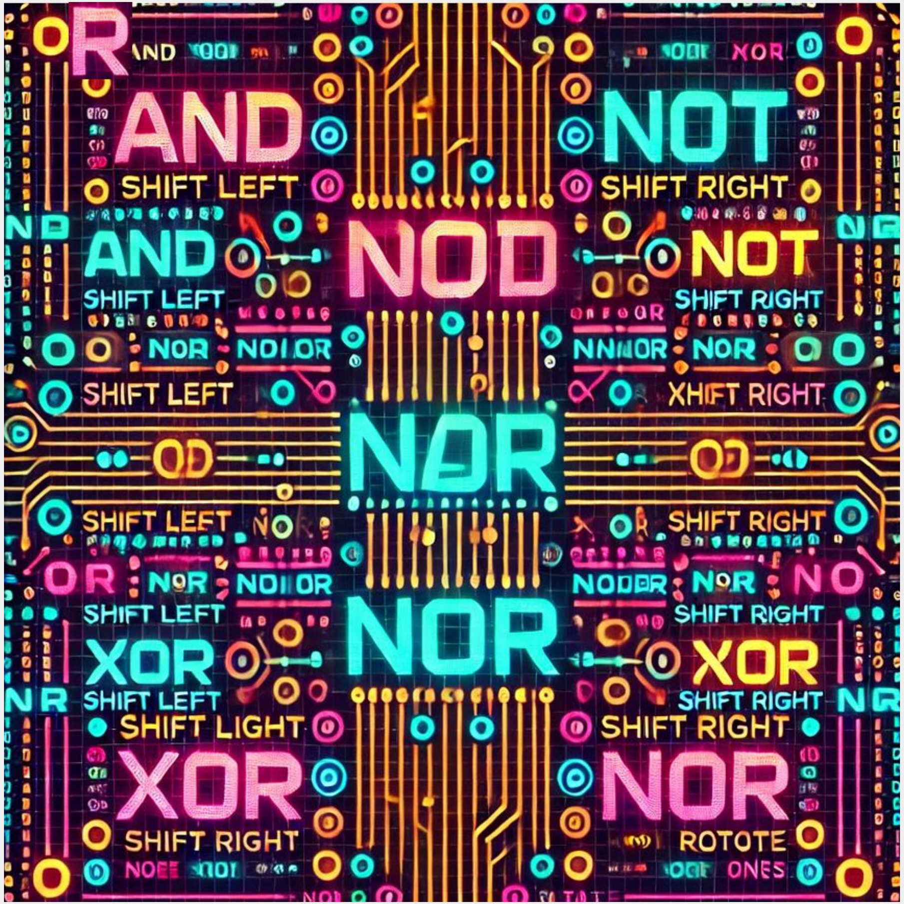
COMP1521 24T2 Lec07

Bitwise Operations

2024

Hammond Pearce

Basically reformatted Abiram's slides



Recap Exercise

Question 1: Convert $3AF_{16}$ to binary?

Question 2: Convert 10101101_2 to hexadecimal?

Question 3: Convert 673_8 to binary?

Question 4: Convert 1000_{10} to binary?

Question 5: Convert 1111_2 to hexadecimal, decimal, and octal?

Question 6: What's the difference in C if a constant value leads with "0x" versus "0b"? Does it change the program?

Quick revision on integer representation

- All data on a computer is represented in binary (base-2)
- Each **bi**nary digi**t** (or bit) can either be a **0** or **1**
- Computers use bytes (groups of 8 bits) as their fundamental units of storage

Quick revision on integer representation

- Information = data + context
  - For example, take the following byte of data:

`01001001`

    - In a numeric context*: this represents **73**
    - In the context of ASCII: this represents ‘**I**’

What about a group of 4 bytes?

- Could be an integer
- Could be an array of 4 characters

* interpreting it as an unsigned or signed (2’s complement) value

Bitwise operations

provide us ways to manipulating the individual bits of a value.

- CPUs provide instructions which implement bitwise operations.
  - MIPS provides 13 bit manipulation instructions
- C provides 6 bitwise operators
  - `&  bitwise AND`
  - `|  bitwise OR`
  - `^  bitwise XOR (eXclusive OR)`
  - `~  bitwise NOT`
  - `<< left shift`
  - `>> right shift`

Bitwise AND (&)

- takes two values (eg. **a & b**) and performs a logical AND between pairs of corresponding bits
    - resulting bits are set to 1 if **both** the original bits in that column are 1

Example:

	128	64	32	16	8	4	2	1
	0	0	1	0	0	1	1	1
&	1	1	1	0	0	0	1	1
	0	0	1	0	0	0	1	1

&	0	1
0	0	0
1	0	1

Used for eg. checking if a particular bit is set (that is, set to 1)

Checking if a number is odd

The obvious way to check if a number is odd in C:

```c
int is_odd(int n) {
    return n % 2 == 1;
}
```

Checking if a number is odd

However, an odd value must have a 1 bit in the 1s place:

128	64	32	16	8	4	2	1
0	0	1	0	0	1	1	1

We can use bitwise AND to check if the last bit is set .

Checking if a number is odd

```
int is_odd(int n) {
    return n & 1;
}
```

If the value is **ODD** (eg 39):

	128	64	32	16	8	4	2	1
	0	0	1	0	0	1	1	1
&	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	1

If the value is **EVEN** (eg 38):

	128	64	32	16	8	4	2	1
	0	0	1	0	0	1	1	0
&	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0

Bitwise OR (|)

- takes two values (eg. **a | b**) and performs a logical OR between pairs of corresponding bits
  - resulting bits are set to 1 if **at least** one of the original bits are 1

Example:

	0	0	1	0	0	1	1	1
|	1	1	1	0	0	0	1	1
	1	1	1	0	0	1	1	1

|	0	1
0	0	1
1	1	1

Used for eg. setting a particular bit

Bitwise XOR (^)

- takes two values (eg. **a ^ b**) and performs an eXclusive OR between pairs of corresponding bits
  - resulting bits is set to 1 if **exactly** one of the original bits are 1

Example:

	0	0	1	0	0	1	1	1
^	1	1	1	0	0	0	1	0
	1	1	0	0	0	1	0	1

^	0	1
0	0	1
1	1	0

Used in eg. cryptography, forcing a bit to flip

Demo: xor.c

MIPS - Bit manipulation instructions

assembly	meaning	bit pattern
and r_d, r_s, r_t	$r_d = r_s \& r_t$	000000ssssstttttddddd00000100100
or r_d, r_s, r_t	$r_d = r_s \mid r_t$	000000ssssstttttddddd00000100101
xor r_d, r_s, r_t	$r_d = r_s \wedge r_t$	000000ssssstttttddddd00000100110
nor r_d, r_s, r_t	$r_d = \sim(r_s \mid r_t)$	000000ssssstttttddddd00000100111
andi r_t, r_s, I	$r_t = r_s \& \mathrm{I}$	001100ssssstttttIIIIIIIIIIIIIIII
ori r_t, r_s, I	$r_t = r_s \mid \mathrm{I}$	001101ssssstttttIIIIIIIIIIIIIIII
xori r_t, r_s, I	$r_t = r_s \wedge \mathrm{I}$	001110ssssstttttIIIIIIIIIIIIIIII
not r_d, r_s	$r_d = \sim r_s$	pseudo-instruction

Demo: odd_even.s

Left shift (<<)

- takes a value and a small positive integer *x* (eg. `a << x`)
- shifts each bit *x* positions to the left
  - any bits that fall off the left vanish
  - new 0 bits are inserted on the right
  - result contains the same number of bits as the input

Example:

```
1 1 1 0 0 0 1 1 << 2
---------------
1 0 0 0 1 1 0 0
```

Implications of left shift

- We moved each bit to the left
- What does this mean mathematically?

Implications of left shift

- We moved each bit to the left
- What does this mean mathematically?
- What would happen if we “left shifted” in decimal?
- E.g. we have the value 123, let us “left shift” by “1”...

Implications of left shift

- We moved each bit to the left
- What does this mean mathematically?
- What would happen if we “left shifted” in decimal?
- E.g. we have the value 123, let us “left shift” by “1”...
- It becomes “1230” - multiplied by 10!

Implications of left shift

- We moved each bit to the left
- What does this mean mathematically?
- What would happen if we “left shifted” in decimal?
- E.g. we have the value 123, let us “left shift” by “1”...
- It becomes “1230” - multiplied by 10!

- So what happens if we left shift in binary? (demo: left_shift.c)

Right shift (>>)

- takes a value and a small positive integer *x* (eg. `a >> x`)
- shifts each bit *x* positions to the right
  - any bits that fall off the right vanish
  - new 0 bits are inserted on the left*
  - result contains the same number of bits as the input

Example:

```
1 1 1 0 0 0 1 1  >> 2
---------------
0 0 1 1 1 0 0 0
```

* for unsigned values

Implications of right shift

- We moved each bit to the right
- What does this mean mathematically?
- What would happen if we “right shifted” in decimal?
- E.g. we have the value 123, let us “right shift” by “1”...

Implications of right shift

- We moved each bit to the right
- What does this mean mathematically?
- What would happen if we “right shifted” in decimal?
- E.g. we have the value 123, let us “right shift” by “1”...
- It becomes “12” - (integer) divided by 10!

- So what happens if we right shift in binary? (demo:right_shift.c)

Issues with shifting (>>)

- Shifts involving negative values may not be portable, and can vary across different implementations
- Common source of bugs in COMP1521 (and elsewhere)
- Always use unsigned values/variables when shifting to be safe/portable

Issues with shifting (>>)

```
// int16_t is a signed type (-32768..32767)
// below operations are undefined for a signed type
int16_t i;

i = -1;
i = i >> 1; // undefined -  shift of a negative value
printf("%d\n", i);

i = -1;
i = i << 1; // undefined -  shift of a negative value
printf("%d\n", i);

i = 32767;
i = i << 1; // undefined -  left shift produces a negative value

uint64_t j;
j = 1 << 33; // undefined - constant 1 is an int
j = ((uint64_t)1) << 33; // ok
j = 1lu << 33; // also ok
```

MIPS - Shift instructions

assembly	meaning	bit pattern
sllv r_d, r_t, r_s	$r_d = r_t \ll r_s$	000000sssssttttddddd00000000100
srlv r_d, r_t, r_s	$r_d = r_t \gg r_s$	000000sssssttttddddd00000000110
srav r_d, r_t, r_s	$r_d = r_t \gg r_s$	000000sssssttttddddd00000000111
sll r_d, r_t, I	$r_d = r_t \ll$ I	00000000000tttttdddddIIIII000000
srl r_d, r_t, I	$r_d = r_t \gg$ I	00000000000tttttdddddIIIII000010
sra r_d, r_t, I	$r_d = r_t \gg$ I	00000000000tttttdddddIIIII000011

- **`srl`** and **`srlv`** shift zeroes into most-significant bit
  - This matches shift in C of unsigned values
- **`sra`** and **`srav`** propagate most-significant bit
  - This ensures that shifting a negative number divides by 2

Demo: bitwise.c

```
$ gcc bitwise.c print_bits.c -o bitwise
$ ./bitwise
Enter a: 23032
Enter b: 12345
Enter c: 3
     a = 0101100111111000 = 0x59f8 = 23032
     b = 0011000000111001 = 0x3039 = 12345
    ~a = 1010011000000111 = 0xa607 = 42503
 a & b = 0001000000111000 = 0x1038 = 4152
 a | b = 0111100111111001 = 0x79f9 = 31225
 a ^ b = 0110100111000001 = 0x69c1 = 27073
a >> c = 0000101100111111 = 0x0b3f = 2879
a << c = 1100111111000000 = 0xcfc0 = 53184
```

Demo: shift_as_multiply.c

```
$ dcc shift_as_multiply.c print_bits.c -o shift_as_multiply
$ ./shift_as_multiply 4
2 to the power of 4 is 16

In binary it is: 00000000000000000000000000010000
$ ./shift_as_multiply 20
2 to the power of 20 is 1048576

In binary it is: 00000000000100000000000000000000
$ ./shift_as_multiply 31
2 to the power of 31 is 2147483648

In binary it is: 10000000000000000000000000000000
```

Exercise 1

Given the following declarations:

```
// a signed 8-bit value
   uint8_t x = 0x55;
   uint8_t y = 0xAA;
```

What is the value of each of these expressions?

```
uint8_t a = x & y;
uint8_t b = x ^ y;
uint8_t c = x << 1;
uint8_t d = y << 2;
```

```
uint8_t e = x >> 1;
uint8_t f = y >> 2;
uint8_t g = x | y;
```

Demo: set_low_bits.c

```
$ dcc set_low_bits.c print_bits.c -o n_ones
$ ./set_low_bits 3

The bottom 3 bits of 7 are ones:
00000000000000000000000000000111
$ ./set_low_bits 19

The bottom 19 bits of 524287 are ones:
00000000000001111111111111111111
$ ./set_low_bits 29

The bottom 29 bits of 536870911 are ones:
00011111111111111111111111111111
```

Demo: set_bit_range.c

```
$ dcc set_bit_range.c print_bits.c -o set_bit_range
$ ./set_bit_range 0 7

Bits 0 to 7 of 255 are ones:
00000000000000000000000011111111
$ ./set_bit_range 8 15

Bits 8 to 15 of 65280 are ones:
00000000000000001111111100000000
$ ./set_bit_range 8 23

Bits 8 to 23 of 16776960 are ones:
00000000111111111111111100000000
$ ./set_bit_range 1 30

Bits 1 to 30 of 2147483646 are ones:
01111111111111111111111111111110
```

Demo: extract_bit_range.c

```
$ dcc extract_bit_range.c print_bits.c -o extract_bit_range
$ ./extract_bit_range 4 7 42

Value 42 in binary is:
00000000000000000000000000101010

Bits 4 to 7 of 42 are:
0010
$ ./extract_bit_range 10 20 123456789

Value 123456789 in binary is:
00000111010110111100110100010101

Bits 10 to 20 of 123456789 are:
11011110011
```

Exercise 2

Given the following declarations:

```
// a signed 8-bit value
    uint8_t x = 0x55;
    uint8_t y = 0xAA;
```

What is the value of each of these expressions?

```
uint8_t h = x && y;
uint8_t i = ~(x | y);
uint8_t j = !(x | y);
uint8_t k = x | (1 << 3);
uint8_t l = x | ~(1 << 3);
```

Demo: pokemon.c

```
#define FIRE_TYPE       0x0001
#define FIGHTING_TYPE   0x0002
#define WATER_TYPE      0x0004
#define FLYING_TYPE     0x0008
#define POISON_TYPE     0x0010
#define ELECTRIC_TYPE   0x0020
#define GROUND_TYPE     0x0040
#define PSYCHIC_TYPE    0x0080
#define ROCK_TYPE       0x0100
#define ICE_TYPE        0x0200
#define BUG_TYPE        0x0400
#define DRAGON_TYPE     0x0800
#define GHOST_TYPE      0x1000
#define DARK_TYPE       0x2000
#define STEEL_TYPE      0x4000
#define FAIRY_TYPE      0x8000
```

Demo: pokemon.c

```
$ dcc pokemon.c print_bits.c -o pokemon
$ ./pokemon
0000010000000000 BUG_TYPE
0000000000010000 POISON_TYPE
1000000000000000 FAIRY_TYPE
1000010000010000 our_pokemon type (1)

Poisonous
1001010000000000 our_pokemon type (2)

Scary
```

Demo: bitset.c

```
$ dcc bitset.c print_bits.c -o bitset
$ ./bitset

Set members can be 0-63, negative number to finish

Enter set a: 1 2 4 8 16 32 -1

Enter set b: 5 4 3 33 -1
a = 0000000000000000000000000000000100000000000000010000000100010110 = 0x100010116 =
4295033110
b = 0000000000000000000000000000001000000000000000000000000000111000 = 0x200000038 =
8589934648
a = {1,2,4,8,16,32}
b = {3,4,5,33}
a union b = {1,2,3,4,5,8,16,32,33}
a intersection b = {4}
cardinality(a) = 6
is_member(42, a) = 0
```

Exercise 3

Write the following in 8 bits of binary for each of the following:

- `25, 65, ~0, ~~1, 0xFF, ~0xFF`
- `(01010101 & 10101010), (01010101 | 10101010)`
- `(x & ~x), (x | ~x)`

How do we do the following in C?

- Given an 8-bit input X, ensure the 3rd bit from the RHS is 1?
- Given an 8-bit input Y, ensure the 3rd bit from the RHS is 0?
- Given an 8-bit input Z, test if the 3rd bit from the RHS is 1?