>
<tr>
<td>$0$</td>
<td>any</td>
<td>$0$</td>
</tr>
<tr>
<td>$1..9$</td>
<td>$0$</td>
<td>$1..9$</td>
</tr>
<tr>
<td>$1..9$</td>
<td>$1$</td>
<td>$10..90$</td>
</tr>
<tr>
<td>$1..9$</td>
<td>$2$</td>
<td>$100..900$</td>
</tr>
<tr>
<td>$1..9$</td>
<td>$-1$</td>
<td>$0.1..0.9$</td>
</tr>
<tr>
<td>$1..9$</td>
<td>$-2$</td>
<td>$0.01..0.09$</td>
</tr>
</tbody>
</table>

How to do scientific notation in binary?

Standard: **IEEE 754 Floating-Point**
IEEE 754 Single Precision Floating Point Format

Representation:

- **S** is one bit representing the sign of the number
- **E** is an 8 bit biased integer representing the exponent
- **F** is an 23-bit unsigned integer

The true value represented is: \((-1)^S \times f \times 2^e\)

- **S** = sign bit
- **e** = **E** – bias
- **f** = \(F/2^n + 1\)
- for single precision numbers \(n=23\), bias = 127
S, E, F are all **fields** within a **representation**. Each is just a bunch of bits.

**S** is the sign bit
- \((-1)^S \rightarrow (-1)^0 = +1\) and \((-1)^1 = -1\)
- Just a sign bit for signed magnitude

**E** is the exponent field
- The E field is a biased-127 representation.
- True exponent is \((E - bias)\)
- The base (radix) is always 2 (implied).
- Some early machines used radix 4 or 16 (IBM)
F (or M) is the fractional or mantissa field.
- It is in a strange form.
- There are 23 bits for F.
- A normalized FP number always has a leading 1.
- No need to store the one, just assume it.
- This MSB is called the HIDDEN BIT.
How to convert 64.2 into IEEE SP

1. Get a binary representation for 64.2
   - Binary of left of radix point is: 1000000
   - Binary of right of radix:
     \[
     \begin{align*}
     .2 \times 2 &= 0.4 & 0 \\
     .4 \times 2 &= 0.8 & 0 \\
     .8 \times 2 &= 1.6 & 1 \\
     .6 \times 2 &= 1.2 & 1
     \end{align*}
     \]
   - Binary for .2: \underline{0011}
   - 64.2 is: \underline{1.10110010111}

2. Normalize binary form
   - Produces: \underline{1.000000 0011 E6}
3. Turn true exponent into bias-127: $6 + 127 = 133$

4. Put it together:
   - 23-bit F is: 000000 00110011001100110
   - S E F is:
   - In hex: 0x42806666

- Since floating point numbers are always stored in normal form, how do we represent 0?
  - 0x0000 0000 and 0x8000 0000 represent 0.
Other special values:
- $+5/0 = \infty$
- $\pm\infty = 0\ 11111111\ 00000...\ (0\times7f80\ 0000)$
- $-7/0 = -\infty$
- $-\infty = 1\ 11111111\ 00000...\ (0\timesff80\ 0000)$
- $0/0$ or $\pm\infty + \pm\infty = \text{NaN}$ (Not a number)
- NaN ? 11111111 ????... (S is either 0 or 1, E=0xff, and F is anything but all zeroes)
- Also de-normalized numbers (beyond scope)
IEEE Floating Point

What is the decimal value for this SP FP number 0x4228 0000?

\[
\begin{array}{c}
\text{16} \\
\text{32} \\
\text{64} \\
\hline
\text{132} \\
\text{-127} \\
\hline
\text{5}
\end{array}
\]

\[1.010100000 \rightarrow 1.01010.0 \cdots \rightarrow 42\]
What is $47.625_{10}$ in SP FP format?

$S = 0$
$E = 5$
$f = 0.1111101...$

$5 + 127 = 132$

$10000100$

$0.10000100 0.1111101 00000000000000000000000000000000$
Floating Point Format

\[ \text{24 bits} \]

What do floating-point numbers represent?

- Rational numbers with non-repeating expansions in the given base within the specified exponent range.
- They do not represent repeating rational or irrational numbers, or any number too small or too large.
IEEE Double Precision FP

- IEEE Double Precision is similar to SP
  - 52-bit M
    - 53 bits of precision with hidden bit
    - 11-bit E, excess 1023, representing $-1023 < - > 2046$
    - One sign bit

- Always use DP unless memory/file size is important unless on a microcontroller
  - SP $\sim 10^{-38} \ldots 10^{38}$
  - DP $\sim 10^{-308} \ldots 10^{308}$

- Be very careful of these ranges in numeric computation
Floating Point Arithmetic

Floating Point operations include
  • Addition
  • Subtraction
  • Multiplication
  • Division

They are complicated because...
Floating Point Addition

Decimal Review

\[ 9.997 \times 10^2 \]
\[ + 4.631 \times 10^{-1} \]

How do we do this?

1. Align decimal points
2. Add

\[
\begin{align*}
9.997 & \quad \times 10^2 \\
+ 0.004631 & \quad \times 10^2 \\
\hline
10.001631 & \quad \times 10^2
\end{align*}
\]

3. Normalize the result
   - Often already normalized
   - Otherwise move one digit

\[ 1.0001631 \times 10^3 \]

4. Possibly round result

\[ 1.000 \times 10^3 \]
Floating Point Addition

Example: $0.25 + 100$ in SP FP

First step: get into SP FP if not already

$0.25 = 0 \ 01111101 \ 00000000000000000000000000000000$
$100 = 0 \ 10000101 \ 10010000000000000000000000000000$

Or with hidden bit

$0.25 = 0 \ 01111101 \ \boxed{1} \ 00000000000000000000000000000000$
$100 = 0 \ 10000101 \ \boxed{1} \ 10010000000000000000000000000000$

Hidden Bit
Floating Point Addition

Second step: Align radix points

- Shifting F left by 1 bit, \textit{decreasing} e by 1
- Shifting F right by 1 bit, \textit{increasing} e by 1
- Shift F right so least significant bits fall off
- Which of the two numbers should we shift?

\[ \begin{array}{c}
0 & \quad 0 & \quad 0 & \quad 0 & \quad 0 & \quad 1 \\
\end{array} \]
Floating Point Addition

Second step: Align radix points cont.

Shift the .25 to increase its exponent so it matches that of 100.

$$0.25's\ e: \ 01111101 - 1111111 (127) = -2$$
$$100's\ e: \ 10000101 - 1111111 (127) = 6$$

Shift .25 by 8 then.

Easier method: Bias cancels with subtraction, so

$$\begin{array}{c}
10000101 \\
- \quad 01111101 \\
\hline
00001000
\end{array}$$
Floating Point Addition

Carefully shifting the 0.25’s fraction

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>HB</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01111101</td>
<td>1</td>
<td>00000000000000000000000000000000 (original value)</td>
</tr>
<tr>
<td>0</td>
<td>01111110</td>
<td>0</td>
<td>10000000000000000000000000000000 (shifted by 1)</td>
</tr>
<tr>
<td>0</td>
<td>01111111</td>
<td>0</td>
<td>01000000000000000000000000000000 (shifted by 2)</td>
</tr>
<tr>
<td>0</td>
<td>10000000</td>
<td>0</td>
<td>00100000000000000000000000000000 (shifted by 3)</td>
</tr>
<tr>
<td>0</td>
<td>10000001</td>
<td>0</td>
<td>00010000000000000000000000000000 (shifted by 4)</td>
</tr>
<tr>
<td>0</td>
<td>10000010</td>
<td>0</td>
<td>00001000000000000000000000000000 (shifted by 5)</td>
</tr>
<tr>
<td>0</td>
<td>10000011</td>
<td>0</td>
<td>00000100000000000000000000000000 (shifted by 6)</td>
</tr>
<tr>
<td>0</td>
<td>10000100</td>
<td>0</td>
<td>00000010000000000000000000000000 (shifted by 7)</td>
</tr>
<tr>
<td>0</td>
<td>10000101</td>
<td>0</td>
<td>00000001000000000000000000000000 (shifted by 8)</td>
</tr>
</tbody>
</table>
Floating Point Addition

Third Step: Add fractions with hidden bit

\[
\begin{align*}
0 & \quad 10000101 \quad 1 \quad 10010000000000000000000000000000 \quad (100) \\
+ & \quad 0 \quad 10000101 \quad 0 \quad 00000001000000000000000000000000 \quad (.25) \\
\hline
0 & \quad 10000101 \quad 1 \quad 1001000100000000000000000000000000
\end{align*}
\]

Fourth Step: Normalize the result

- Get a ‘1’ back in hidden bit
- Already normalized most of the time
- Remove hidden bit and finished
Floating Point Addition

Normalization example

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>HB</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>011</td>
<td>1</td>
<td>1100</td>
</tr>
<tr>
<td>+</td>
<td>011</td>
<td>1</td>
<td>1011</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>11</td>
<td>0111</td>
</tr>
</tbody>
</table>

Need to shift so that only a 1 in HB spot

| 0 | 100 | 1 | 1011 | 1 -> discarded |
Floating Point Subtraction

- Mantissa’s are sign-magnitude
- Watch out when the numbers are close

\[
\begin{align*}
1.23455 \times 10^2 \\
-1.23456 \times 10^2 \\
\end{align*}
\]

- A many-digit normalization is possible
  This is why FP addition is in many ways more difficult than FP multiplication
Steps to do subtraction

1. Align radix points
2. Perform sign-magnitude operand swap if needed
   - Compare magnitudes (with hidden bit)
   - Change sign bit if order of operands is changed.
3. Subtract
4. Normalize
5. Round
## Floating Point Subtraction

### Simple Example:

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>HB</th>
<th>F</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>011</td>
<td>1</td>
<td>1011</td>
<td>smaller</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>011</td>
<td>1</td>
<td>1101</td>
<td>bigger</td>
<td></td>
</tr>
</tbody>
</table>

switch order and make result negative

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>HB</th>
<th>F</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>011</td>
<td>1</td>
<td>1101</td>
<td>bigger</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>011</td>
<td>1</td>
<td>1011</td>
<td>smaller</td>
<td></td>
</tr>
</tbody>
</table>

| 1 | 011 | 0  | 0010  |       |       |

1 000 1 0000 switched sign, renormalized
Floating Point Multiplication

Decimal example:

\[ 3.0 \times 10^1 \times 5.0 \times 10^2 \]

How do we do this?

1. Multiply mantissas
   \[ 3.0 \times 5.0 = 15.00 \]

2. Add exponents
   \[ 1 + 2 = 3 \]

3. Combine
   \[ 15.00 \times 10^3 \]

4. Normalize if needed
   \[ 1.50 \times 10^4 \]
Floating Point Multiplication

Multiplication in binary (4-bit F)

\[
\begin{array}{c}
0.10000100 \times 1.0011000000 \\
\times 1.0011000000 \\
\hline
1.0011000000 \\
\end{array}
\]

Step 1: Multiply mantissas (put hidden bit back first!!)
Floating Point Multiplication

Second step: Add exponents, subtract extra bias.

\[
\begin{array}{c}
10000100 \\
+ 00111100 \\
\hline
11000000
\end{array}
\quad
\begin{array}{c}
11000000 \\
- 01111111 \\
\hline
01000001
\end{array} \quad (127)
\]

Third step: Renormalize, correcting exponent
\[
1 \ 01000001 \ 10.00110000
\]
Becomes
\[
1 \ 01000010 \ 1.000110000
\]

Fourth step: Drop the hidden bit
\[
1 \ 01000010 \ 000110000
\]
\[
= 0xA10C0000
\]
Floating Point Multiplication

Multiply these SP FP numbers together

\[ 0x49FC0000 \times 0x4BE00000 \]
Floating Point Division

- True division
  - Unsigned, full-precision division on mantissas
    - This is much more costly (e.g. 4x) than mult.
  - Subtract exponents
- Faster division
  - Newton’s method to find reciprocal
  - Multiply dividend by reciprocal of divisor
  - May not yield exact result without some work
  - Similar speed as multiplication
- Not covered in this class!
Floating Point Summary

- Has 3 portions, S, E, F/M
- Do conversion in parts
- Arithmetic is signed magnitude
- Subtraction could require many shifts for renormalization
- Multiplication is easier since do not have to match exponents