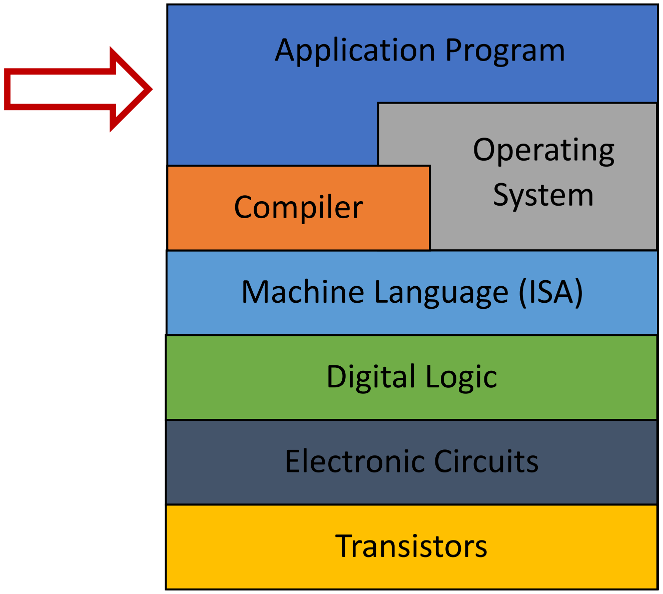
KU THE UNIVERSITY OF KANSAS

EEC 388: Embedded Systems

Lecture 4

Instructor: Tamzidul Hoque, Assistant Professor,
Dept. of EECS, University of Kansas
(hoque@ku.edu), office: Eaton 2038

Context



Application Program

Operating System

Compiler

Machine Language (ISA)

Digital Logic

Electronic Circuits

Transistors

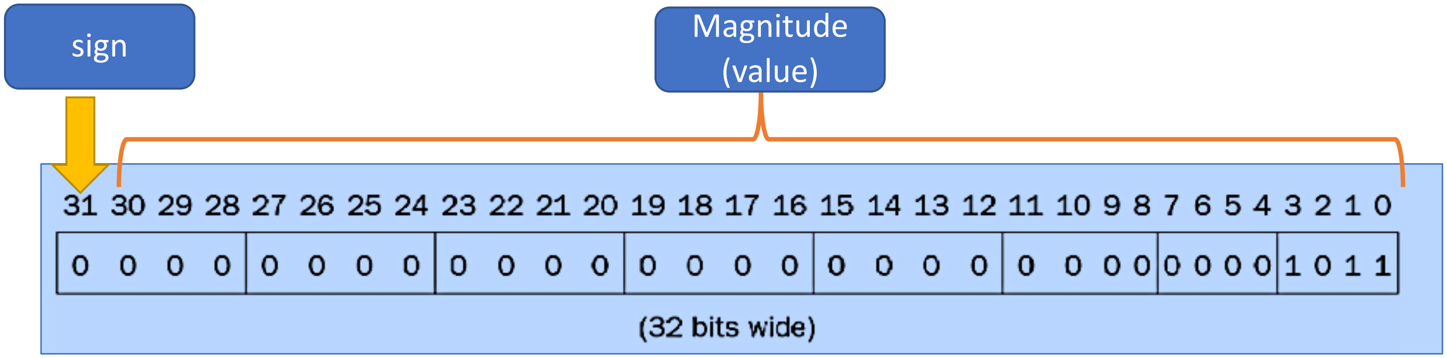
Representing Signed number

- Solution 1: **Sign magnitude method**: Reserve a bit to represent the sign
- Challenges:
  - *balance* – equal number of negatives and positives
  - *ambiguous zero* – There will be +0 and -0
  - How do we determine the sign of a number after arithmetic (like add)

Sign Magnitude:

000 = 0
001 = +1
010 = +2
011 = +3
100 = 0
101 = -1
110 = -2
111 = -3

ambiguous zero

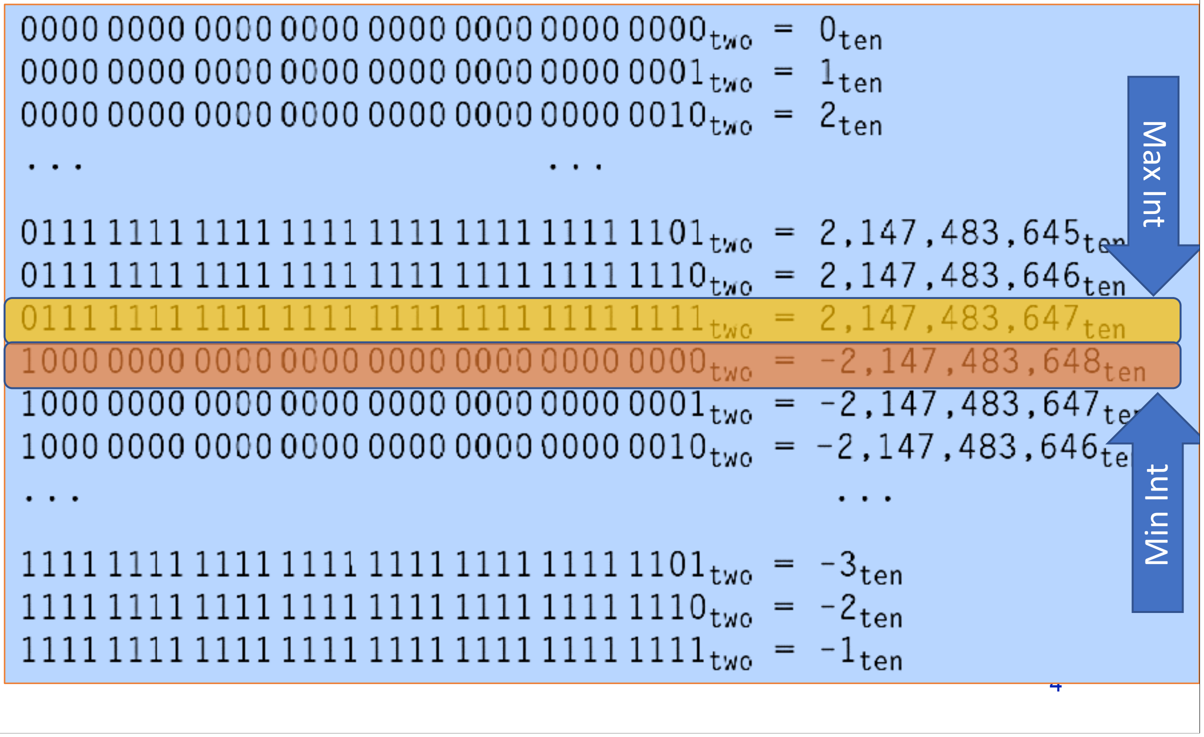


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Representing Signed number

- **Solution 2:** Using **“two’s complement”** representation
- Eliminates negative zero and makes arithmetic in hardware easier
- Positive half of the numbers starts with a 0 in MSB
- +ve Range: 0 to 2^{31}-1
- After that-ve numbers
- MSB of each –ve is 1
- **What is the –ve range?**



$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

. . . . . .

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{te}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{te}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{te}$

. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

In Class Quiz 1: Submit via Canvas

- Subtract 8 from 9 in 2’s complement negation method (without doing subtraction)
- Assume we have 8-bit digits (not 32 bits), ignore carry beyond 8 bits

- 9 in binary: 0000_1001
- 8 in binary: 0000_1000
- Step 1: -8 in binary: ?
- Step 2: 9+(-8) in binary:?

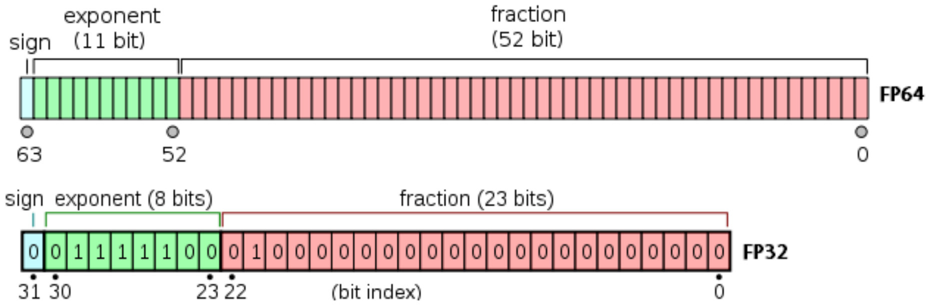
Fractional Numbers: Float and double

Float

- IEEE 754 single precision floating point numbers
- 1-bit sign, 8-bits exponent, 23-bits fraction
- 6 significant decimal digits of precision

Double

- 1-bit sign, 11-bits exponent, 52-bits fraction
- 15-17 significant decimal digits of precision



Operators

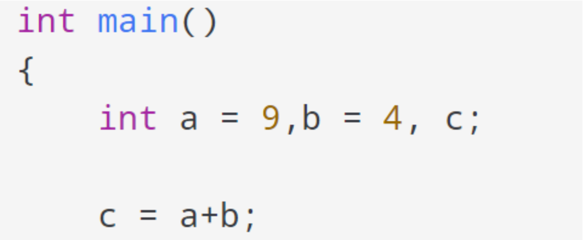
- A symbol that tells the compiler to perform specific mathematical or logical functions

- Used to manipulate data
- Arithmetic: +, -, /, *, ++, --, %
- Relational: <, <=, >, =>, ==, !=
- Logical: ||, &&, !
- Bitwise: <<, >>, |, &, ^, ~

Arithmetic Operators

Perform math operations

- + = add
- - = subtract
- / = divide
- * = multiply
- ++ = increment
- -- = decrement
- % = modulus (remainder)



```
int main()
{
    int a = 9,b = 4, c;

    c = a+b;
```

var++ → var = var + 1
var-- → var = var - 1

10 % 2 = 0
10 % 3 = 1

Increment and decrement (++, --)

var++ (used in prefix)

- First use the value and then increase it by one

++var (used in postfix)

- Increment the value and then use it

```
a = 1; b = 2; c = 3;
x = a-- + b++ - ++c; //x=1+2-4
x →  -1
```

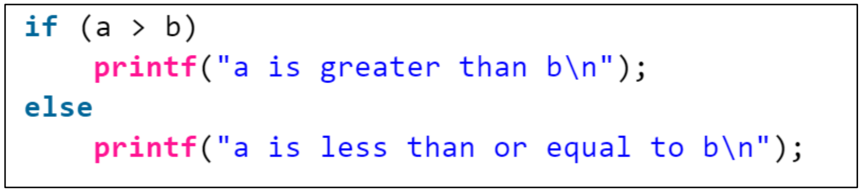
Example

```
int main(){

      int var1 = 100;

      printf("%d\n",        var1++);
      printf("%d\n",        ++var1);
      printf("%d\n",        var1--);
      printf("%d\n",        --var1);

return 0;
}
```

Relational Operators

- Checks the relationship between two operands.
- C uses its int type to represent true and false
- If the relation is true, a relational operation returns 1; if the relation is false, it returns value 0.

- <   = less than
- <=   = less than or equal
- >   = greater than
- =>   = greater than or equal
- ==   = equal
- !=   = not equal



```
if (a > b)
    printf("a is greater than b\n");
else
    printf("a is less than or equal to b\n");
```

Relational Operators: Example

```
main() {

int a = 21;
int b = 10;

if( a == b ) {
          printf("Line 1 - a is equal to b\n" );
} else {
          printf("Line 1 - a is not equal to b\n" );
}

printf (" %d ",  a > b);
printf (" %d ",  a < b);

…
```

Logical Operators

- Commonly used in Conditional expression in C programming (sometime combines multiple conditions)
  - if (cond0 || cond1)
  - if (cond0 && cond1)
  - if (!cond0)

- A logical operation returns either 0 or 1 (false/true)

- || = logical OR → Returns true only if either one operand is true
- && = logical AND → Returns true only if all operands are true
- ! = logical NOT → Returns true only if the operand is 0

Logical Operators: Example

- If the result of the logical operator is true then 1 is returned otherwise 0 is returned.

Let's say,

- int a = 5, b = 5, c = 10;

- If ((a == b) && (c > 5)) evaluates to True or ?
- If ((a == b) && (c < b)) evaluates to False o ?
- If ((a == b) || !(c > 5)) evaluates to T ?
- If (!(a == b) || (c < b)) evaluates to ?

Bitwise Operators

- Used for manipulating data at the bit level
- Operates on **int** type data (including short, long, unsigned), and returns int data types as well.

- | = bitwise OR
- & = bitwise AND
- ^ = bitwise XOR
- ~ = bitwise one's complement
- << = left shift
- >> = right shift

10011011 | 01101100 = 11110111

10011011 & 01101100 = 00001000

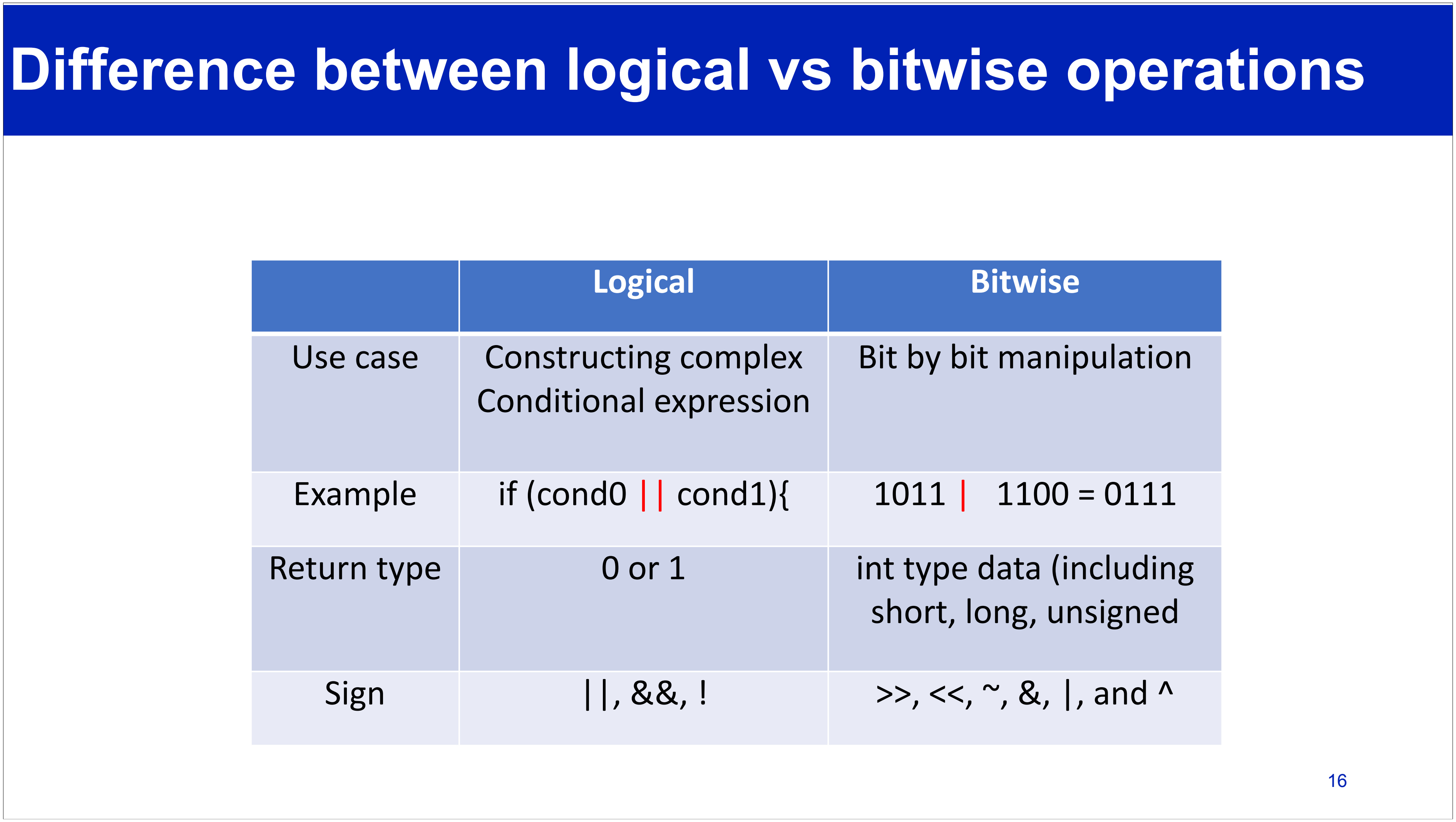
10011011 ^ 01101100 = 11110111

~ 10011011 = 01100100

10011011 << 2 = 011011**00**

10011011 >> 2 = **00**100110

Zero shifts in for "unsigned" types. How about signed ??



Difference between logical vs bitwise operations

	Logical	Bitwise
Use case	Constructing complex Conditional expression	Bit by bit manipulation
Example	if (cond0 || cond1){	1011 | 1100 = 0111
Return type	0 or 1	int type data (including short, long, unsigned
Sign	||, &&, !	>>, <<, ~, &, |, and ^

Bitwise vs Logical AND/OR

- Solve by yourself

Operand 1	Operand 2	Logical AND (A && B)	Bitwise AND (A & B)
0b1000	0b0111		
0b1100	0b1100		
0b0000	0b0000		

For logical AND/OR, any operand greater than 0 is assumed 1.

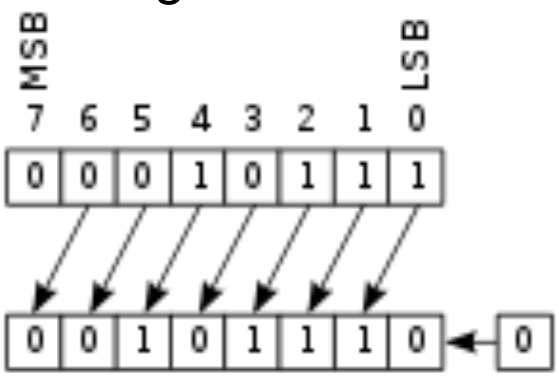
Assignment Operators

Arithmetic and bitwise operators can be combined with an assignment for simplified expressions

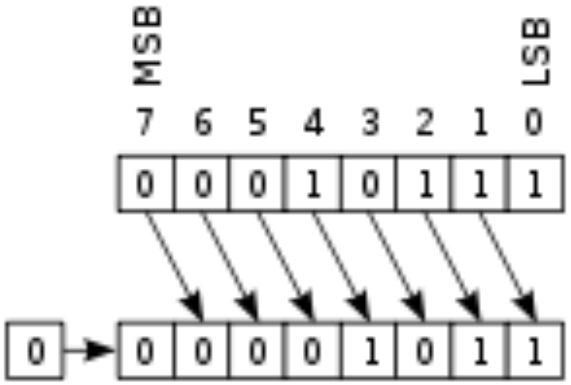
- var0 + = var1; → var0 = var0 + var1;
- var0 >> = 5; → var0 = var0 >> 5;

Bitwise Operators

- << = left shift.
  - Example: c=a<<2 // shift left by 2 bits
- >> = right shift .
  - Example: c=a>>3 // shift right by 3 bits
- Application of bitwise shift: Cryptographic algorithms, mult/division, encoding/decoding etc.



Logical left shift



Logical right shift

left shift

- Table below shows **left shift** for multiplication:
  - Applies for **unsigned number**

	Syntax	Binary (8 bit)	Decimal	Hint
	x=7	0000 0111	7	
Example 1	y=x<<1	0000 1110	14	7*2 or 7*2^1
Example 2	z=y<<3	0111 0000	112	14*8 or 7*2^3
Example 3	i=z<<2	1100 0000	192	Bit is lost

- Ex 1: left shifting by 1 bit is equivalent to multiplying by 2
- Ex 2: left shifting by **n** bits is equivalent to multiplying by **2^n**
- Ex 3: If a 1 passes the MSB due to shifting, the bit is lost and the above rule does not work

right shift

- The result of a Right Shift operation is a **division** by 2^n , where n is the number of shifted bit positions.
  - Applies **for unsigned number**
  - **Ignores the remainder**

	Syntax	Binary (4 bit)	Decimal	Hint
	x=7	**0111**	7	
Example 1	y=x>>1	0**011**	3	7/2=3.5
Example 2	z=x<<2	00**01**	1	7/4=1.75
Example 3	i=x<<4	0000	0	7/16=0.45

- Ex 1: right shifting by 1 bit is equivalent to division by 2
- Ex 2: Shifting by n bits is equivalent to division by 2^n

Flow Control Statements: Branching

Two main component:

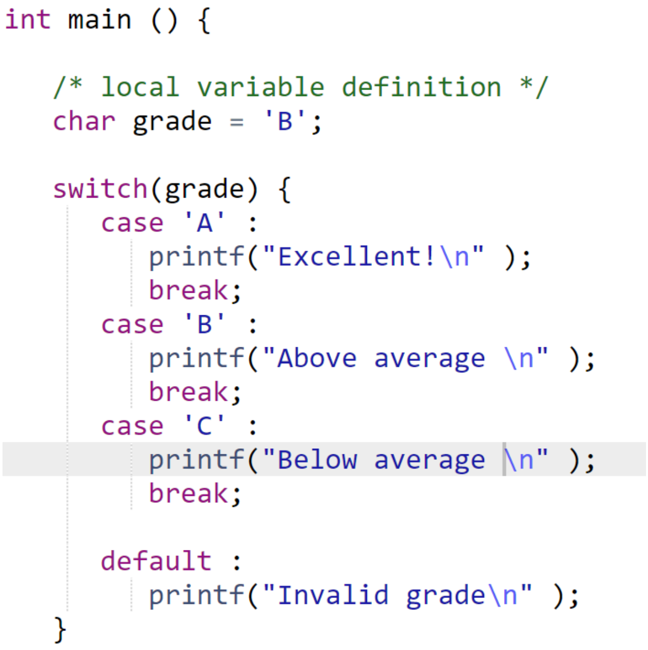
- An expression in parenthesis
- A statement or block of statements

If the expression is true:

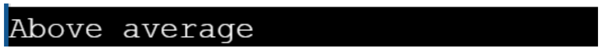
- then the statement or block of statements gets executed
- *Otherwise, these statements are skipped.*

```
if (condition) {
  // code
}

if (condition) {
  // code
} else {
  // code
}

if (condition) {
  // code
} else if (condition)
{
  // code
} else {
  // code
}
```

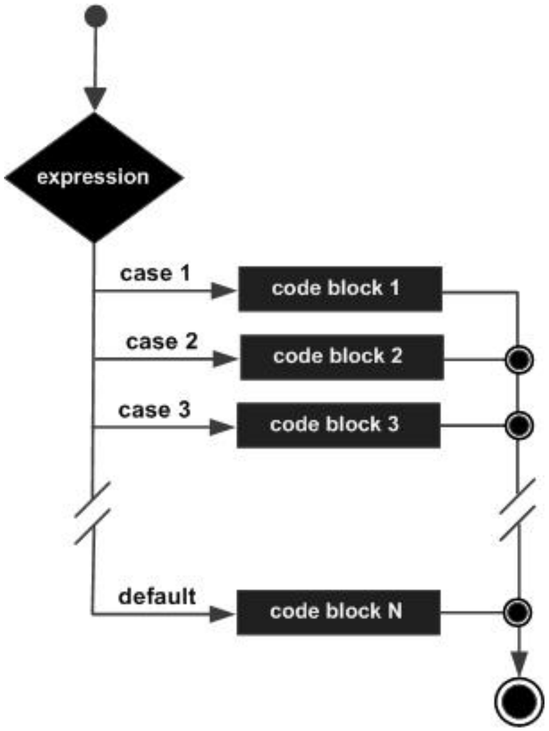
Branching using Switch Statement



```c
int main () {

   /* local variable definition */
   char grade = 'B';

   switch(grade) {
      case 'A' :
         printf("Excellent!\n" );
         break;
      case 'B' :
         printf("Above average \n" );
         break;
      case 'C' :
         printf("Below average \n" );
         break;

      default :
         printf("Invalid grade\n" );
   }
```



```
Above average
```



Flow Control Statements: Loops

Loops provide a way to repeat commands

- Provides control on how many times to loop

C provides a number of looping way

- While
- Do… while
- for

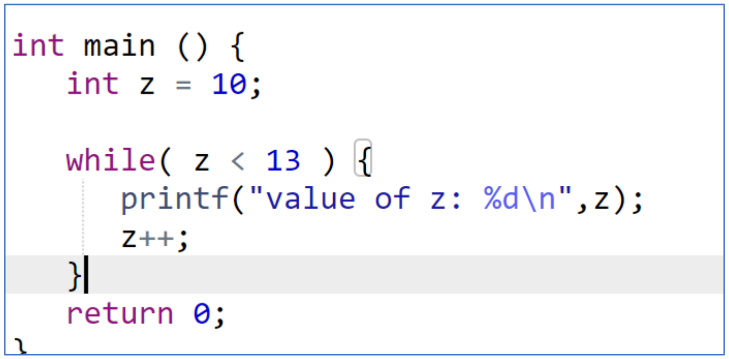
```
while (condition) {
  // code
}
```

```
do {
  // code
} while (condition);
```

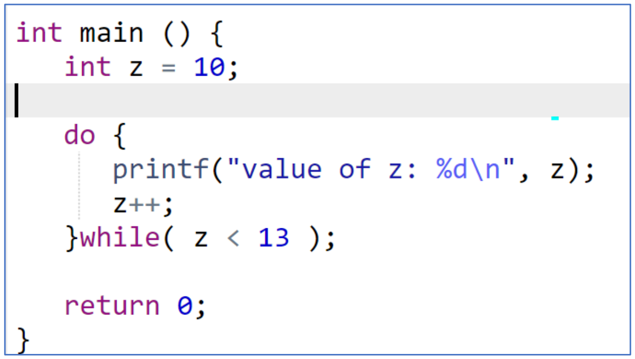
```
for (init;
condition;expression) {
  // code
}
```

While and Do… While

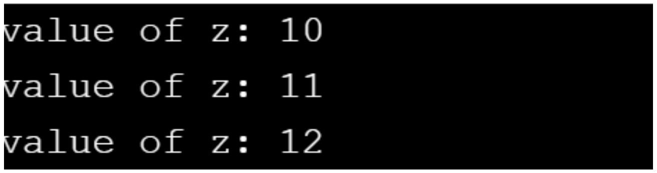
A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.



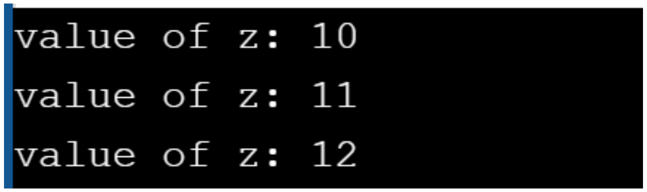
```
int main () {
   int z = 10;

   while( z < 13 ) {
      printf("value of z: %d\n",z);
      z++;
   }
   return 0;
}
```



```
int main () {
   int z = 10;

   do {
      printf("value of z: %d\n", z);
      z++;
   }while( z < 13 );

   return 0;
}
```



```
value of z: 10
value of z: 11
value of z: 12
```

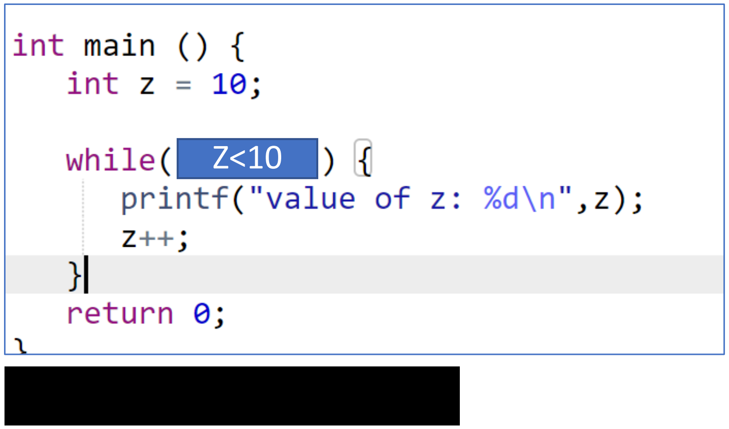


```
value of z: 10
value of z: 11
value of z: 12
```

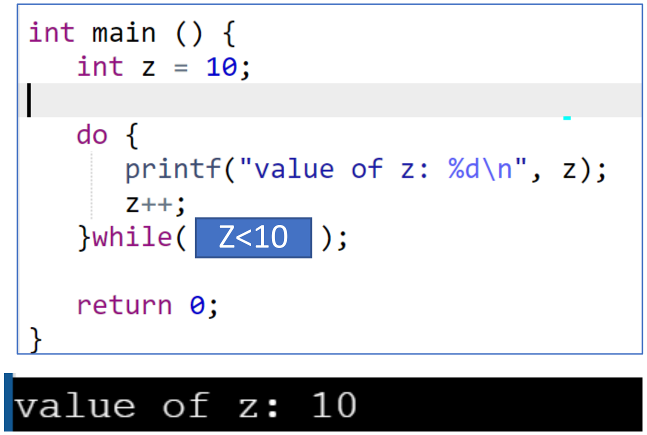
While and Do… While

If you change the relational statement inside the while for both of them to (z<10):

- For while: The print statement will never execute
- For while… do: The print statement will execute once



```
int main () {
   int z = 10;

   while( Z<10 ) {
      printf("value of z: %d\n",z);
      z++;
   }
   return 0;
}
```

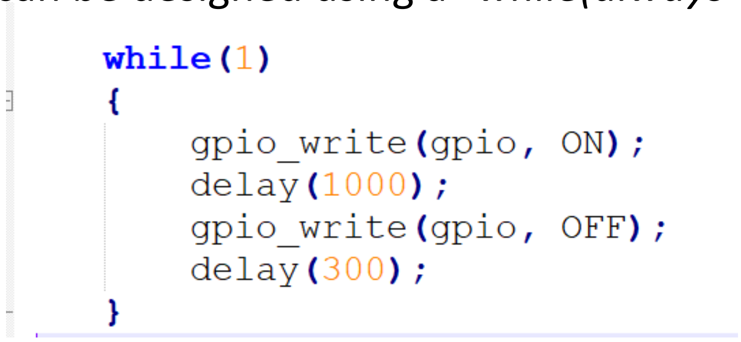


```
int main () {
   int z = 10;

   do {
      printf("value of z: %d\n", z);
      z++;
   }while( Z<10 );

   return 0;
}
```

```
value of z: 10
```

Use of while/do-while in embedded programming

- General purpose computers have operating system (OS).
- The OS waits for your instructions to run the desired programs/ application
- Many embedded systems do not have OS
- Programs written for such systems need infinite loop that constantly waits for user inputs
  - Otherwise, the program will just run once and exit
- Such infinite loop can be designed using a “*while(always true) {* “ style code



```
while(1)
{
    gpio_write(gpio, ON);
    delay(1000);
    gpio_write(gpio, OFF);
    delay(300);
}
```

Why we need do-while?

- Do-while lets the code execute once at least before checking the condition
  - But when we need such code?
  - Useful when the condition can be properly checked only after some operation is done
  - Ex: when the condition involves checking some data source that is only valid after first-round of code execution
  - Can also be achieved with while-do, but code readability will differ

```
do {
      Read_keyboard()
      …
      }
while (!keyboard_data=exit);
```

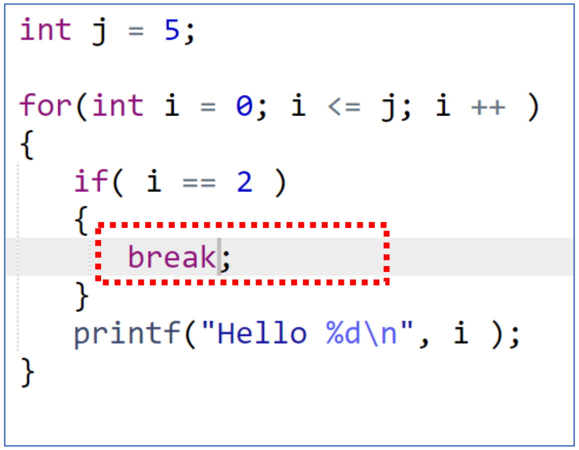
Break and Continue

```
while ( condition ) {

  // code 0
  if ( condition1 ) {
        break;
  }

  // code 1
  if ( condition2 ) {
     continue;
  }

  // code 2
}

// code 3
```

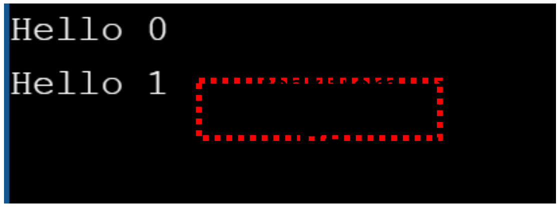
Exit while loop and start executing “code 3”

Skip this iteration and start executing the next iteration which can be “code 0” or “code 3”

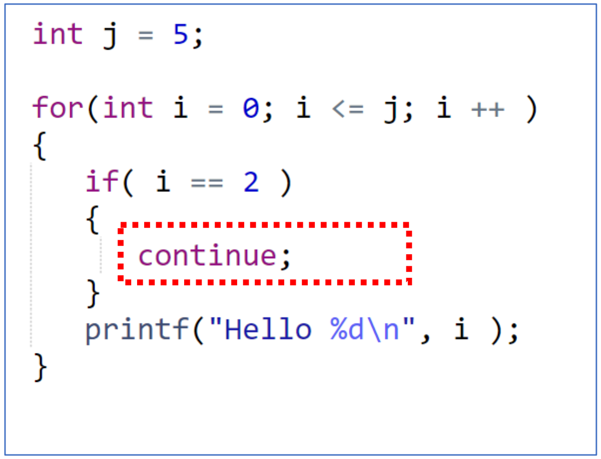
Break and Continue



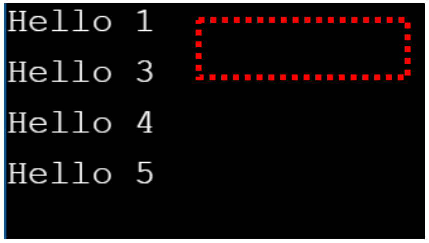
```
int j = 5;

for(int i = 0; i <= j; i ++ )
{
    if( i == 2 )
    {
        break;
    }
    printf("Hello %d\n", i );
}
```



```
Hello 0
Hello 1
```



```
int j = 5;

for(int i = 0; i <= j; i ++ )
{
    if( i == 2 )
    {
        continue;
    }
    printf("Hello %d\n", i );
}
```



```
Hello 1
Hello 3
Hello 4
Hello 5
```

Functions in C

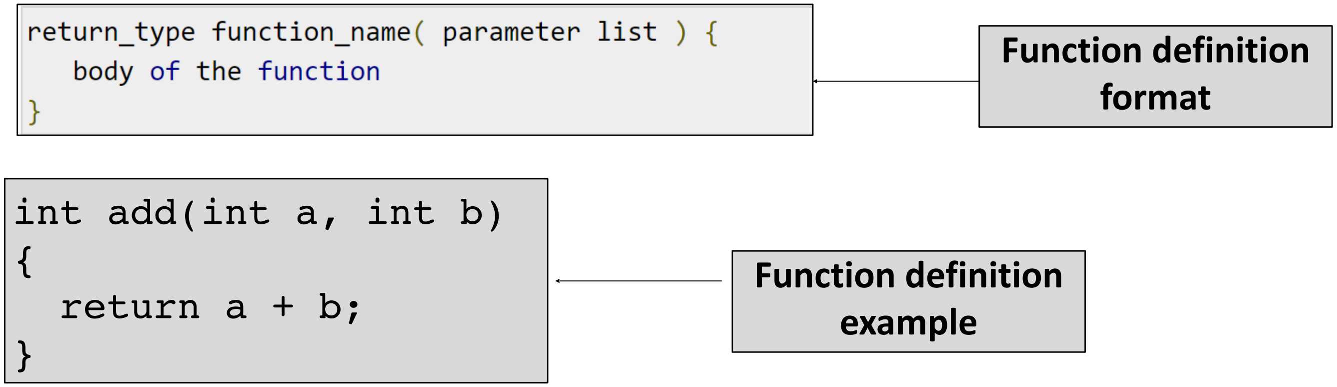
- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is main()
- A standard C library provides many built in functions
- We can define additional functions.

- Use of function involves three things
  - Function Definition
  - Function Declaration
  - Function Calling

Defining a Function

Function definition

- Contain Function header and function body.



```
return_type function_name( parameter list ) {
   body of the function
}
```

Function definition format

```
int add(int a, int b)
{
  return a + b;
}
```

Function definition example

If you have many functions, defining all within main() function is not desirable

- Use header file (in later slides)

Functions Declaration

- Function declaration
  - A function declaration tells the compiler about a function name and how to call the function
  - The actual body of the function can be defined separately

```
#include <stdio.h>

int add(int a, int b);

void main()
{
   int c = add(1, 1);
   printf(“%d\n”, c);
}
```

Function declaration
return_type function_name(parameter list);

- Parameter names are not important in function declaration, only their type is required
  - So we can also declare→ *int add(int, int);*

Calling a Function

- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value

```
#include <stdio.h>

int add(int a, int b);

void main()
{
   int c = add(1, 1);
   printf(“%d\n”, c);
}
```

Calling the function

- Parameter names are not important in function declaration only their type is required
  - So we can also declare→ *int add(int, int);*

Using Header files for function

- Defining all of function in main() is not a good idea
- Solution: Move func definitions to different .c files and create a .h header file that contain their declaration
- #include the header file in the main.c
- A header file looks like a normal C file,
    - it holds the declarations of your functions

```
#include <stdio.h>
#include “addition.h”
~~int add(int a, int b);~~

void main()
{
   int c = add(1, 1);
   printf(“%d\n”, c);
}
```

Main.c

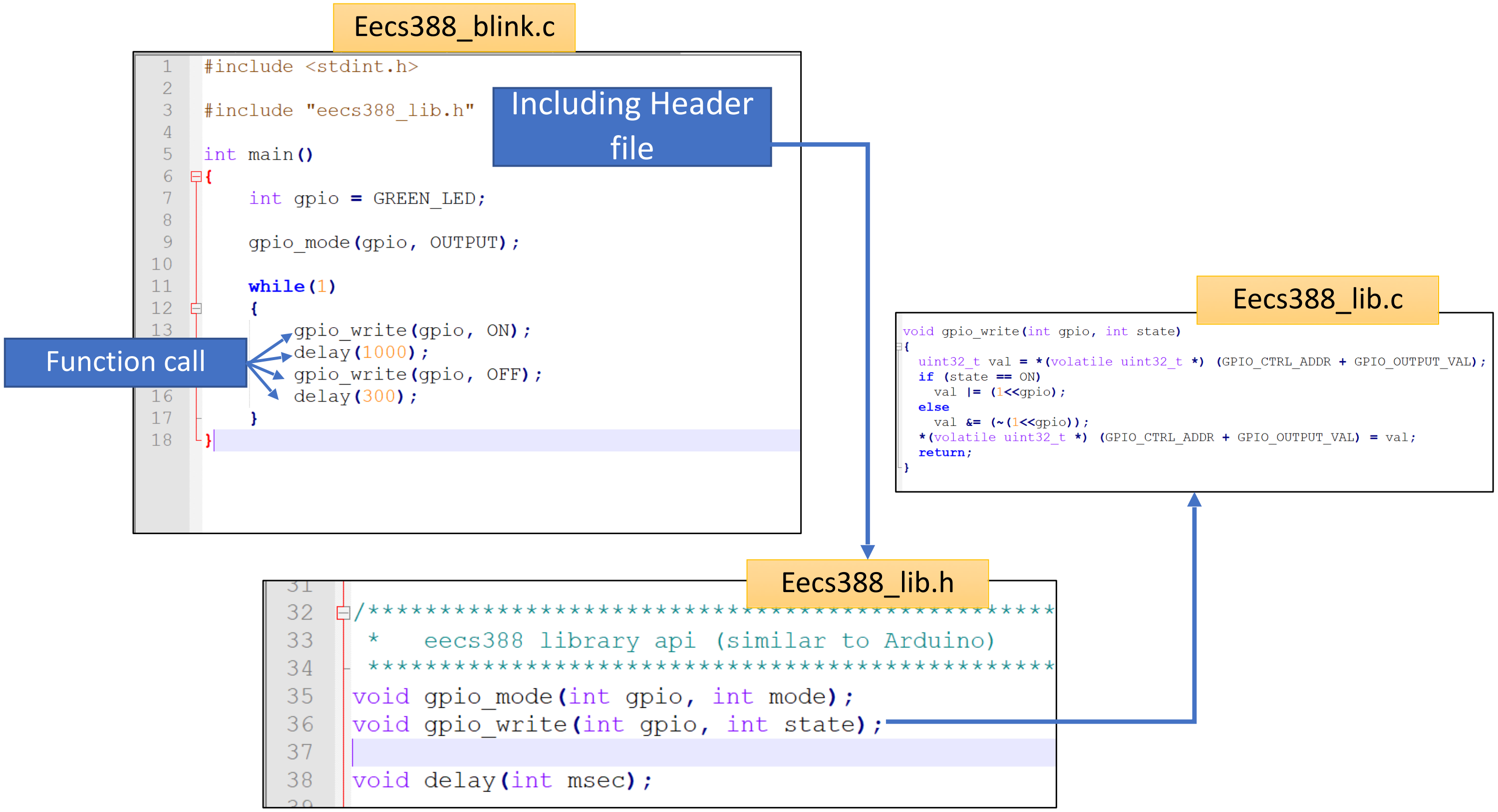
```
int add(int a, int b)
{
  return a + b;
}
```

addition.c

```
int add(int a, int b);
```

addition.h

Example from Lab



Eecs388_blink.c

```c
#include <stdint.h>

#include "eecs388_lib.h"

int main()
{
    int gpio = GREEN_LED;

    gpio_mode(gpio, OUTPUT);

    while(1)
    {
        gpio_write(gpio, ON);
        delay(1000);
        gpio_write(gpio, OFF);
        delay(300);
    }
}
```

Including Header file

Function call

Eecs388_lib.h

```c
/*****************************************************************
 *   eecs388 library api (similar to Arduino)
 *****************************************************************
void gpio_mode(int gpio, int mode);
void gpio_write(int gpio, int state);

void delay(int msec);
```

Eecs388_lib.c

```c
void gpio_write(int gpio, int state)
{
  uint32_t val = *(volatile uint32_t *) (GPIO_CTRL_ADDR + GPIO_OUTPUT_VAL);
  if (state == ON)
    val |= (1<<gpio);
  else
    val &= (~(1<<gpio));
  *(volatile uint32_t *) (GPIO_CTRL_ADDR + GPIO_OUTPUT_VAL) = val;
  return;
}
```