Bitwise Operators

- CPUs typically provide instructions which operate on individual bits of values.
  - MIPS provides 13 bit manipulation instructions
  - other CPUs can provide more
- C provides 6 operators which operate on individual bits of values

```plaintext
x & y     // bitwise and
x | y     // bitwise or
x ^ y     // bitwise exclusive-or (XOR)
~ x       // bitwise not
x << n    // left shift
x >> n    // right shift
```

Bitwise AND: &

The & operator
- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical AND on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
00100111     AND    0  1
& 11100011
--------------
0  0  0
00100011
```

Used for e.g. checking whether a bit is set
Checking for Odd Numbers

The obvious way to check for odd numbers in C:

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

We can use `&` to achieve the same thing:

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

Bitwise OR: 

The `|` operator

- takes two values (1, 2, 4, 8 bytes), treats as sequence of bits
- performs logical OR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
00100111       OR | 0 1
| 11100011
-------
0 0 1
```

Used for e.g. ensuring that a bit is set

Bitwise NEG: ~

The `~` operator

- takes a single value (1, 2, 4, 8 bytes), treats as sequence of bits
- performs logical negation of each bit
- result contains same number of bits as input

Example:

```
~ 00100111     NEG | 0 1
------
11011000     1 0
```

Used for e.g. creating useful bit patterns
**Bitwise XOR: ^**

The `^` operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical XOR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

\[
\begin{array}{c|c|c}
00100111 & ^ & 11100011 \\
11100011 \hline
00100111 & 0 & 1 \\
11100011 & 0 & 1 \\
11001000 & 1 & 0 \\
\end{array}
\]

Used in e.g. generating hashes, graphic operation, cryptography

---

**Left Shift: <<**

The `<<` operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- also takes a small positive integer \(x\)
- moves (shifting) each bit \(x\) positions to the left
- left-end bit vanishes; right-end bit replaced by zero
- result contains same number of bits as input

Example:

\[
\begin{align*}
00100111 & \quad << \quad 2 & & 00100111 & \quad << \quad 8 \\
----------- & \quad ----------- & & ----------- & \quad ----------- \\
10011100 & & 00000000 & & 00000000 \\
\end{align*}
\]

---

**Right Shift: >>**

The `>>` operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- also takes a small positive integer \(x\)
- moves (shifting) each bit \(x\) positions to the right
- right-end bit vanishes; left-end bit replaced by zero(*)
- result contains same number of bits as input

Example:

\[
\begin{align*}
00100111 & \quad >> \quad 2 & & 00100111 & \quad >> \quad 8 \\
----------- & \quad ----------- & & ----------- & \quad ----------- \\
00001001 & & 00000000 & & 00000000 \\
\end{align*}
\]

- shifts involving negative values are not portable (implementation defined)
- common source of bugs in COMP1521 and elsewhere
- always use unsigned values/variables to be safe/portable.
### MIPS - Bit Manipulation Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and r_d, r_s, r_t</code></td>
<td>( r_d = r_s \land r_t )</td>
<td>000000000000000000000000000000001000</td>
</tr>
<tr>
<td><code>or r_d, r_s, r_t</code></td>
<td>( r_d = r_s \lor r_t )</td>
<td>000000000000000000000000000000001001</td>
</tr>
<tr>
<td><code>xor r_d, r_s, r_t</code></td>
<td>( r_d = r_s \oplus r_t )</td>
<td>000000000000000000000000000000001010</td>
</tr>
<tr>
<td><code>nor r_d, r_s, r_t</code></td>
<td>( r_d = \neg (r_s \lor r_t) )</td>
<td>000000000000000000000000000000001011</td>
</tr>
<tr>
<td><code>andi r_t, r_s, I</code></td>
<td>( r_t = r_s \land I )</td>
<td>000000000000000000000000000000001111</td>
</tr>
<tr>
<td><code>ori r_t, r_s, I</code></td>
<td>( r_t = r_s \lor I )</td>
<td>000000000000000000000000000000001110</td>
</tr>
<tr>
<td><code>xori r_t, r_s, I</code></td>
<td>( r_t = r_s \oplus I )</td>
<td>000000000000000000000000000000001110</td>
</tr>
<tr>
<td><code>not r_d, r_s</code></td>
<td>( r_d = \neg r_s )</td>
<td>pseudo-instruction</td>
</tr>
</tbody>
</table>

spim translates `not r_d, r_s` to `nor r_d, r_s`, \$0

### MIPS - Shift Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sllv r_d, r_t, r_s</code></td>
<td>( r_d = r_t \ll r_s )</td>
<td>000000000000000000000000000000001000</td>
</tr>
<tr>
<td><code>srlv r_d, r_t, r_s</code></td>
<td>( r_d = r_t \gg r_s )</td>
<td>000000000000000000000000000000001001</td>
</tr>
<tr>
<td><code>srav r_d, r_t, r_s</code></td>
<td>( r_d = r_t \gg r_s )</td>
<td>000000000000000000000000000000001011</td>
</tr>
<tr>
<td><code>sll r_d, r_t, I</code></td>
<td>( r_d = r_t \ll I )</td>
<td>000000000000000000000000000000001111</td>
</tr>
<tr>
<td><code>srl r_d, r_t, I</code></td>
<td>( r_d = r_t \gg I )</td>
<td>000000000000000000000000000000001110</td>
</tr>
<tr>
<td><code>sra r_d, r_t, I</code></td>
<td>( r_d = r_t \gg I )</td>
<td>000000000000000000000000000000001111</td>
</tr>
</tbody>
</table>

srl and srlv shift zeros into most-significant bit
- this matches shift in C of `unsigned` value
sra and srav propagate most-significant bit
- this ensure shifting a negative number divides by 2
spim provides rol and ror pseudo-instructions which rotate bits
- real instructions on some MIPS versions
- no simple C equivalent

### bitwise.c: showing results of bitwise operation

```c
$ dcc 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 = 0110100111100001 = 0x69c1 = 27073
a >> c = 0000101100111111 = 0x0b3f = 2879
a << c = 1100111111111111 = 0xcfc0 = 53184
```
### bitwise.c: code

```c
uint16_t a = 0;
printf("Enter a: ");
scanf("%hd", &a);
uint16_t b = 0;
printf("Enter b: ");
scanf("%hd", &b);
printf("Enter c: ");
int c = 0;
scanf("%d", &c);
print_bits_hex("a = ", a);
print_bits_hex("b = ", b);
print_bits_hex("~a = ", ~a);
print_bits_hex("a & b = ", a & b);
print_bits_hex("a | b = ", a | b);
print_bits_hex("a ^ b = ", a ^ b);
print_bits_hex("a >> c = ", a >> c);
print_bits_hex("a << c = ", a << c);
```

### shift_as_multiply.c: using shift to multiply by \(2^n\)

```c
int n = strtol(argv[1], NULL, 0);
uint32_t power_of_two;
int n_bits = 8 * sizeof power_of_two;
if (n >= n_bits) {
    fprintf(stderr, "n is too large\n");
    return 1;
}
power_of_two = 1;
power_of_two = power_of_two << n;
printf("2 to the power of %d is %u\n", n, power_of_two);
printf("In binary it is: ");
print_bits(power_of_two, n_bits);
printf("\n");
```
$ dcc set_low_bits.c print_bits.c -o n_ones
$ ./set_low_bits 3
The bottom 3 bits of 7 are ones:
0000000000000000000000000111
$ ./set_low_bits 19
The bottom 19 bits of 524287 are ones:
000000000000000000000001111111
$ ./set_low_bits 29
The bottom 29 bits of 536870911 are ones:
000111111111111111111111111111

```c
int n = strtol(argv[1], NULL, 0);
uint32_t mask;
int n_bits = 8 * sizeof(mask);
assert(n >= 0 && n < n_bits);
mask = 1;
mask = mask << n;
mask = mask - 1;
printf("The bottom \%d bits of \%u are ones:
", n, mask);
print_bits(mask, n_bits);
printf("\n");
```

$ 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:
000000000000000000000000111111
$ ./set_bit_range 8 15
Bits 8 to 15 of 65280 are ones:
00000000000000000000000000000000
$ ./set_bit_range 8 23
Bits 8 to 23 of 16776960 are ones:
00000000000000000000000000000000
$ ./set_bit_range 1 30
Bits 1 to 30 of 2147483646 are ones:
01111111111111111111111111111110
```c
int low_bit = strtol(argv[1], NULL, 0);
int high_bit = strtol(argv[2], NULL, 0);
uint32_t mask;
int n_bits = 8 * sizeof(mask);

int mask_size = high_bit - low_bit + 1;
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;
printf("Bits %d to %d of %u are ones:\n", low_bit, high_bit, mask);
print_bits(mask, n_bits);
printf("\n");
```

```c
$ dcc extract_bit_range.c print_bits.c -o extract_bit_range
$ ./extract_bit_range 4 7 42
Value 42 in binary is:
000000000000000000000000101010
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
```

```c
int mask_size = high_bit - low_bit + 1;
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;
// get a value with the bits outside the range low_bit..high_bit set to zero
uint32_t extracted_bits = value & mask;
// right shift the extracted_bits so low_bit becomes bit 0
extracted_bits = extracted_bits >> low_bit;
printf("Value %u in binary is:\n", value);
print_bits(value, n_bits);
printf("\n");
printf("Bits %d to %d of %u are:\n", low_bit, high_bit, value);
print_bits(extracted_bits, mask_size);
printf("\n");
```

```c
extract_bit_range.c: using << and - to set a range of bits
```

```c
extract_bit_range.c: extracting a range of bits
```

```c
extract_bit_range.c: extracting a range of bits
```
print_bits.c: extracting the n-th bit of a value

```c
void print_bits(uint64_t value, int how_many_bits) {
    // print bits from most significant to least significant
    for (int i = how_many_bits - 1; i >= 0; i--) {
        int bit = get_nth_bit(value, i);
        printf("%d", bit);
    }
}
```

// extract the nth bit from a value
```c
int get_nth_bit(uint64_t value, int n) {
    // shift the bit right n bits
    // this leaves the n-th bit as the least significant bit
    uint64_t shifted_value = value >> n;
    // zero all bits except the the least significant bit
    int bit = shifted_value & 1;
    return bit;
}
```

print_int_in_hex.c: print an integer in hexadecimal

- write C to print an integer in hexadecimal instead of using:
  ```c
  printf("%x", n)
  ```

```
$ dcc print_int_in_hex.c -o print_int_in_hex
$ ./print_int_in_hex
Enter a positive int: 42
42 = 0x0000002A
$ ./print_int_in_hex
Enter a positive int: 65535
65535 = 0x0000FFFF
$ ./print_int_in_hex
Enter a positive int: 3735928559
3735928559 = 0xDEADBEEF
```

print_int_in_hex.c: main

```c
int main(void) {
    uint32_t a = 0;
    printf("Enter a positive int: ");
    scanf("%u", &a);
    printf("%u = 0x", a);
    print_hex(a);
    printf("\n");
    return 0;
}
```
print_int_in_hex.c: print_hex - extracting digit

```c
// sizeof returns number of bytes in n's representation
// each byte is 2 hexadecimal digits
int n_hex_digits = 2 * (sizeof n);
// print hex digits from most significant to least significant
for (int which_digit = n_hex_digits - 1; which_digit >= 0; which_digit--)
{
    // shift value across so hex digit we want
    // is in bottom 4 bits
    int bit_shift = 4 * which_digit;
    uint32_t shifted_value = n >> bit_shift;
    // mask off (zero) all bits but the bottom 4 bytes
    int hex_digit = shifted_value & 0xF;
    // hex digit will be a value 0..15
    // obtain the corresponding ASCII value
    // "0123456789ABCDEF" is a char array
    // containing the appropriate ASCII values (+ a '\0')
    int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
    putchar(hex_digit_ascii);
}
```

int_to_hex_string.c: convert int to a string of hex digits

Write C to convert an integer to a string containing its hexadecimal digits.

Could use the C library function snprintf to do this.

```bash
$ dcc int_to_hex_string.c -o int_to_hex_string
$ ./int_to_hex_string
Enter a positive int: 42
42 = 0x0000002A
$ ./int_to_hex_string
Enter a positive int: 65535
65535 = 0x0000FFFF
$ ./int_to_hex_string
Enter a positive int: 3735928559
3735928559 = 0xDEADBEEF
$ 
```

int_to_hex_string.c: main

```c
int main(void) {
    uint32_t a = 0;
    printf("Enter a positive int: ");
    scanf("%u", &a);
    char *hex_string = int_to_hex_string(a);
    // print the returned string
    printf("%u = 0x%s\n", a, hex_string);
    free(hex_string);
    return 0;
}
```
int_to_hex_string.c: convert int to a string of hex digits

```
// size of return number of bytes in n's representation
// each byte is 2 hexadecimal digits
int n_hex_digits = 2 * (sizeof n);

// allocate memory to hold the hex digits + a terminating 0
char *string = malloc(n_hex_digits + 1);

// print hex digits from most significant to least significant
for (int which_digit = 0; which_digit < n_hex_digits; which_digit++) {
    // shift value across so hex digit we want
    // is in bottom 4 bits
    int bit_shift = 4 * which_digit; uint32_t shifted_value = n >> bit_shift;
    // mask off (zero) all bits but the bottom 4 bits
    int hex_digit = shifted_value & 0xF;
    // hex_digit will be a value 0..15
    // obtain the corresponding ASCII value
    // "0123456789ABCDEF" is a char array
    // containing the appropriate ASCII values
    int hex_digit_ascii = "0123456789ABCDEF"[(hex_digit + 1) % 16];
    // terminate the array
    string[n_hex_digits - which_digit - 1] = hex_digit_ascii;
}

// return string
return string;
```
```c
uint32_t hex_string_to_int(char *hex_string) {
    uint32_t value = 0;
    for (int i = 0; hex_string[i] != 0; i++) {
        int ascii_hex_digit = hex_string[i];
        int digit_as_int = hex_digit_to_int(ascii_hex_digit);
        value = value << 4;
        value = value | digit_as_int;
    }
    return value;
}
```

```c
int hex_digit_to_int(int ascii_digit) {
    if (ascii_digit >= '0' && ascii_digit <= '9') {
        // the ASCII characters '0' ... '9' are contiguous
        // in other words they have consecutive values
        // so subtract the ASCII value for '0' yields the corresponding integer
        return ascii_digit - '0';
    }
    if (ascii_digit >= 'A' && ascii_digit <= 'F') {
        // for characters 'A' .. 'F' obtain the
        // corresponding integer for a hexadecimal digit
        return 10 + (ascii_digit - 'A');
    }
    fprintf(stderr, "Bad digit '%c'\n", ascii_digit);
    exit(1);
}
```

```c
// 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
```
```c
int xor_value = strtol(argv[1], NULL, 0);
if (xor_value < 0 || xor_value > 255) {
    fprintf(stderr, "Usage: %s <xor-value>\n", argv[0]);
    return 1;
}
int c;
while ((c = getchar()) != EOF) {
    // exclusive-or
    // ^ | 0 1
    // ----|-----
    // 0 | 0 1
    // 1 | 1 0
    int xor_c = c ^ xor_value;
    putchar(xor_c);
}
```

```
$ echo Hello Andrew|xor 42
bOFFE
kDNX0
$ echo Hello Andrew|xor 42|cat -A
bOFFE$
kDNX0
$ echo Hello |xor 42
bOFFE $ echo -n 'bOFFE '|xor 42
Hello
$ echo Hello|xor 123|xor 123
Hello
$
```
**pokemon.c: using an int to represent a set of values**

- Simple example of a single integer specifying a set of values
- Interacting with hardware often involves this sort of code

```c
uint16_t our_pokemon = BUG_TYPE | POISON_TYPE | FAIRY_TYPE;
```

// example code to check if a pokemon is of a type:
```c
if (our_pokemon & POISON_TYPE) {
    printf("Poisonous\n"); // prints
}
if (our_pokemon & GHOST_TYPE) {
    printf("Scary\n"); // does not print
}
```

**pokemon.c: using an int to represent a set of values**

// example code to add a type to a pokemon
```c
our_pokemon |= GHOST_TYPE;
```

// example code to remove a type from a pokemon
```c
our_pokemon &= ~POISON_TYPE;
```

```c
printf(" our_pokemon type (2)\n");
if (our_pokemon & POISON_TYPE) {
    printf("Poisonous\n"); // does not print
}
if (our_pokemon & GHOST_TYPE) {
    printf("Scary\n"); // prints
}
```

**bitset.c: using an int to represent a set of values**

```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 = 0000000000000000000000000000000010000000000000010000000100010110 = 0x1000101:
b = 0000000000000000000000000000001000000000000000000000000000111000 = 0x2000000:
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
```
bitset.c: main

printf("Set members can be 0-%d, negative number to finish\n", MAX_SET_MEMBER);
set a = set_read("Enter set a: ");
set b = set_read("Enter set b: ");
print_bits_hex("a = ", a);
print_bits_hex("b = ", b);
set_print("a = ", a);
set_print("b = ", b);
set_print("a union b = ", set_union(a, b));
set_print("a intersection b = ", set_intersection(a, b));
printf("cardinality(a) = %d\n", set_cardinality(a));
printf("is_member(42, a) = %d\n", (int)set_member(42, a));

bitset.c: common set operations

set set_add(int x, set a) {
    return a | ((set)1 << x);
}

set set_union(set a, set b) {
    return a | b;
}

set set_intersection(set a, set b) {
    return a & b;
}

// return 1 iff x is a member of a, 0 otherwise
int set_member(int x, set a) {
    assert(x >= 0 && x < MAX_SET_MEMBER);
    return (a >> x) & 1;
}

bitset.c: counting set members

// return size of set
int set_cardinality(set a) {
    int n_members = 0;
    while (a != 0) {
        n_members += a & 1;
        a >>= 1;
    }
    return n_members;
}
set set_read(char *prompt) {
    printf("%s", prompt);
    set a = EMPTY_SET;
    int x;
    while (scanf("%d", &x) == 1 && x >= 0) {
        a = set_add(x, a);
    }
    return a;
}

void set_print(char *description, set a) {
    printf("%s", description);
    printf("\{\n");
    int n_printed = 0;
    for (int i = 0; i < MAX_SET_MEMBER; i++) {
        if (set_member(i, a)) {
            if (n_printed > 0) {
                printf(",\n");
            }
            printf("%d", i);
            n_printed++;
        }
    }
    printf("\}\n");
}

Exercise: Bitwise Operations

Given the following variable declarations:

// a signed 8-bit value
unsigned char x = 0x55;
unsigned char y = 0xAA;

What is the value of each of the following expressions:

- \((x \& y)\) (x ^ y)
- \((x << 1)\) (y << 1)
- \((x >> 1)\) (y >> 1)
Exercise: Bit-manipulation

Assuming 8-bit quantities and writing answers as 8-bit bit-strings:

What are the values of the following:

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

How can we achieve each of the following:

- ensure that the 3rd bit from the RHS is set to 1
- ensure that the 3rd bit from the RHS is set to 0