Queen Mary
University of London

ECS404U: Computer Systems & Networks

2020/21 – Semester 1

Prof. Edmund Robinson, Dr. Arman Khouzani

Lab 3: *Basic Binary Operations & Representing Negative Numbers in Two's Complement*

October 05/07, 2020

Deadline for submitting your proof of work: ***Next week's Thursday, at 10:00 AM***

Student Name:

Yujeong JUNG

Student ID:

200699145

Brief Feedback to student (commendations, areas for improvement):

Overview In the previous lab, we practised how to present unsigned integers in binary (unsigned meaning not including negatives, and integer meaning whole numbers, not fractions). This lab contains 4 parts: 1. First, we learn about how to do operations on these (unsigned) binary integers, namely addition and multiplication and bitwise logical operations. 2. The second part is about presenting negative numbers in binary using 2's complement system. 3. The third part is about hexadecimal. 4. The fourth part is about real-life examples.

Note: If you have not yet completed the week 2 lab, then this should be the priority. Remember that demonstrators are available during lab sessions to help you (so please raise hand for help if you need it). Recall that the deadline for submitting the proof of work for last week is Thursday of this week, at 10:00 AM.

1 Basic Binary Operations

Learning Objective:

- carry out the basic operations of long “addition” and “multiplication” of unsigned binary integers;
- carry out simple “bitwise” logical operations (”and”, ”or”, ”not”);

As always in this module, acceptable answers must include the detailed working.

1.1 Basic Binary Arithmetic

1. Carry out the following long addition sums in binary.

   Example:

$$\begin{array}{rl} 1100\ \ 0011 & \\ 1001\ \ 0110 & + \\ \hline 1\ \ 0000\ \ 1100 & \text{carries} \\ \hline 1\ \ 0101\ \ 1001 & \text{solution} \end{array}$$

   (a)

$$\begin{array}{rl} 1001\ \ 1101 & \\ 1001\ \ 0111 & + \\ \hline \boxed{1\ \ 0000\ 1010} & \text{carries} \\ \hline \boxed{1\ 0011\ 1100} & \text{solution} \end{array}$$

   (b)

$$\begin{array}{rl} 1100\ \ 1110 & \\ 0101\ \ 0101 & + \\ \hline \boxed{1\ 1001\ 1011} & \text{carries} \\ \hline \boxed{1\ 0010\ 0100} & \text{solution} \end{array}$$

2. Carry out the following long multiplication sums in binary.

   Example:

$$\begin{array}{rl} 1001\ \ 1001 & \\ 1001 & \times \\ \hline 1001\ \ 1001 & \\ 100\ \ 1100\ \ 1000 & \\ \hline 1\ \ 0011\ \ 0000 & \text{carries} \\ \hline 101\ \ 0110\ \ 0001 & \text{solution} \\ \hline \end{array}$$

(a)

$$\begin{array}{rl}
1101\ 1001 & \\
101 & \times \\
\hline
\boxed{1101\ 1001} & \\
\boxed{110110\ 01} & \\
\hline
\boxed{11\ 1011\ 1101} & \text{carries} \\
\hline
\boxed{100\ 0011\ 1101} & \text{solution}
\end{array}$$

(b) This is harder. In some cases you need to put a carry for a column two ahead.

$$\begin{array}{rl}
1110\ 1011 & \\
1011 & \times \\
\hline
\boxed{1110\ 1011} & \\
\boxed{11101\ 011} & \\
\boxed{1110101\ 1} & \\
\hline
\boxed{1\ 0011\ 1101} & \text{carries (one ahead)} \\
\boxed{11\ 0100\ 0111} & \text{carries (two ahead)} \\
\hline
\boxed{1010\ 0001\ 1001} & \text{solution} \\
\hline
\end{array}$$

3. Compute the bitwise ‘not’, which we represent by the symbol ‘¬’ (or sometimes by the symbol ∼) on the following bit sequence.

Nice point 1: Unlike operators like ‘`and`’ and ‘`or`’, the ‘`not`’ operator only takes only one input argument, i.e., performs an operation on only one “operand”. Incidentally, such an operator is called “unary”.

Nice point 2: The bitwise ‘`not`’ operator is sometimes also described as the binary 1’s complement (to be contrasted with the 2’s complement, which we will see in the next section!)

Example:
$$\begin{array}{lr}
\neg & 0110 \\
\hline
 & 1001 \\
\hline
\end{array}$$

(a)

$$\begin{array}{lr}
\neg & 0001\ 1100 \\
\hline
 & \boxed{1110\ 0011} \\
\hline
\end{array}$$

(b)

$$\begin{array}{lr}
\neg & 1100\ 1001 \\
\hline
 & \boxed{0011\ 0110} \\
\hline
\end{array}$$

4. Compute the bitwise ‘and’, designated by the symbol‘ &’, and bitwise ‘or’ (denoted by the symbol ‘|’) for the following pairs of bit sequences.

   Example: first bit sequence: 1110, second bit sequence: 0110:

```
1110          1110
0110   &      0110   |
----          ----
0110          1110
----          ----
```

   (a) 1010 1100 and 1001 0101

```
   1010 1100              1010 1100
   1001 0101   &          1001 0101   |
   ---------              ---------
   1000 0100              1011 1101
   ---------              ---------
```

   (b) 1000 0111 and 1010 1010

```
   1000 0111              1000 0111
   1010 1010   &          1010 1010   |
   ---------              ---------
   1000 0010              1010 1111
   ---------              ---------
```

2 Representing Negative Numbers: *Two's Complement*

Learning Objective: Familiarise yourself with the representation of positive and negative numbers in two's complement. We will make use of the simple 8-bit format. By the end of the section:

- you should be able to convert numbers between decimal and two's complement;
- you should be able to recognise when a representation is positive or negative; and
- you should have validated the claim that the unsigned addition algorithm can be used for signed addition by testing it in a few cases.

5. Convert the following from 8-bit two's complement to decimal.

Example: 1001 1101

digits	1	0	0	1	1	1	0	1
powers	-128	64	32	16	8	4	2	1
worth	-128			16	8	4		1

So 1001 1101 represents:
$-128 + 16 + 8 + 4 + 1$
$= -99$

(a) 1110 0111

digits	1	1	1	0	0	1	1	1
powers	-128	64	32	16	8	4	2	1
worth	-128	64	32	0	0	4	2	1

So 1110 0111 represents:
-128+64+32+4+2
$=$ -25

(b) 1111 0011

digits	1	1	1	1	0	0	1	1
powers	-128	64	32	16	8	4	2	1
worth	-128	64	32	16	0	0	2	1

So 1111 0011 represents:
-128+64+32+16+
$=$ -130

(c) 0001 0110

digits	0	0	0	1	0	1	1	0
powers	-128	64	32	16	8	4	2	1
worth	0	0	0	16	0	4	2	0

So 0001 0110 represents:

16+4+2

$=$ 22

6. Convert the following from decimal to 8-bit two's complement.

Example: -47

-47 is negative, so it is represented as $-47 = -128 + (128 - 47) = -128 + 81$.

powers	-128	64	32	16	8	4	2	1
diff	81	17		1				0
digits	1	1	0	1	0	0	0	1

-47 is represented by: 1101 0001

(a) -38

-38 is negative, so it is represented as $-38 = -128 + (128 - 38) = -128 +$ 90.

powers	-128	64	32	16	8	4	2	1
diff	90							
digits	1	1	0	1	1	0	1	0

-38 is represented by: 1101 1010

(b) 77

77 is positive, so its representation is as unsigned.

powers	-128	64	32	16	8	4	2	1
diff								
digits	0	1	0	0	1	1	0	1

77 is represented by: 0100 1101

(c) -2

-2 is negative, so it is represented by $-128 +$ 126

powers	-128	64	32	16	8	4	2	1
diff								
digits	1	1	1	1	1	1	1	0

-2 is represented by: 1111 1110

7. (a) The following bit patterns represent numbers in 8-bit two's complement. State whether the numbers are positive or negative:

i.	1101 0000	Negative
ii.	0101 0000	Positive
iii.	0010 1111	Positive
iv.	1111 0000	Negative

(b) Explain how you can tell that these are positive or negative without working out the value of the number.

In 8-bit word, the sign bit determine the number is positive or negative. If the sign bit is 0, this means the number is positive. However, if the sign bit is 1, the number is negative.

8. Validate addition in two's complement by checking cases.

(a) $(-47) + 22 = -25$

i. -47 is represented in 8-bit two's complement as:

1101 0001

ii. 22 is represented in two's complement as:

0001 0110

iii. Carry out the binary long addition of these two representations:

```
  1101 0001
  0001 0110  +
  1100 0110  carries
  1110 0111
```

iv. The rightmost 8 bits of this are:

-25

v. Check that this is the two's complement representation of -25: $(-47) + 22 = -25$.

(b) $(-25) + (-13) = -38$

i. -25 is represented in two's complement as:

1101 0111

ii. -13 is represented in two's complement as:

1111 0011

iii. Carry out the binary long addition of these two representations:

1110 0111	
1111 0011	+
1 0000 0100	carries
1 1101 1010	

iv. The rightmost 8 bits of this are:

-38

v. Check that this is the two's complement representation of -38: $(-25) + (-13) = -38$.

3 Hexadecimal

Learning Objective: convert between hex and standard binary representations of bit sequences.

9. Convert the following hexadecimal sequences into binary: each hexadecimal digit represents four bits.

Example: 4af5

4	a	f	5
0100	1010	1111	0101

4af5 represents: 0100 1010 1111 0101.

i. ab01

a	b	0	1
1010	1011	0000	0001

ab01 represents: 1010 1011 0000

ii. 9c2e

9	c	2	e
1001	1100	0010	1110

9c2e represents: 1001 1100 0010

iii. `da48`

d	a	4	8
1101	1010	0100	1000

`da48` represents: 1101 1010 0100

10. Convert the following bit sequences into hexadecimal.

Example: `1100 0110 0000 1111`

1100	0110	0000	1111
c	6	0	f

So `1100 0110 0000 1111` in binary is `c60f` in hex.

i. `0111 1001 1101 0010`

0111	1001	1101	0010
7	9	d	2

So `0111 1001 1101 0010` in binary is 79d2 in hex.

ii. `1010 1011 1100 1110`

1010	1011	1100	1110
a	b	c	e

So `1010 1011 1100 1110` in binary is abce in hex.

iii. `1101 1110 1010 1101`

1101	1110	1010	1101
d	e	a	d

So `1101 1110 1010 1101` in binary is dead in hex.

4 Real life examples

Learning objective: to give some real-life examples of these notations.

11. This question illustrates how MAC addresses correspond to bit sequences.

A machine's MAC address consists six two-digit pairs, each digit is hex, and the pairs are separated by dividers. The MAC address is therefore a $6 * 2 * 4 = 48$ bit sequence. Give the bit sequences corresponding to the following MAC addresses. Translate each digit or letter into four bits as before. Ignore punctuation such as colons or dashes:

i. 32:00:2e:7b:00:00

0011 0010 : 0000 0000 : 0010 1110 : 0111 1011 : 0000 0000 : 0000 0000

ii. BF-18-0E-77-16-7F

1011 1111 - 0001 1000 - 0000 1110 - 0111 0111 - 0001 0110 - 0111 1111

12. (Optional) This question illustrates how the computer stores integers.

    i. Download the file `printint.c` from QMPlus (under the same folder as for this week's lab manual) to a suitable directory. Here is the code:

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc,char* *argv) {
// declare an integer array of length 1
int a[1];
// convert string given as argument to integer, and store in a[0]
a[0] = atoi(argv[1]);
// write integer to standard output as binary
fwrite(a,sizeof(int),1,stdout);
}
```

    Explanation: the first two lines tell C to use some standard libraries. The rest of the file is the actual program. In C that is a function called `main`. The arguments to the program are an array of strings, called "`argv`". We are going to run the program by typing: `./printint -47`. So `argv[0]` is the string `./printint` and `argv[1]` is the string "-47". This is a sequence of three characters, and is not the same as an integer. So we have to convert it to the integer -47, and we do this using a standard function `atoi` (alphabetic to integer). We store this as the first and only element of the array `a`. Finally, we print `a` to the standard output as a binary stream using the function `fwrite`.

    ii. Compile the file to produce an executable: `gcc -o printint printint.c`

    iii. Now run the program by typing `./printint -47`. You should see something like:

```
edmundr$ ./printint -47
????edmundr$
```

    The issue is that the terminal is not equipped to display the binary stream being generated. So we run it through a utility, that will convert each binary `0` to a character '0', and each binary `1` to a character '1'. This can be done in Unix by using a pipe, in which you pipe the output of one program to the input of another. Pipes are given by a single `|`. The program we want is `xxd -b`. Type `./printint -47 | xxd -b`. You should see something like:

```
edmundr$ ./printint -47 | xxd -b
00000000: 11010001 11111111 11111111 11111111                          ....
edmundr$
```

    This takes some explanation:
    `00000000:` is an index
    `11010001 11111111 11111111 11111111`: the stream contains four bytes given in this order `....` these four bytes do not correspond to printable characters.
    Now we saw that in eight-bit two's complement -47 is `1101 0001`. To get 32-bit two's complement we pad positive numbers with 0's and negative numbers with 1's. So -47 in 32-bit two's complement is `11111111 11111111 11111111 11010001`. But computers use Little Endian byte order (see later in lectures), which means they reverse the order of the bytes. So we get: `11010001 11111111 11111111 11111111`. Verify that this is what you got.

Give the results printed for:

iv. -39

v. -74

End of questions