UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CSE013E: COMPUTER SYSTEMS AND "C" PROGRAMMING

Bit Manipulation

Introduction

In a computer all data is stored as numbers, which are themselves stored as a sequence of bits. In many instances it is useful to directly manipulate these bits. This could be to compress data, optimize calculations, or to manage packed Boolean data. It can even be useful when dealing with processor-specific registers that control hardware functions.

Operators

For performing bit manipulations, C offers a wide selection of operators. It should be noted that these operators are only defined for integer variables.

|, &, ~, ^: These operators are used for the basic bitwise operations of OR, AND, NOT, and XOR. They are called bitwise because they operate between equivalent bits in separate numbers simultaneously. These can also be used in augmented assignment where they modify the left-hand operand based on the right-hand one: |=, &=, ^=.

<<, >>: These operators provide bit-shifting capabilities, one for left-shifting and one for right-shifting. Individual bits are not moved, but every bit is shifted by the right-hand operand. As bits are shifted off of the end of the storage range, they are removed and new bits shifted in during this are 0s. Be warned that when using the bit-shift operators, that they have higher precedence than the bitwise-OR, bitwise-AND, and bitwise-XOR and the use of parenthesis may be required. Similar to the bitwise Boolean operators, these operators can also augment assignment: <<=, >>=.

Basic Operations

The following examples demonstrate the basic operations available from bitwise operators. More complicated operations can be performed by compounding many basic operations together, possibly including arithmetic operations. Constants used for bitwise operations are usually written in capitalized hexadecimal notation (like 0x1A); this is because it is easier to read than binary or decimal since each hexadecimal digit maps directly to groups of 4 bits each. Also note that bits are numbered starting from 0 and going from right-to-left, though cardinality still starts at 1: in the number 0x1A, bit 3 is the 4^{th} bit and is a 1.

Setting a bit refers to making a bit a 1 value regardless of what it was before. This can be done by OR-ing the appropriate bit with a 1 (make a truth table to see why). Setting the 4^{th} bit of a number can be done as follows. Note the use of a *bitmask*, which is a

selector for which bits are being operated on. In this case the bits in the bitmask that are high indicate which bits will be set to 1.

177 in binary	`1011 0001`
Bitmask for setting the 4th bit	`0000 1000`
Result	`1011 1001`

Performing this operation in C can be done with the augmented assignment for OR:

```
value |= 0x10;
```

Clearing a bit refers to clearing a bit's value to 0. This can be done by AND-ing the appropriate bit with a 0 (make a truth table to see why). This means that the bitmask will have 0s to indicate which bits will be cleared. To clear bits 3 and 7 of a number:

177 in binary	`1011 0001`
Bitmask for clearing the 4th and 7th bit	`0111 0111`
Result	`0011 0001`

Performing this operation in C can be done with the augmented assignment for AND:

```
value &= 0x77;
```

Toggling a bit refers to switching its value and is a bitwise-XOR operation done with the ^ operator. The following code inverts bit 3 and 4 of the number:

177 in binary	`1011 0001`
Bitmask for toggling the 4th and 5th bit	`0001 1000`
Result	`1010 1001`

Performing this operation in C can be done with the augmented assignment for XOR:

```
value ^= 0x0004;
```

Bit-toggling can also be done for all bits by using the bitwise-not operator (~). This can be useful for making constants easier to read. By writing constants as a NOT of a number, it also makes the constants independent of the bit-length of the datatype. The following code would clear the desired bits regardless of whether `value` is 8 bits or 64:

```
value &= ~0x98;
```

Reading a bit in a number, such as a flag or Boolean setting, can be done with the bitwise-AND operator using a bitmask where the bits that should be read are 1. In this example only the lowest-7 bits of `ADC1BUF0` register is stored in `value`:

```
int value = ADC1BUF0 & 0x007F;
```

By adding specific comparisons using 0 or the bitmask, reading a bit can be extended to a few useful comparison operations:

If all bits are set	`if ((VARIABLE & MASK) == MASK) { ... }`

If any bits are set	`if ((VARIABLE & MASK) != 0) { ... }`
	
	OR
	
	`if (VARIABLE & MASK) { ... }`

If all bits are clear	`if ((VARIABLE & MASK) == 0) { ... }`
If any bits are clear	`if ((VARIABLE & MASK) != MASK) { ... }`

Moving bits can be accomplished with the << and >> operators. These operators move bits left or right by whatever number follows them. They have a myriad of uses, declaring constants where specific bits are in specific locations. In the following example, the integer value indicates that the system has GPS onboard if bit 6 is high, a temperature sensor if bit 3 is high, and a motor if bit 15 is high. To specify that the system has all of these features, either of the following can be written, though it should be clear which is more readable:

```
uint16_t onboardSensors = (1 << 15) | (1 << 6) | (1 << 3);
                          OR
uint16_t onboardSensors = 0x8048;
```

A right-shift can also be useful for removing the low-order bits of a number, for example when removing parts of a number and wanting to work with it using ordinary arithmetic. If `bigData` is a 32-bit number, but only the top byte is required, the following can be done and then value

```
uint32_t value = bigData >> 24;
```

Arithmetic operations can also be done with the bitwise operators. Bitwise operations are also commonly faster than arithmetic operations by up to a couple orders of magnitude. If high performance is required, relying on bitwise operations will be essential. The provided table lists some common arithmetic operations that can be performed by the bitwise operators.:

Multiplication by a multiple of 2	`NUMBER << log2(MULTIPLE)` `34 * 16 ≡ 34 << 4 (= 544)`

Division by a multiple of 2 (rounding down)	`NUMBER >> log2(MULTIPLE)` `234 / 32 ≡ 234 >> 5 (= 7)`

Limit number
(number must be a multiple of 2 - 1)
	`NUMBER & CAP`
	
	Limit number to 1023 max:
	`int x = y & 1023;`

Invert range
(where range is from 0 to multiple of 2 - 1)
	`~NUMBER & RANGE`
	
	Invert number in the range [127]:
	`int x = ~y & 127`

Compound Operations

While the bitwise operators may seem only somewhat useful at this point, it is their combination which can result in some powerful compound operations. The following examples demonstrate actual problems encountered in real embedded coding projects and their solutions using bitwise and arithmetic operations.

1) Switching endianness of a 32-bit number:

To fully understand endianness, I refer you to Wikipedia or any introductory computer architecture textbook. In brief the endianness of a number indicates how the individual bytes composing it are stored. The bytes are stored such that increasing values go from either left-to-right or right-to-left. In embedded systems it is common to have systems of different endianness communicating, which requires converting the received data. The following code converts a big-endian 32-bit integer into a little-endian one (note the use of parenthesis to enforce the proper order of evaluation):

```
int32_t output = (input & 0xFF000000) >> 24 |
                 (input & 0x00FF0000) >> 8  |
                 (input & 0x0000FF00) << 8  |
                 (input & 0x000000FF) << 24;
```

2) Enabling pins

On processors hardware settings are controlled by setting the values of integers in specific memory locations, referred to as special function registers (SFRs). For example, on the PIC32 processor the analog-to-digital conversion (ADC) hardware is partially configured via the `AD1PCFGL` variable that is mapped to a specific memory location by provided libraries. Say for example that the atmospheric pressure sensor is connected to the 1st ADC, the temperature sensor to the 4th, and the position sensor to the 5th for a

robot. To read these values, the ADC must be configured to enable these inputs through the 16-bit ADC1PCFGL register by clearing the appropriately-numbered bits:

```
AD1PCFGL &= ~(0x0015);
```

3) Bit-masking to filter CAN messages

The Controller Area Network (CAN) is a protocol that defines a high-bandwidth, noise-tolerant bus that many devices can use to share data between each other. It operates similar to the Internet Protocol (IP) that powers the Internet. The CAN protocol is actively used in the automotive and marine industries for connecting a myriad of microcontrollers and sensors within the vehicle. Additionally the CAN protocol is also used for the OBD-2 diagnostics connection that car mechanics use to diagnose car troubles.

For the CAN 2.0B protocol, data is sent as a packet with up to 8 bytes of data and a 29-bit identifier specifying the type of data. This identifier is used by devices on the bus to filter out only messages that relate to them. For example, this allows the vehicle's cruise control microcontroller to ignore messages sent about the air pressure in each tire while still communicating the vehicle's main electronic control unit (ECU).

While this filtering can be done in hardware, it can be simpler to implement in software using bit masks. For example if all of the CAN packets that relate to the cruise control functionality only use bits 6, 7, and 8 in the identifier, the rest of the bits are irrelevant. Those bits can be ignored by utilizing a bit mask:

```
int filterMask = 0x000001C0;
switch (msgIdentifier & filterMask) {
case 0x00000040:
      // Handle messages with 001 in bits 6, 7, and 8
      break;
case 0x00000080:
      // Handle messages with 010 in bits 6, 7, and 8
      break;
case 0x000000C0:
      // Handle messages with 011 in bits 6, 7, and 8
      break;
                                    .
                                    .
                                    .
}
```

Bitfields

As working with bit-packed data is very common, there is actually a way to specify bitfields as part of the C language. The following code declares a bitfield for a 16-bit integer with 3 components, an EID field that's 2 bits long, a gap of unused space of 3 bits, and an 11-bit SID field.

```
struct canFilter {
         unsigned EID:2;
         unsigned :3;
         unsigned SID:11;
};
```

There are several things to notice about bitfields here:

- They add a ":BITSIZE" part to the standard struct definition.
- The "int" type was left out of the declaration as it is implied, all that is necessary is the declaration of the signedness of the field.
- There can be unnamed fields that refer to unused space. This is common when dealing with memory-mapped peripherals on processors as the used space is required to fit a word on that architecture.

There is a caveat to bitfields, though, in that they are very much NOT portable because almost all details are implementation specific, which means they can vary by compiler or processor type. This includes how large the resultant data type is, how the fields are aligned in memory, and the ordering of the fields. Even so, they do offer many benefits over working with raw integers. For example, setting the EID of a CanFilter struct versus an integer:

```
struct CanFilter x;
x.EID = 3;
x.SID = 77;
```

```
unsigned int x;
x = (x & ~0xC000) | (3 << 14);
x = (x & ~0x07FF) | 77;
```

Unions

While bitfields can be quite useful for individual fields, it can sometimes still be useful for accessing data as a complete integer. This is a perfect example of when the *union* can be useful. The union allows a variable to exist simultaneously as two separate datatypes that can be used interchangeably.

For example, unions are commonly defined for the special function registers in microcontrollers as part of the provided standard library. Continuing with the above example, a union can be used to define a variable as both an integer and the CanFilter struct:

```
union CanFilter {
         struct {
            unsigned EID:2;
            unsigned :3;
            unsigned SID:11;
         } bitfield;
         unsigned int raw;
}
```

And now modifying a variable with the datatype of `union CanFilter` can be done with either method:

```
union CanFilter x;
x.bitfield.EID = 3;
x.bitfield.SID = 77;
```

```
union CanFilter x;
x.raw = (x & ~0xC000) | (3 << 14);
x.raw = (x & ~0x07FF) | 77;
```