The Hardware/Software Interface
CSE351 Spring 2015

Lecture 4

Instructor:
Katelin Bailey

Teaching Assistants:
Announcements
Announcements

• Reminder: Lab 0 is due this evening, unless you’ve emailed me about an extension due to setup issues
  • Accounts are made: drop by the front desk in CSE to pick up your account sheet
Announcements

• Reminder: Lab 0 is due this evening, unless you’ve emailed me about an extension due to setup issues
  • Accounts are made: drop by the front desk in CSE to pick up your account sheet

• Auditors are welcome!
  • But we won’t grade your assignments.
  • Let me know if you want to be added to the Catalyst resources
Announcements

• **Reminder: Lab 0 is due this evening, unless you’ve emailed me about an extension due to setup issues**
  • Accounts are made: drop by the front desk in CSE to pick up your account sheet

• **Auditors are welcome!**
  • But we won’t grade your assignments.
  • Let me know if you want to be added to the Catalyst resources

• **The first couple weeks of this class are overwhelming!**
  • Pointers are hard! Binary is hard!
  • But both are key to this class.
  • Do the readings (both posted online and K&R)!
  • Come to office hours, section, email with questions if you have them
Today

• Brief review of topics from last lecture
• Things we didn’t get to last lecture:
  • Boolean algebra and bitwise manipulations
• Integers
  • Representation of integers: unsigned and signed
  • Integers in C
  • Sign extension
  • Arithmetic and shifting
Today

• Brief review of topics from last lecture
• Things we didn’t get to last lecture:
  • Boolean algebra and bitwise manipulations
• Integers
  • Representation of integers: unsigned and signed
  • Integers in C
  • Sign extension
  • Arithmetic and shifting

Get as far as we can with these and continue on Wednesday
Today

• **Brief review of topics from last lecture**
• **Things we didn’t get to last lecture:**
  • Boolean algebra and bitwise manipulations
• **Integers**
  • Representation of integers: unsigned and signed
  • Integers in C
  • Sign extension
  • Arithmetic and shifting
**Arrays in C (review)**

**Declaration:**
```c
int a[6];
```

**Indexing:**
- `a[0] = 0x015f;`
- `a[5] = a[0];`

**No bounds check:**
- `a[6] = 0xBAD;`
- `a[-1] = 0xBAD;`

**Pointers:**
- `int* p;`
- `p = a;`
- `p = &a[0];`
- `*p = 0xA;`

**array indexing = address arithmetic**
- `p[1] = 0xDB;`
- `*(p + 1) = 0xDB;`
- `p = p + 2;`
- `*p = a[1] + 1;`

Arrays are adjacent locations in memory storing the same type of data object.

*a* is a name for the array’s address, not a pointer to the array.

The address of `a[i]` is the address of `a[0]` plus *i* times the element size in bytes.
Null-terminated Strings (review)
Null-terminated Strings *(review)*

- For example, “Harry Potter” can be stored as a 13-byte array.
Null-terminated Strings *(review)*

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>a</td>
<td>r</td>
<td>y</td>
<td>P</td>
<td>o</td>
<td>t</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td></td>
<td></td>
<td>\0</td>
</tr>
</tbody>
</table>

Harry Potter \0
Null-terminated Strings *(review)*

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>97</td>
<td>114</td>
<td>114</td>
<td>121</td>
<td>32</td>
<td>80</td>
<td>111</td>
<td>116</td>
<td>116</td>
<td>101</td>
<td>114</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Harry y Potter \0

char s[16];

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>‘H’</td>
<td>‘a’</td>
<td>‘r’</td>
<td>‘r’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>‘y’</td>
<td>‘ ’</td>
<td>‘P’</td>
<td>‘o’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x04</td>
</tr>
<tr>
<td>‘t’</td>
<td>‘t’</td>
<td>‘e’</td>
<td>‘r’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x08</td>
</tr>
<tr>
<td>‘\0’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x0C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x10</td>
</tr>
</tbody>
</table>
Null-terminated Strings *review*

- For example, “Harry Potter” can be stored as a 13-byte array:

```
72 97 114 114 121 32 80 111 116 116 101 114 0
H a r r y P o t t e r \0
```

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: `string[12] = '\0';`

```c
char s[16];
```

```
| 'H' | 'a' | 'r' | 'r' | 0x00 |
| 'y' | ' ' | 'P' | 'o' | 0x04 |
| 't' | 't' | 'e' | 'r' | 0x08 |
| '\0' | 0x0C |
```
Null-terminated Strings *(review)*

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>a</td>
<td>r</td>
<td>r</td>
<td>y</td>
<td>P</td>
<td>o</td>
<td>t</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\0</td>
</tr>
</tbody>
</table>

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: `string[12] = '\0';`

- How do we compute the string length?
Null-terminated Strings (review)

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>a</td>
<td>r</td>
<td>y</td>
<td>P</td>
<td>o</td>
<td>t</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td>\0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: string[12] = '\0';

- How do we compute the string length?

Don’t worry about wasted space yet! We’ll hit it in a few weeks.
Today

• Brief review of topics from last lecture

• Things we didn’t get to last lecture:
  • Boolean algebra and bitwise manipulations

• Integers
  • Representation of integers: unsigned and signed
  • Integers in C
  • Sign extension
  • Arithmetic and shifting
Examining Data Representations

• Code to print byte representation of data
  • Any data type can be treated as a byte array by casting it to char.
  • C has unchecked casts. << DANGER >>

printf directives:

%p Print pointer
\t Tab
%x Print value as hex
\n New line
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char.
  - C has unchecked casts. << DANGER >>

```c
typedef char byte;  // size of char == 1 byte

void show_bytes(byte* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

printf directives:
- `%p` Print pointer
- `\t` Tab
- `%x` Print value as hex
- `\n` New line
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char.
  - C has unchecked casts. << DANGER >>

```c
typedef char byte;  // size of char == 1 byte

void show_bytes(byte* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}

void show_int (int x) {
    show_bytes((byte *) &x, sizeof(int));
}

printf directives:
%p Print pointer
\t Tab
%x Print value as hex
\n New line
```
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by **casting** it to `char`.
  - C has **unchecked** casts. **<< DANGER >>**

```c
typedef char byte; // size of char == 1 byte

void show_bytes(byte* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p \t 0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

**Example of how casting to byte allows pointer arithmetic by byte instead of int.**

```c
void show_int (int x) {
    show_bytes( (byte *) &x, sizeof(int));
}
```

**printf directives:**
- `%p` Print pointer
- `	` Tab
- `%x` Print value as hex
- `\n` New line
show_bytes Execution Example
show_bytes Execution Example

```c
int a = 12345;  // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a);  // show_bytes((pointer) &a, sizeof(int));
```
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((pointer) &a, sizeof(int));

Result (Linux):

int a = 12345;
0x11fffffcb8 0x39
0x11fffffcb9 0x30
0x11fffffcb9 0x00
0x11fffffcb9 0x00
Boolean Algebra

Poll: How many of you have taken logic?
Boolean Algebra

Poll: How many of you have taken logic?

• Developed by George Boole in 19th Century
Boolean Algebra

Poll: How many of you have taken logic?

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
Boolean Algebra

Poll: How many of you have taken logic?

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
Boolean Algebra

Poll: How many of you have taken logic?

• Developed by George Boole in 19th Century
  • Algebraic representation of logic
    • Encode “True” as 1 and “False” as 0
  • AND: A&B = 1 when both A is 1 and B is 1
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: $A \& B = 1$ when both $A$ is 1 and $B$ is 1
  - OR: $A | B = 1$ when either $A$ is 1 or $B$ is 1

Poll: How many of you have taken logic?
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1
  - OR: \( A | B = 1 \) when either \( A \) is 1 or \( B \) is 1
  - XOR: \( A ^ B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both

Poll: How many of you have taken logic?
Boolean Algebra

Poll: How many of you have taken logic?

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1
  - OR: \( A \mid B = 1 \) when either \( A \) is 1 or \( B \) is 1
  - XOR: \( A \wedge B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both
  - NOT: \( \sim A = 1 \) when \( A \) is 0 and vice-versa
Boolean Algebra

Poll: How many of you have taken logic?

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1
  - OR: \( A \mid B = 1 \) when either \( A \) is 1 or \( B \) is 1
  - XOR: \( A \^ B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both
  - NOT: \( \sim A = 1 \) when \( A \) is 0 and vice-versa
  - DeMorgan’s Law: \( \sim (A \mid B) = \sim A \& \sim B \)
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1
  - OR: \( A | B = 1 \) when either \( A \) is 1 or \( B \) is 1
  - XOR: \( A ^ B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both
  - NOT: \( \sim A = 1 \) when \( A \) is 0 and vice-versa
  - DeMorgan’s Law: \( \sim (A \mid B) = \sim A \& \sim B \)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: How many of you have taken logic?
Boolean Algebra

Poll: How many of you have taken logic?

• Developed by George Boole in 19th Century
  • Algebraic representation of logic
    • Encode “True” as 1 and “False” as 0
  • AND: $A \& B = 1$ when both $A$ is 1 and $B$ is 1
  • OR: $A \mid B = 1$ when either $A$ is 1 or $B$ is 1
  • XOR: $A \wedge B = 1$ when either $A$ is 1 or $B$ is 1, but not both
  • NOT: $\sim A = 1$ when $A$ is 0 and vice-versa
  • DeMorgan’s Law: $\sim (A \mid B) = \sim A \& \sim B$

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\mid</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean Algebra

*Developed by George Boole in 19th Century*
- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0
- **AND:** $A \& B = 1$ when both $A$ is 1 and $B$ is 1
- **OR:** $A \mid B = 1$ when either $A$ is 1 or $B$ is 1
- **XOR:** $A \^ B = 1$ when either $A$ is 1 or $B$ is 1, but not both
- **NOT:** $\sim A = 1$ when $A$ is 0 and vice-versa
- **DeMorgan’s Law:** $\sim (A \mid B) = \sim A \& \sim B$

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

|  |  &   |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
|  |   0   |   0   |   0   |   0   |   0   |   1   |   1   |   1   |
| 0 |   0   |   0   |   0   |   0   |   1   |   1   |   1   |   1   |   0   |
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: \(A \& B = 1\) when both \(A\) is 1 and \(B\) is 1
  - OR: \(A | B = 1\) when either \(A\) is 1 or \(B\) is 1
  - XOR: \(A ^ B = 1\) when either \(A\) is 1 or \(B\) is 1, but not both
  - NOT: \(\sim A = 1\) when \(A\) is 0 and vice-versa
  - DeMorgan’s Law: \(\sim (A | B) = \sim A \& \sim B\)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
\& \\
01010101 \\
\hline
01000001
\end{array}
\]
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
\& 01010101 \\
\hline
01000001
\end{array} \quad \quad \begin{array}{c}
01101001 \\
\| 01010101 \\
\hline
01111101
\end{array}
\]
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

<table>
<thead>
<tr>
<th>01101001</th>
<th>01101001</th>
<th>01101001</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; 01010101</td>
<td></td>
<td>01010101</td>
</tr>
<tr>
<td>01000001</td>
<td></td>
<td>01111101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & \quad \mid 01010101 & \quad \wedge 01010101 & \quad \sim 01010101 \\
01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
\end{align*}
\]
General Boolean Algebras

- **Operate on bit vectors**
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
\& 01010101 \\
01000001
\end{array}
\quad
\begin{array}{c}
01101001 \\
\mid 01010101 \\
01111101
\end{array}
\quad
\begin{array}{c}
01101001 \\
^ 01010101 \\
00111100
\end{array}
\quad
\begin{array}{c}
01101001 \\
\sim 01010101 \\
10101010
\end{array}
\]

- **All of the properties of Boolean algebra apply**
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{ccc}
01101001 & \text{&} & 01101001 \\
01010101 & | & 01010101 \\
\hline
01000001 & | & 01111101 \\
\hline
01111101 & \text{^} & 01111101 \\
01010101 & \sim & 01010101 \\
\hline
00111100 & \sim & 10101010 \\
\hline
\end{array}
\]

- All of the properties of Boolean algebra apply

\[
\begin{array}{c}
01010101 \\
\text{^} \\
01010101 \\
\hline
00000000
\end{array}
\]
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

  \[
  \begin{array}{ccc}
  01101001 & 01101001 & 01101001 \\
  \& 01010101 & | 01010101 & ^ 01010101 \\
  01000001 & 01111101 & 00111100 & \sim 01010101 \\
  \end{array}
  \]

- All of the properties of Boolean algebra apply

  \[
  \begin{array}{c}
  01010101 \\
  \uparrow 01010101 \\
  00000000
  \end{array}
  \]

- How does this relate to set operations?
Representing & Manipulating Sets
Representing & Manipulating Sets

- Representation
Representing & Manipulating Sets

- Representation
  - A \( w \)-bit vector represents subsets of \( \{0, \ldots, w-1\} \)
Representing & Manipulating Sets

- Representation
  - A \( w \)-bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_j = 1 \) iff \( j \in A \)
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j = 1$ iff $j \in A$

  01101001 \hspace{1cm} \{ 0, 3, 5, 6 \}
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of \{0, ..., $w-1$\}
  - $a_j = 1$ iff $j \in A$

01101001 \quad \{ 0, 3, 5, 6 \}

76543210
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j = 1$ iff $j \in A$

  \[
  \begin{align*}
  01101001 & \rightarrow \{ 0, 3, 5, 6 \} \\
  76543210 & 
  \end{align*}
  \]
Representing & Manipulating Sets

- **Representation**
  - A \( w \)-bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_j = 1 \) iff \( j \in A \)

  \[
  \begin{array}{c|c}
  \text{Binary} & \text{Set} \\
  \hline
  01101001 & \{0, 3, 5, 6\} \\
  76543210 & \\
  01010101 & \{0, 2, 4, 6\} \\
  \end{array}
  \]
Representing & Manipulating Sets

• Representation
  • A $w$-bit vector represents subsets of $\{0, \ldots, w-1\}$
  • $a_j = 1$ iff $j \in A$

01101001  \quad \{ 0, 3, 5, 6 \} 
76543210

01010101  \quad \{ 0, 2, 4, 6 \} 
76543210
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_j = 1 \) iff \( j \in A \)
    
    \[
    \begin{align*}
    01101001 & \quad \{ 0, 3, 5, 6 \} \\
    76543210 & \\
    01010101 & \quad \{ 0, 2, 4, 6 \} \\
    76543210 & 
    \end{align*}
    \]

- **Operations**
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of \{0, ..., $w-1$\}
  - $a_j = 1$ iff $j \in A$
    
    | 01101001 | \{ 0, 3, 5, 6 \} |
    | 76543210 |

    | 01010101 | \{ 0, 2, 4, 6 \} |
    | 76543210 |

- **Operations**
  - & Intersection
    
    | 01000001 | \{ 0, 6 \} |
Representing & Manipulating Sets

• Representation
  • A w-bit vector represents subsets of \{0, \ldots, w-1\}
  • \(a_j = 1\) iff \(j \in A\)
    
    \[
    \begin{align*}
    01101001 \quad & \{0, 3, 5, 6\} \\
    76543210 \\
    
    01010101 \quad & \{0, 2, 4, 6\} \\
    76543210
    \end{align*}
    \]

• Operations
  • \& Intersection
    
    \[
    \begin{align*}
    01000001 \quad & \{0, 6\} \\
    \end{align*}
    \]
  • | Union
    
    \[
    \begin{align*}
    01111101 \quad & \{0, 2, 3, 4, 5, 6\} \\
    \end{align*}
    \]
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of \{0, ..., w−1\}
  - $a_j = 1$ iff $j \in A$

  \[
  \begin{align*}
  01101001 & \quad \{ 0, 3, 5, 6 \} \\
  76543210 & \\
  \hline
  01010101 & \quad \{ 0, 2, 4, 6 \} \\
  76543210 & 
  \end{align*}
  \]

- **Operations**
  - & Intersection \quad 01000001 \quad \{ 0, 6 \}
  - | Union \quad 01111101 \quad \{ 0, 2, 3, 4, 5, 6 \}
  - ^ Symmetric difference \quad 00111100 \quad \{ 2, 3, 4, 5 \}
Representing & Manipulating Sets

- **Representation**
  - A \( w \)-bit vector represents subsets of \{0, \ldots, w-1\}
  - \(a_j = 1\) iff \(j \in A\)

\[
\begin{align*}
01101001 & \quad \{0, 3, 5, 6\} \\
76543210 & \\
01010101 & \quad \{0, 2, 4, 6\} \\
76543210 & \\
\end{align*}
\]

- **Operations**
  - & Intersection \(01000001\) \(\{0, 6\}\)
  - | Union \(01111101\) \(\{0, 2, 3, 4, 5, 6\}\)
  - ^ Symmetric difference \(00111100\) \(\{2, 3, 4, 5\}\)
  - ~ Complement \(10101010\) \(\{1, 3, 5, 7\}\)
Bit-Level Operations in C
Bit-Level Operations in C

• &   |   ^   ~
Bit-Level Operations in C

- &  |  ^  ~
- Apply to any “integral” data type
Bit-Level Operations in C

- & | ^ ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
Bit-Level Operations in C

- \& \ | \ ^ \ ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
Bit-Level Operations in C

- &   |   ^   ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Examples (char data type)
Bit-Level Operations in C

- &   |   ^   ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
- Examples (char data type)
  - \(~0x41\) --> 0xBE
Bit-Level Operations in C

- & | ^ ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
- Examples (char data type)
  - ~0x41 --> 0xBE
  - ~01000001₂ --> 10111110₂
Bit-Level Operations in C

- &     |      ^    ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Examples (char data type)
  - ~0x41 --> 0xBE
  - ~01000001₂ --> 10111110₂
  - ~0x00 --> 0xFF
Bit-Level Operations in C

- & | ^ ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Examples (char data type)
  - \~0x41 \rightarrow \text{0xBE}
  - \~01000001_2 \rightarrow 10111110_2
  - \~0x00 \rightarrow \text{0xFF}
  - \~00000000_2 \rightarrow 11111111_2
Bit-Level Operations in C

- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Examples (char data type)
  - \( \sim 0x41 \rightarrow 0xBE \)
    \( \sim 01000001_2 \rightarrow 10111110_2 \)
  - \( \sim 0x00 \rightarrow 0xFF \)
    \( \sim 00000000_2 \rightarrow 11111111_2 \)
  - \( 0x69 \ & \ 0x55 \rightarrow 0x41 \)
Bit-Level Operations in C

- &    |   ^   ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Examples (char data type)
  - \(~0x41\) --> \(0xBE\)
    \(~01000001_2\) --> \(10111110_2\)
  - \(~0x00\) --> \(0xFF\)
    \(~00000000_2\) --> \(11111111_2\)
  - \(0x69 \& 0x55\) --> \(0x41\)
    \(01101001_2\) \& \(01010101_2\) --> \(01000001_2\)
Bit-Level Operations in C

- & | ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
- Examples (char data type)
  - ~0x41 --> 0xBE
    - ~01000001 \(\rightarrow\) 10111110
  - ~0x00 --> 0xFF
    - ~00000000 \(\rightarrow\) 11111111
  - 0x69 & 0x55 --> 0x41
    - 01101001 \& 01010101 \(\rightarrow\) 01000001
  - 0x69 | 0x55 --> 0x7D
Bit-Level Operations in C

- &   |   ^   ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors

Examples (char data type)
- \(~0x41\) --> \(0xBE\)
  \(~01000001_2\) --> \(10111110_2\)
- \(~0x00\) --> \(0xFF\)
  \(~00000000_2\) --> \(11111111_2\)
- \(0x69 \& 0x55\) --> \(0x41\)
  \(01101001_2 \& 01010101_2\) --> \(01000001_2\)
- \(0x69 \mid 0x55\) --> \(0x7D\)
  \(01101001_2 \mid 01010101_2\) --> \(01111101_2\)
Bit-Level Operations in C

- & | ^ ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- **Examples (char data type)**
- ~0x41 --> 0xBE
  ~01000001₂ --> 10111110₂
- ~0x00 --> 0xFF
  ~00000000₂ --> 11111111₂
- 0x69 & 0x55 --> 0x41
  01101001₂ & 01010101₂ --> 01000001₂
- 0x69 | 0x55 --> 0x7D
  01101001₂ | 01010101₂ --> 01111101₂
- **Many bit-twiddling puzzles in Lab 1**
Contrast: Logic Operations in C
Contrast: Logic Operations in C

- Contrast to logical operators
Contrast: Logic Operations in C

- Contrast to logical operators

  - && | | !
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&     ||     !
  - 0 is “False”
Contrast: Logic Operations in C

- Contrast to logical operators
  - && | !
  - 0 is “False”
  - Anything nonzero is “True”
Contrast: Logic Operations in C

- Contrast to logical operators
  - \&\&     |     ||     |
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
Contrast: Logic Operations in C

- Contrast to logical operators
  - && | |
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation
Contrast: Logic Operations in C

• Contrast to logical operators
  • && || !
    • 0 is “False”
    • Anything nonzero is “True”
    • Always return 0 or 1
    • Early termination a.k.a. short-circuit evaluation

• Examples (char data type)
Contrast: Logic Operations in C

- Contrast to logical operators
  - && || !
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation

- Examples (char data type)
  - !0x41 --> 0x00
Contrast: Logic Operations in C

- Contrast to logical operators
  - && | ||
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation

- Examples (char data type)
  - !0x41 --> 0x00
  - !0x00 --> 0x01
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&  |  ||  !
    - 0 is “False”
    - Anything nonzero is “True”
    - Always return 0 or 1
    - Early termination  a.k.a. short-circuit evaluation

- Examples (char data type)
  - !0x41  -->  0x00
  - !0x00  -->  0x01
  - !!0x41  -->  0x01
Contrast: Logic Operations in C

- Contrast to logical operators
  - && | || !
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation

- Examples (char data type)
  - !0x41 --> 0x00
  - !0x00 --> 0x01
  - !!0x41 --> 0x01
  - 0x69 && 0x55 --> 0x01
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&   ||    !
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination  a.k.a. short-circuit evaluation

- Examples (char data type)
  - !0x41  -->  0x00
  - !0x00  -->  0x01
  - !!0x41  -->  0x01
  - 0x69 && 0x55  -->  0x01
  - 0x69 || 0x55  -->  0x01
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&  |  ||  !
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation

- Examples (char data type)
  - !0x41 --> 0x00
  - !0x00 --> 0x01
  - !!0x41 --> 0x01
  - 0x69 && 0x55 --> 0x01
  - 0x69 || 0x55 --> 0x01
  - p && *p++ (avoids null pointer access, null pointer = x00000000)
Today

• Brief review of topics from last lecture
• Things we didn’t get to last lecture:
  • Boolean algebra and bitwise manipulations
• Integers
  • Representation of integers: unsigned and signed
  • Integers in C
  • Sign extension
  • Arithmetic and shifting
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:

get_mpg:
  pushq   %rbp
  movq    %rsp, %rbp
  ...
  popq    %rbp
  ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

OS:

Windows 8

Mac

Memory, data, &
addressing

Integers & floats

Machine code & C

x86 assembly

Procedures & stacks

Arrays & structs

Memory & caches

Processes

Virtual memory

Memory allocation

Java vs. C
But before we get to integers....

- Encode a standard deck of playing cards.
- 52 cards in 4 suits
  - How do we encode suits, face cards?
- What operations do we want to make easy to implement?
  - Which is the higher value card?
  - Are they the same suit?
Two possible representations

- 52 cards – 52 bits with bit corresponding to card set to 1

  low-order 52 bits of 64-bit word

  - “One-hot” encoding
  - Drawbacks:
    - Hard to compare values and suits
    - Large number of bits required
Two possible representations

• **52 cards – 52 bits with bit corresponding to card set to 1**

  “One-hot” encoding

  Drawbacks:
  • Hard to compare values and suits
  • Large number of bits required

• **4 bits for suit, 13 bits for card value – 17 bits with two set to 1**

  Pair of one-hot encoded values

  Easier to compare suits and values
  • Still an excessive number of bits

• **Can we do better?**
Two better representations

- Binary encoding of all 52 cards – only 6 bits needed
  - Fits in one byte
  - Smaller than one-hot encodings.
  - How can we make value and suit comparisons easier?

low-order 6 bits of a byte
Two better representations

• Binary encoding of all 52 cards – only 6 bits needed

  • Fits in one byte
  • Smaller than one-hot encodings.
  • How can we make value and suit comparisons easier?

• Binary encoding of suit (2 bits) and value (4 bits) separately

  • Also fits in one byte, and easy to do comparisons
Compare Card Suits

cchar hand[5];       // represents a 5-card hand
cchar card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( sameSuitP(card1, card2) ) { ... }
#define SUIT_MASK 0x30

int sameSuitP(char card1, char card2) {
    return (! (card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}

SUIT_MASK = 0x30 = 00110000

char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];

...  
if ( sameSuitP(card1, card2) ) { ... }
#define SUIT_MASK 0x30

int sameSuitP(char card1, char card2) {
    return (! (card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}

returns int  

SUIT_MASK = 0x30 = 00110000

mask: a bit vector that, when bitwise ANDed with another bit vector υ, turns all 
but the bits of interest in υ to 0

card1, card2; // two cards to compare

card1 = hand[0];
card2 = hand[1];

if ( sameSuitP(card1, card2) ) { ... }

char hand[5]; // represents a 5-card hand
Compare Card Values

```c
char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
card1 = hand[0];   // two cards to compare
card2 = hand[1];
...
if ( greaterValue(card1, card2) ) { ... }
```
Define VALUE_MASK = 0x0F

int greaterValue(char card1, char card2) {
    return ((unsigned int)(card1 & VALUE_MASK) >
        (unsigned int)(card2 & VALUE_MASK));
}

VALUE_MASK = 0x0F = 0 0 0 0 1 1 1 1

char hand[5]; // represents a 5-card hand
char card1, card2; // two cards to compare

... if (greaterValue(card1, card2)) { ... }
#define VALUE_MASK 0x0F

```c
int greaterValue(char card1, char card2) {
    return ((unsigned int)(card1 & VALUE_MASK) >
            (unsigned int)(card2 & VALUE_MASK));
}
```

- **mask**: a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all but the bits of interest in \( v \) to 0.

VALUE_MASK = 0x0F = \[0000011111\]

- suit
- value

char hand[5]; // represents a 5-card hand
char card1, card2; // two cards to compare

card1 = hand[0];
card2 = hand[1];
...

if (greaterValue(card1, card2) ) { ... }
Today

• Brief review of topics from last lecture
• Things we didn’t get to last lecture:
  • Boolean algebra and bitwise manipulations
• Integers
  • **Representation of integers: unsigned and signed**
    • Integers in C
    • Sign extension
    • Arithmetic and shifting
Encoding Integers
The hardware (and C) supports two flavors of integers:

- **unsigned** – only the non-negatives
- **signed** – both negatives and non-negatives
The hardware (and C) supports two flavors of integers:

- unsigned – only the non-negatives
- signed – both negatives and non-negatives

There are only $2^W$ distinct bit patterns of $W$ bits, so...

- Can not represent all the integers
- Unsigned values: 0 ... $2^W-1$
- Signed values: $-2^{W-1}$ ... $2^{W-1}-1$
Encoding Integers

- The hardware (and C) supports two flavors of integers:
  - unsigned – only the non-negatives
  - signed – both negatives and non-negatives

- There are only $2^W$ distinct bit patterns of $W$ bits, so...
  - Can not represent all the integers
  - \textit{Unsigned values:} $0$ ... $2^W - 1$
  - \textit{Signed values:} $-2^{W-1}$ ... $2^{W-1} - 1$

- Reminder: terminology for binary representations
The hardware (and C) supports two flavors of integers:

- *unsigned* – only the non-negatives
- *signed* – both negatives and non-negatives

There are only $2^W$ distinct bit patterns of $W$ bits, so...

- Can not represent all the integers
- *Unsigned values*: $0 \ldots 2^W - 1$
- *Signed values*: $-2^{W-1} \ldots 2^{W-1} - 1$

Reminder: terminology for binary representations

- “Most-significant” or “high-order” bit(s)
- “Least-significant” or “low-order” bit(s)

```
0110010110101001
```
Unsigned Integers
Unsigned Integers

• Unsigned values are just what you expect
  • $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0$
  • Useful formula: $1 + 2 + 4 + 8 + \ldots + 2^{N-1} = 2^N - 1$
Unsigned Integers

- Unsigned values are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0$
  - Useful formula: $1 + 2 + 4 + 8 + \ldots + 2^{N-1} = 2^N - 1$

- Add and subtract using the normal “carry” and “borrow” rules, just in binary.
Unsigned Integers

• Unsigned values are just what you expect
  • $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0$
  • Useful formula: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$

• Add and subtract using the normal “carry” and “borrow” rules, just in binary.

\[
\begin{array}{c}
00111111 \\
+00001000 \\
\hline
01000111
\end{array}
\]

63
+ 8
71
Unsigned Integers

- Unsigned values are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0$
  - Useful formula: $1 + 2 + 4 + 8 + \ldots + 2^{N-1} = 2^N - 1$

- Add and subtract using the normal “carry” and “borrow” rules, just in binary.

- Why would you care about unsigned integers?
Unsigned Integers

- Unsigned values are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0$
  - Useful formula: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$

- Add and subtract using the normal “carry” and “borrow” rules, just in binary.

- Why would you care about unsigned integers?
- How would you make signed integers?
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
Signed Integers: Sign-and-Magnitude

• Let's do the natural thing for the positives
  • They correspond to the unsigned integers of the same value
  • Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127
- But, we need to let about half of them be negative
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): \(0x00 = 0, 0x01 = 1, \ldots, 0x7F = 127\)
- But, we need to let about half of them be negative
  - Use the **high-order bit** to indicate negative: call it the **“sign bit”**
Signed Integers: Sign-and-Magnitude

• Let's do the natural thing for the positives
  • They correspond to the unsigned integers of the same value
    • Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127

• But, we need to let about half of them be negative
  • Use the high-order bit to indicate negative: call it the “sign bit”
    • Call this a “sign-and-magnitude” representation
Let's do the natural thing for the positives
  • They correspond to the unsigned integers of the same value
    • Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127

But, we need to let about half of them be negative
  • Use the high-order bit to indicate negative: call it the “sign bit”
    • Call this a “sign-and-magnitude” representation
  • Examples (8 bits):
Signed Integers: Sign-and-Magnitude

• Let's do the natural thing for the positives
  • They correspond to the unsigned integers of the same value
    • Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127

• But, we need to let about half of them be negative
  • Use the **high-order bit** to indicate negative: call it the **“sign bit”**
    • Call this a “sign-and-magnitude” representation
  • Examples (8 bits):
    • 0x00 = 00000000₂ is non-negative, because the sign bit is 0
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127
- But, we need to let about half of them be negative
  - Use the high-order bit to indicate negative: call it the "sign bit"
    - Call this a "sign-and-magnitude" representation
  - Examples (8 bits):
    - 0x00 = 00000000₂ is non-negative, because the sign bit is 0
    - 0x7F = 01111111₂ is non-negative
Signed Integers: Sign-and-Magnitude

• Let's do the natural thing for the positives
  • They correspond to the unsigned integers of the same value
    • Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127

• But, we need to let about half of them be negative
  • Use the high-order bit to indicate negative: call it the “sign bit”
    • Call this a “sign-and-magnitude” representation
  • Examples (8 bits):
    • 0x00 = 00000000₂ is non-negative, because the sign bit is 0
    • 0x7F = 01111111₂ is non-negative
    • 0x85 = 10000101₂ is negative
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127
- But, we need to let about half of them be negative
  - Use the high-order bit to indicate negative: call it the “sign bit”
    - Call this a “sign-and-magnitude” representation
  - Examples (8 bits):
    - 0x00 = 00000000₂ is non-negative, because the sign bit is 0
    - 0x7F = 01111111₂ is non-negative
    - 0x85 = 10000101₂ is negative
    - 0x80 = 10000000₂ is negative...
Signed Integers: Sign-and-Magnitude

• How should we represent -1 in binary?
  • \(10000001_2\)
    Use the MSB for + or -, and the other bits to give magnitude.

Most Significant Bit

![Diagram showing signed integers and their binary representations with the MSB indicated as the sign bit.]
Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - \(10000001_2\)
  
  Use the MSB for + or -, and the other bits to give magnitude.
  
  (Unfortunate side effect: there are **two representations of 0**!)
Sign-and-Magnitude Negatives

- How should we represent \(-1\) in binary?
  - \(10000001\)_2
    Use the MSB for + or -, and the other bits to give magnitude.
    (Unfortunate side effect: there are two representations of 0!)
- Another problem: arithmetic is cumbersome.
  - Example:
    \(4 - 3 \neq 4 + (-3)\)
Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - \(10000001_2\)
    Use the MSB for + or -, and the other bits to give magnitude.
    (Unfortunate side effect: there are two representations of 0!)
  - Another problem: **arithmetic is cumbersome.**

- Example:
  \(4 - 3 \neq 4 + (-3)\)

\[ \begin{array}{c}
+1000 \\
+1011 \\
= 1111
\end{array} \]
How should we represent -1 in binary?

- $10000001_2$
  
  Use the MSB for + or -, and the other bits to give magnitude.
  
  (Unfortunate side effect: there are two representations of 0!)

- Another problem: **arithmetic is cumbersome.**

- Example:
  
  $4 - 3 \neq 4 + (-3)$

  \[
  \begin{array}{c}
  \text{0100} \\
  +1011 \\
  \hline
  \text{1111}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{0100} + 1011 = 1111
  \end{array}
  \]

How do we solve these problems?