Overview

- Computer representation of “things”
- Unsigned Numbers
- Signed Numbers: search for a good representation
- Shortcuts
- In Conclusion

Review: The Programmer’s Model of a Microcomputer

Instruction Set:
- ldr r0, [r2, #0]
- add r2, r3, r4

Memory:
- 80000004 ldr r0, [r2, #0]
- 80000008 add r2, r3, r4
- 8000000B 23456
- 80000010 AEF0

Memory mapped I/O
- 80000100 input
- 80000108 output

Registers:
- r0 - r3, pc

Addressing Modes:
- ldr r12, [r1,#0]
- mov r1, r3

Programmer’s Model

How to access data in registers and memory? i.e. how to determine and specify the data address in registers and memory

Review: Compilation

- How to turn notation programmers prefer into notation computer understands?
- Program to translate C statements into Assembly Language instructions; called a compiler
- Example: compile by hand this C code:
  - a = b + c;
  - d = a - e;
- Easy:
  - add r1, r2, r3
  - sub r4, r1, r6
- Big Idea: compiler translates notation from 1 level of abstraction to lower level
What do computers do?

° Computers manipulate representations of things!

° What can you represent with N bits?
  • $2^N$ things!

° Which things?
  • Numbers! Characters! Pixels! Dollars! Position! Instructions!
  • Depends on what operations you do on them

Decimal Numbers: Base 10

° Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

° Example:
  
  $3271 = (3 \times 10^3) + (2 \times 10^2) + (7 \times 10^1) + (1 \times 10^0)$

Numbers: positional notation

° Number Base $B$ => $B$ symbols per digit:
  • Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  Base 2 (Binary): 0, 1

° Number representation:
  • $d_{31}d_{30} \ldots d_2d_1d_0$ is a 32 digit number
  • value = $d_{31} \times B^{31} + d_{30} \times B^{30} + \ldots + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$

° Binary: 0, 1
  • $1011010 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 \times 1 = 64 + 16 + 8 + 2 = 90$
  • Notice that 7 digit binary number turns into a 2 digit decimal number
  • A base that converts to binary easily?

Hexadecimal Numbers: Base 16 (#1/2)

° Digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

° Normal digits have expected values

° In addition:
  • A $\rightarrow$ 10
  • B $\rightarrow$ 11
  • C $\rightarrow$ 12
  • D $\rightarrow$ 13
  • E $\rightarrow$ 14
  • F $\rightarrow$ 15
Hexadecimal Numbers: Base 16 (#2/2)

- Example (convert hex to decimal):
  \[ B28F0DD = (B \times 16^6) + (2 \times 16^5) + (8 \times 16^4) + (F \times 16^3) + (0 \times 16^2) + (D \times 16^1) + (D \times 16^0) \]
  \[ = (11 \times 16^6) + (2 \times 16^5) + (8 \times 16^4) + (15 \times 16^3) + (0 \times 16^2) + (13 \times 16^1) + (13 \times 16^0) \]
  \[ = 187232477 \text{ decimal} \]

- Notice that a 7 digit hex number turns out to be a 9 digit decimal number

Decimal vs. Hexadecimal vs. Binary

- Examples:
  00 0
  01 1
  02 1001
  03 11
  04 100
  05 111
  06 110
  07 111
  08 1000
  09 1001
  10 11010
  11 11011
  12 1110
  13 1111

- 1010 1100 0101 (binary) = ? (hex)
- 10111 (binary) = 0001 0111 (binary) = ? (hex)
- 3F9 (hex) = ? (binary)

Hex to Binary Conversion

- HEX is a more compact representation of Binary!
- Each hex digit represents 16 decimal values.
- Four binary digits represent 16 decimal values.
- Therefore, each hex digit can replace four binary digits.

- Example:
  0011 1011 1001 1010 1100 1010 0000 0000two
  \[ \text{C uses notation 0x3b9aca00} \]

Which Base Should We Use?

- Decimal: Great for humans; most arithmetic is done with these.
- Binary: This is what computers use, so get used to them. Become familiar with how to do basic arithmetic with them (+,-,*,/).
- Hex: Terrible for arithmetic; but if we are looking at long strings of binary numbers, it’s much easier to convert them to hex in order to look at four bits at a time.
How Do We Tell the Difference?

- In general, append a subscript at the end of a number stating the base:
  - $10_{10}$ is in decimal
  - $10_2$ is binary ($= 2_{10}$)
  - $10_{16}$ is hex ($= 16_{10}$)

- When dealing with ARM computer:
  - Hex numbers are preceded with “&” or “0x”
    - $&10 == 0x10 == 10_{16} == 16_{10}$
    - Note: Lab software environment only supports “0x”
  - Binary numbers are preceded with “0b”
  - Octal numbers are preceded with “0”
  - Everything else by default is Decimal

Inside the Computer

- To a computer, numbers are always in binary; all that matters is how they are printed out: binary, decimal, hex, etc.

- As a result, it doesn’t matter what base a number in C is in...
  - $32_{10} == 0x20 == 100000_2$

- … only the value of the number matters.

What to do with representations of numbers?

- Just what we do with numbers!
  - Add them
  - Subtract them
  - Multiply them
  - Divide them
  - Compare them

- Example: $10 + 7 = 17$
  - $100001$

Addition Table

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>
Addition Table (binary)

```
+ | 0 | 1 |
---|---|---|
0 | 0 | 1 |
1 | 1 | 10
```

Addition Table (Hex)

```
+ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 |
2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 |
3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 |
4 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 |
5 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 |
6 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 |
7 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
8 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
9 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
A | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
B | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A |
C | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B |
D | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C |
E | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D |
F | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | 1E |
```

Quiz # 1 Result

<table>
<thead>
<tr>
<th>Title</th>
<th>N</th>
<th>% Correct Of:</th>
<th>Discrimination</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Whole Group</td>
<td>Upper 25%</td>
<td>Lower 25%</td>
</tr>
<tr>
<td>Arrays 1</td>
<td>131</td>
<td>70</td>
<td>91</td>
<td>48</td>
</tr>
<tr>
<td>Data types</td>
<td>131</td>
<td>67</td>
<td>91</td>
<td>27</td>
</tr>
<tr>
<td>Pointer to functions</td>
<td>131</td>
<td>45</td>
<td>69</td>
<td>18</td>
</tr>
<tr>
<td>Pointers</td>
<td>131</td>
<td>84</td>
<td>96</td>
<td>60</td>
</tr>
<tr>
<td>Pointers and addresses</td>
<td>131</td>
<td>83</td>
<td>100</td>
<td>54</td>
</tr>
<tr>
<td>Pointer initialisation</td>
<td>131</td>
<td>74</td>
<td>93</td>
<td>42</td>
</tr>
</tbody>
</table>

Overall Mean: 71.1 %

Bicycle Computer (Embedded)

- **P. Brain**
  - wireless heart monitor strap
  - record 5 measures: speed, time, current distance, elevation and heart rate
  - Every 10 to 60 sec.
  - 8KB data => 33 hours
  - Stores information so can be uploaded through a serial port into PC to be analyzed

http://www.specialized.com
Limits of Computer Numbers

° Bits can represent anything!

° Characters?
  • 26 letter => 5 bits
  • upper/lower case + punctuation => 7 bits (in 8) (ASCII)
  • rest of the world’s languages => 16 bits (unicode)

° Logical values?
  • 0 -> False, 1 => True

° colors?

° locations / addresses? commands?
  ° but N bits => only $2^N$ things

What if too big?

° Binary bit patterns above are simply representatives of numbers

° Numbers really have an infinite number of digits
  - with almost all being zero except for a few of the rightmost digits: e.g: 0000000 … 000098 == 98
  - Just don’t normally show leading zeros

° Computers have fixed number of digits
  - In general, adding two n-bit numbers can produce an (n+1)-bit result.
  - Since computers use fixed, 32-bit integers, this is a problem.
  - If result of add (or any other arithmetic operation), cannot be represented by these rightmost hardware bits, overflow is said to have occurred

Overflow Example

° Example (using 4-bit numbers):
  +15  1111
  +3   0011
  +18  10010

  • But we don’t have room for 5-bit solution, so the solution would be 0010, which is +2, which is wrong.

How avoid overflow, allow it sometimes?

° Some languages detect overflow (Ada), some don’t (C and JAVA)

° ARM has N, Z, C and V flags to keep track of overflow
  • Refer Book!
  • Will cover details later
Comparison

° How do you tell if \( X > Y \)?
° See if \( X - Y > 0 \)
→ We need representation for both +ve and −ve numbers

How to Represent Negative Numbers?

° So far, unsigned numbers
° Obvious solution: define leftmost bit to be sign!
  • \( 0 \Rightarrow + \), \( 1 \Rightarrow - \)
  • Rest of bits can be numerical value of number
° Representation called **sign and magnitude**
° ARM uses 32-bit integers. \(+1_{\text{ten}}\) would be:
  \[0000 \, 0000 \, 0000 \, 0000 \, 0000 \, 0000 \, 0000 \, 0001\]
° And \(-1_{\text{ten}}\) in sign and magnitude would be:
  \[1000 \, 0000 \, 0000 \, 0000 \, 0000 \, 0000 \, 0000 \, 0001\]

Shortcomings of sign and magnitude?

° Arithmetic circuit more complicated
  • Special steps depending whether signs are the same or not
° Also, Two zeros
  • \(0x00000000 = +0_{\text{ten}}\)
  • \(0x80000000 = -0_{\text{ten}}\)
  • What would it mean for programming?
° Sign and magnitude abandoned because another solution was better

Another try: complement the bits

° Example: \(7_{10} = 00111_2\) \(-7_{10} = 11000_2\)
° Called **one's Complement**
° Note: positive numbers have leading 0s, negative numbers have leadings 1s.

\[0000 \, 0001 \, ... \, 0111\]
\[1000 \, ... \, 1111 \, 0 \, 1111\]
° What is \(-0000\)?
° How many positive numbers in \(N\) bits?
° How many negative ones?
Shortcomings of ones complement?

- Arithmetic not too hard

- Still two zeros
  - 0x00000000 = +0<sub>ten</sub>
  - 0xFFFFFFFF = -0<sub>ten</sub>
  - What would it mean for programming?

- One's complement eventually abandoned because another solution was better

Search for Negative Number Representation

- Obvious solution didn't work, find another

- What is result for unsigned numbers if tried to subtract large number from a small one?
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
  - 3 – 7 = –4
    - 000011
    - 111100
    - 111111
  - 3 – 4 = –1
    - 000000
    - 111111

- With no obvious better alternative, pick representation that made the hardware simple:
  - leading 0s => positive, leading 1s => negative
  - 000000...xxx is >=0, 111111...xxx is < 0

- This representation called two's complement.

Two’s Complement

- 0000 ... 0000 0000 0000 0000<sub>two</sub> = 0<sub>ten</sub>
- 0000 ... 0000 0000 0000 0001<sub>two</sub> = 1<sub>ten</sub>
- 0000 ... 0000 0000 0000 0010<sub>two</sub> = 2<sub>ten</sub>
- 0000 ... 0000 0000 0000 0011<sub>two</sub> = 3<sub>ten</sub>
- 0000 ... 0000 0000 0000 1000<sub>two</sub> = –2<sub>ten</sub>
- 0000 ... 0000 0000 0000 1001<sub>two</sub> = –3<sub>ten</sub>

- One zero, 31st bit ➔ >=0 or <0, called sign bit
  - but one negative with no positive –2,147,483,648<sub>ten</sub>
Two’s Complement Formula, Example

Recognizing role of sign bit, can represent positive and negative numbers in terms of the bit value times a power of 2:

- \( d_{31} \times -2^{31} + d_{30} \times 2^{30} + ... + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0 \)

Example

\[ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{\text{two}} \]

\[ = 1 \times -2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + ... + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \]

\[ = -2^{31} + 2^{30} + 2^{29} + ... + 2^2 + 0 + 0 \]

\[ = -2,147,483,648_{\text{ten}} + 2,147,483,644_{\text{ten}} \]

\[ = -4_{\text{ten}} \]

Two’s complement shortcut: Negation

- Invert every 0 to 1 and every 1 to 0, then add 1 to the result

- Sum of number and its inverted representation must be \( (111...111_{\text{two}} = -1_{\text{ten}}) \)

- Let \( x' \) mean the inverted representation of \( x \)

- Then \( x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x \)

Example: -4 to +4 to -4x:

\[ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{\text{two}} \]

\[ x' : 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{\text{two}} \]

\[ +1 : 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_{\text{two}} \]

\[ ()' : 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{\text{two}} \]

\[ +1 : 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{\text{two}} \]

And in Conclusion...

- We represent “things” in computers as particular bit patterns: N bits \( \rightarrow 2^N \)
  - numbers, characters, ... (data)

- Decimal for human calculations, binary to understand computers, hex to understand binary

- 2’s complement universal in computing: cannot avoid, so learn

- Computer operations on the representation correspond to real operations on the real thing

- Overflow: numbers infinite but computers finite, so errors occur