Late Policy: 100% up to the deadline
50% of the points up to 12 hours past

- Submit Early and often
- Dropbox

"Bana"

Big Manipulation

Ifc

ADC conv

Gate

Middle

I F / 0

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Bit manipulation

- Force/set bit: Make it 1
- Clear/reset bit: Make it 0
- Toggle bit: If it was 1 -> 0, 0 -> 1

Bitwise operators:
- OR
- AND
- XOR

Test
- Is a certain bit 1 or 0?

Example:
- If (Bump fewer) 0x0BF

Set a bit
- 1 in position I want to force
- Mask

Example:
- Clear a bit & (~mask)
- Toggle a bit ^ mask
\[
\text{TEST a bit} \quad 0100 \quad 0111 \\
\underline{0000 \ 0100}
\]

(var d work) = 0 if all bit was zero.
mark if all bit was one.

\[
\text{RHTS} 
\]

\[
\text{SHFT} \quad << \\
\text{RHSQ} \quad (1 << 3)
\]

(var num)

\[
\begin{align*}
& 40 & 18 & \text{which é€œ} \\
& 10 & 10 & \text{fit é€œ}
\end{align*}
\]

\[
\text{TEMP} = (\text{le} << 2); \\
\text{TEMP} = (\text{le} >> 2);
\]

\[
\text{unsigned int} \quad \text{temp} = 0; \\
\quad \text{temp} = (\text{le} << 2); \\
\quad \text{temp} = (\text{le} >> 2);
\]

\[
\quad \text{temp} = (\text{le} >> 2); \\
\quad \text{temp} = (\text{le} << 2); \\
\quad \text{temp} = (\text{le} << 2);
\]

\[
\rightarrow (x^2 \times 8 \times (x-1))
\]
**while Loop**

Syntax

```
while (expression) statement
```

- If `expression` is true, `statement` will be executed and then `expression` will be re-evaluated to determine whether or not to execute `statement` again.

- It is possible that `statement` will never execute if `expression` is false when it is first evaluated.
**while Loop**

**Syntax**

```
while (expression) statement
```

**Flow Diagram**

- **START**
- **expression?**
  - TRUE
  - statement
  - FALSE
  - END

**Example**

```
int i = 0;  // Loop counter initialized outside of loop
while (i < 5)  // Condition checked at start of loop iterations
{
    printf("Loop iteration %d\n", i++);  // Loop counter incremented manually inside loop
}
```

**Expected Output:**

- Loop iteration 0
- Loop iteration 1
- Loop iteration 2
- Loop iteration 3
- Loop iteration 4
**while Loop**

- The *expression* must always be there, unlike with a *for* loop
- *while* is used more often than *for* when implementing an infinite loop, though it is only a matter of personal taste
- Frequently used for main loop of program

**Note**

Infinite Loops

A `while` loop with `expression = 1` will execute indefinitely (can leave loop via `break` statement)

```c
while (1) {
    // do something
    ... // for even
}
```

**do-while Loop**

**Syntax**

```c
do statement while (expression);
```

- *statement* is executed and then
  *expression* is evaluated to determine whether or not to execute *statement* again
- *statement* will always execute at least once, even if the expression is false when the loop starts
**do-while Loop**

**Syntax**

```
do statement while (expression);
```

**Flow Diagram**

START

```
statement
```

expression?

TRUE

FALSE

END

---

**Example (Code Fragment)**

```c
int i = 0;  // Loop counter initialized outside of loop
do {
    printf("Loop iteration %d\n", i++);
} while (i < 5);  // Condition checked at end of loop iterations
```

**Expected Output:**

- Loop iteration 0
- Loop iteration 1
- Loop iteration 2
- Loop iteration 3
- Loop iteration 4
**break Statement**

**Syntax**

```
break;
```

- Causes immediate termination of a loop even if the exit condition hasn't been met.
- Exits from a `switch` statement so that execution doesn't fall through to next `case` clause.

---

**Flow Diagram Within a `while` Loop**

```
START

expression?

TRUE

statement

break

FALSE

break

statement

END
```

---

**CMPE-013/L: “C” Programming**

Gabriel Hugh Elkaim – Spring 2013
**break Statement**

**Example**

```c
int i = 0;

while (i < 10)
{
    i++;
    if (i == 5) break;
    printf("Loop iteration %d\n", i);
}
```

Expected Output:
- Loop iteration 1
- Loop iteration 2
- Loop iteration 3
- Loop iteration 4

Exit from the loop when `i = 5`. Iteration 6-9 will not be executed.

---

**continue Statement**

**Syntax**

```c
continue;
```

- Causes program to jump back to the beginning of a loop without completing the current iteration
**continue Statement**

Flow Diagram Within a `while` Loop

```
continue;
```

Example (Code Fragment)

```c
int i = 0;

while (i < 6)
{
    i++;
    if (i == 2) continue;
    printf("Loop iteration %d\n", i);
}
```

Expected Output:

- Loop iteration 1
- Loop iteration 3
- Loop iteration 4
- Loop iteration 5

Iteration 2 does not print.
Lab Exercise 7

Loops

Exercise 07

Loops

• Open the lab Project:

On the class website
Examples -> Lab7.zip

Open MPLAB®X and select Open Project Icon (Ctrl + Shift + O)
Open the Project listed above.

If you already have a project open in MPLAB X, close it by “right clicking” on the open project and selecting “Close”
Exercise 07

Loops

Solution: Step 1

```c
/*###########################################################################
# STEP 1: Create a for loop to iterate the block of code below. The loop
# should do the following:
#   * Initialize counter1 to 1
#   * Loop as long as counter1 is less than 5
#   * Increment counter1 on each pass of the loop
# (HINT: for(init; test; action))
###########################################################################*/

intVariable1 *= counter1;
printf("FOR: intVariable1 = %d, counter1 = %d\n", intVariable1, counter1);
```

//end of for loop block
Exercise 07
Loops

Solution: Step 2

```c
/*###########################################################################
# STEP 2: Create a while loop to iterate the block of code below. The loop
# should run until charVariable1 is 0.
###########################################################################*/

//Loop as long as charVariable1 is not 0
while( charVariable1 != 0)
{
    charVariable1--;  
    charVariable2 += 5;  
    printf("WHILE: charVariable1 = %d, charVariable2 = %d\n", charVariable1, charVariable2);
}

//end of while loop block
```

Solution: Step 3

```c
/*###########################################################################
# STEP 3: Create a do...while loop to iterate the block of code below. The loop
# should run until counter1 is greater than 100
###########################################################################*/

do  
{  
    counter1 += 5;  
    counter2 = counter1 * 3;  
    printf("DO: counter1 = %d, counter2 = %d\n", counter1, counter2);
}  
while(counter1 <= 100);  
```

//end of do...while block
Exercise 07
Loops

Solution: Step 1

/*###########################################################################
# STEP 1: Create a for loop to iterate the block of code below. The loop
# should do the following:
# * Initialize counter1 to 1
# * Loop as long as counter1 is less than 5
# * Increment counter1 on each pass of the loop
# (HINT: for(init; test; action))
###########################################################################*/

//Write the opening line of the for loop
for( counter1 = 1 ; counter1 < 5 ; counter1++)
{
    intVariable1 *= counter1;
    printf("FOR: intVariable1 = %d, counter1 = %d
", intVariable1, counter1);
}
//end of for loop block

Solution: Step 2

/*###########################################################################
# STEP 2: Create a while loop to iterate the block of code below. The loop
# should run until charVariable1 is 0.
# Loop as long as charVariable1 is not 0
###########################################################################*/

//Loop as long as charVariable1 is not 0
while( charVariable1 != 0)
{
    charVariable1--;
    charVariable2 += 5;
    printf("WHILE: charVariable1 = %d, charVariable2 = %d\n", charVariable1, charVariable2);
}
//end of while loop block
Exercise 07
Loops

Solution: Step 3

```c
/*###########################################################################
# STEP 3: Create a do...while loop to iterate the block of code below.
#       The loop should run until counter1 is greater than 100
############################################################################*/
do
    //Write opening line of do loop
    counter1 += 5;
    counter2 = counter1 * 3;
    printf("DO: counter1 = %d, counter2 = %d\n", counter1, counter2);
    while(counter1 <= 100);
    //Write closing line of loop - test counter1
//end of do...while block
```

Exercise 07
Conclusions

- C Provides three basic looping structures
  - for – checks loop condition at top, automatically executes iterator at bottom
  - while – checks loop condition at top, you must create iterator if needed
  - do...while – checks loop condition at bottom, you must create iterator if needed
Questions?
Operators

Gabriel Hugh Elkaim

Spring 2012

Definition

An arithmetic expression is an expression that contains one or more operands and arithmetic operators.

- Operands may be variables, constants or functions that return a value
  - A microcontroller register is usually treated as a variable
- There are 9 arithmetic operators that may be used
  - Binary Operators: +, -, *, /, %
  - Unary Operators: +, -, ++, --
Operators

Arithmetic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>x * y</td>
<td>Product of x and y</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>x / y</td>
<td>Quotient of x and y</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
<td>x % y</td>
<td>Remainder of x divided by y</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>x + y</td>
<td>Sum of x and y</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>x - y</td>
<td>Difference of x and y</td>
</tr>
<tr>
<td>+ (unary)</td>
<td>Positive</td>
<td>+x</td>
<td>Value of x</td>
</tr>
<tr>
<td>- (unary)</td>
<td>Negative</td>
<td>-x</td>
<td>Negative value of x</td>
</tr>
</tbody>
</table>

NOTE - An int divided by an int returns an int:

10/3 = 3
Use modulo to get the remainder:

10%3 = 1

Operators

Division Operator

• If both operands are an integer type, the result will be an integer type (int, char)
• If one or both of the operands is a floating point type, the result will be a floating point type (float, double)

Example: Integer Divide

```c
int a = 10;
int b = 4;
float c;  // (Q∩I)
c = a / b;
```

c = 2.000000 ✗

Because: int / int ➞ int

Example: Floating Point Divide

```c
int a = 10;
float b = 4.0f;
float c;
c = a / b;
```

c = 2.500000 ✓

Because: float / int ➞ float
Operators
Implicit Type Conversion

• In many expressions, the type of one operand will be temporarily "promoted" to the larger type of the other operand

Example

```c
int x = 10;
float y = 2.0, z;
z = x * y; // x promoted to float
```

• A smaller data type will be promoted to the largest type in the expression for the duration of the operation

Operators
Implicit Arithmetic Type Conversion Hierarchy

- long double
- double
- float
- unsigned long long
- long long
- unsigned long
- long
- unsigned int
- int
- unsigned short
- short
- unsigned char
- char
Operators

Arithmetic Expression Implicit Type Conversion

• Example implicit type conversions

Assume \( x \) is defined as:
\[
\text{short } x = -5;
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Implicit Type Conversion</th>
<th>Expression’s Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-x)</td>
<td>x is promoted to int</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>(x \times -2L)</td>
<td>x is promoted to long because (-2L) is a long</td>
<td>long</td>
<td>10</td>
</tr>
<tr>
<td>(8/x)</td>
<td>x is promoted to int</td>
<td>int</td>
<td>-1</td>
</tr>
<tr>
<td>(8%x)</td>
<td>x is promoted to int</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>(8.0/x)</td>
<td>x is promoted to double because (8.0) is a double</td>
<td>double</td>
<td>-1.6</td>
</tr>
</tbody>
</table>

Operators

Applications of the Modulus Operator (%)

• Truncation: \( x \% 2^n \) where \( n \) is the desired word width (e.g. 8 for 8 bits: \( x \% 256 \))
  - Returns the value of just the lower \( n \)-bits of \( x \)
• Can be used to break apart a number in any base into its individual digits

Example

```c
#define MAX_DIGITS 6
long number = 123456;
int i, radix = 10; char digits[MAX_DIGITS];
for (i = 0; i < MAX_DIGITS; i++)
{
    if (number == 0) break;
    digits[i] = (char)(number % radix);
    number /= radix;
}
```
Operators
Arithmetic: Increment and Decrement

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Increment</td>
<td>x++</td>
<td>Use x then increment x by 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>++x</td>
<td>Increment x by 1, then use x</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td>x--</td>
<td>Use x then decrement x by 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--x</td>
<td>Decrement x by 1, then use x</td>
</tr>
</tbody>
</table>

**Postfix Example**

```c
x = 5;
y = (x++) + 5;
// y = 10
// x = 6
```

**Prefix Example**

```c
x = 5;
y = (++x) + 5;
// y = 11
// x = 6
```
Operators

Definition

An assignment statement is a statement that assigns a value to a variable.

- Two types of assignment statements
  - Simple assignment
    \[ \text{variable} = \text{expression}; \]
    The expression is evaluated and the result is assigned to the variable
  - Compound assignment
    \[ \text{variable} = \text{variable op expression}; \]
    The variable appears on both sides of the =

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
<td>( x = y )</td>
<td>Assign ( x ) the value of ( y )</td>
</tr>
<tr>
<td>+=</td>
<td></td>
<td>( x += y )</td>
<td>( x = x + y )</td>
</tr>
<tr>
<td>-=</td>
<td></td>
<td>( x -= y )</td>
<td>( x = x - y )</td>
</tr>
<tr>
<td>*=</td>
<td></td>
<td>( x *= y )</td>
<td>( x = x * y )</td>
</tr>
<tr>
<td>/=</td>
<td></td>
<td>( x /= y )</td>
<td>( x = x / y )</td>
</tr>
<tr>
<td>%=</td>
<td></td>
<td>( x %= y )</td>
<td>( x = x % y )</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Assignment</td>
<td>( x &amp;= y )</td>
<td>( x = x &amp; y )</td>
</tr>
<tr>
<td>^=</td>
<td>Assignment</td>
<td>( x ^= y )</td>
<td>( x = x ^ y )</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td></td>
<td>( x</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td></td>
<td>( x &lt;&lt;= y )</td>
<td>( x = x &lt;&lt; y )</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td></td>
<td>( x &gt;&gt;= y )</td>
<td>( x = x &gt;&gt; y )</td>
</tr>
</tbody>
</table>
Operators
Compound Assignment

• Statements with the same variable on each side of the equals sign:

Example

```c
x = x + y;
```

This operation may be thought of as: The new value of `x` will be set equal to the current value of `x` plus the value of `y`.

• May use the shortcut assignment operators (compound assignment):

Example

```c
x += y;    //Increment x by the value y
```

Operators
Compound Assignment

Example

```c
int x = 2;    //Initial value of x is 2
x *= 5;    //x = x * 5
```

**Before statement is executed:** `x = 2`

**After statement is executed:** `x = 10`

```c
x *= 5;
```

Is equivalent to:

```c
x = (x * 5);
```

Evaluate right side first:

```c
x = (2 * 5);
```

Assign result to `x`:

```c
x = 10;
```
Operators

Relational if \( x = y \) Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (FALSE = 0, TRUE # 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>( x &lt; y )</td>
<td>1 if ( x ) less than ( y ), else 0</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>( x \leq y )</td>
<td>1 if ( x ) less than or equal to ( y ), else 0</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>( x &gt; y )</td>
<td>1 if ( x ) greater than ( y ), else 0</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>( x \geq y )</td>
<td>1 if ( x ) greater than or equal to ( y ), else 0</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>( x == y )</td>
<td>1 if ( x ) equal to ( y ), else 0</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>( x \neq y )</td>
<td>1 if ( x ) not equal to ( y ), else 0</td>
</tr>
</tbody>
</table>

In conditional expressions, any non-zero value is interpreted as TRUE. A value of 0 is always FALSE.

Operators

Difference Between = and ==

Be careful not to confuse = and ==. They are not interchangeable!

- = is the assignment operator
  \( x = 5 \) assigns the value 5 to the variable \( x \)
- == is the 'equals to' relational operator
  \( x == 5 \) tests whether the value of \( x \) is 5

```c
if (x == 5)
{
    do if value of x is 5
}
```
Operators

Difference Between = and ==

- What happens when the following code is executed?

```c
void main(void)
{
    int x = 2;         //Initialize x
    if (x = 5)          //If x is 5,…
    {
        printf("Hi!");   //…display "Hi!"
    }
}
```

Operators

Logical

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (FALSE = 0, TRUE ≠ 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>x &amp;&amp; y</td>
<td>1 if both x ≠ 0 and y ≠ 0, else 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>!x</td>
<td>1 if x = 0, else 0</td>
</tr>
</tbody>
</table>

In conditional expressions, any non-zero value is interpreted as TRUE. A value of 0 is always FALSE.
# Operators

## Bitwise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (for each bit position)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>x &amp; y</td>
<td>1, if 1 in both x and y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0, if 0 in x or y or both</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0, if 0 in both x and y</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>x ^ y</td>
<td>1, if 1 in x or y but not both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0, if 0 or 1 in both x and y</td>
</tr>
<tr>
<td>~</td>
<td>Bitwise NOT</td>
<td>~x</td>
<td>1, if 0 in x</td>
</tr>
<tr>
<td>(One's Complement)</td>
<td></td>
<td></td>
<td>0, if 1 in x</td>
</tr>
</tbody>
</table>

- The operation is carried out on each bit of the first operand with each corresponding bit of the second operand

## Difference Between & and &&

- & is the bitwise AND operator
  
  \[
  0b1010 \& 0b1101 \rightarrow 0b1000
  \]

- && is the logical AND operator
  
  \[
  0b1010 \&\& 0b1101 \rightarrow 0b0001 \text{ (TRUE)}
  \]

  \[
  <\text{Non-Zero Value}> \&\& <\text{Non-Zero Value}> \rightarrow 1 \text{ (TRUE)}
  \]

  ```c
  if (x \&\& y) {
    do if x and y are both TRUE (non-zero)
  }
  ```

- Be careful not to confuse & and &&. They are not interchangeable!
Operators
Difference Between & and &&

• What happens when each of these code fragments are executed?

Example 1 – Using A Bitwise AND Operator

```c
char x = 0b1010;
char y = 0b0101;
if (x & y) printf("Hi!");
```

Example 2 – Using A Logical AND Operator

```c
char x = 0b1010;
char y = 0b0101;
if (x && y) printf("Hi!");
```

Operators
Logical Operators and Short Circuit Evaluation

• The evaluation of expressions in a logical operation stops as soon as a TRUE or FALSE result is known

Example

If we have two expressions being tested in a logical AND operation:

`expr1 && expr2`

The expressions are evaluated from left to right. If `expr1` is 0 (FALSE), then `expr2` would not be evaluated at all since the overall result is already known to be false.

<table>
<thead>
<tr>
<th>expr1</th>
<th>expr2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X(0)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X(1)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

`expr2` is not evaluated in the first two cases since its value is not relevant to the result.
Operators

Logical Operators and Short Circuit Evaluation

• The danger of short circuit evaluation

Example

```c
if (!(z = x + y) && (c = a + b))
{
    z += 5;
    c += 10; // Initial value of c may not be correct
}
```

It is perfectly legal in C to logically compare two assignment expressions in this way, though it is not usually good programming practice. A similar problem exists when using function calls in logical operations, which is a very common practice. The second function may never be evaluated.

Operators

Shift

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Shift Left</td>
<td><code>x &lt;&lt; y</code></td>
<td>Shift x by y bits to the left</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Shift Right</td>
<td><code>x &gt;&gt; y</code></td>
<td>Shift x by y bits to the right</td>
</tr>
</tbody>
</table>

Shift Left Example:

- `x = 5;`  \[
\text{\(x = 0b00000101 = 5\)}
\]
- `y = x << 2;` \[
\text{\(y = 0b000010100 = 20\)}
\]

• In both shift left and shift right, the bits that are shifted out are lost
• For shift left, 0's are shifted in (Zero Fill)
Operators
Shift – Special Cases

• Logical Shift Right (Zero Fill)

If \( x \) is **UNSIGNED** (unsigned char in this case):
\[
x = 250; \quad \text{// } x = 0b11111010 = 250
\]
\[
y = x \gg 2; \quad \text{// } y = 0b00111110 = 62
\]

• Arithmetic Shift Right (Sign Extend)

If \( x \) is **SIGNED** (char in this case):
\[
x = -6; \quad \text{// } x = 0b11111010 = -6
\]
\[
y = x \gg 2; \quad \text{// } y = 0b11111110 = -2
\]

Operators
Power of 2 Integer Divide vs. Shift Right

• If you are dividing by a power of 2, it will usually be more efficient to use a right shift instead

\[
y = x / 2^n \quad \rightarrow \quad y = x \gg n
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}\]

\( 10_{10} \) \quad \text{Right Shift} \quad 5_{10}

• Works for integers or fixed point values
Operators
Power of 2 Integer Divide vs. Shift in MPLAB® C30

Example: Divide by 2

```
int x = 20;
int y;
y = x / 2;
```

Example: Right Shift by 1

```
int x = 20;
int y;
y = x >> 1;
```

10:               y = x / 2;
0028B 804000    mov.w 0x0800,0x0000
0028A 20022     mov.w #0x2,0x0004
0028C 090011    repeat #17
0028E D80002    div.sw 0x0000,0x0004
00290 884010    mov.w 0x0a000,0x0802

y = 10

9:               y = x >> 1;
00282 804000    mov.w 0x0800,0x0000
00284 DE8042    asr 0x0000,#1,0x0000
00286 884010    mov.w 0x0a000,0x0802

y = 10

---

Operators
Power of 2 Integer Divide vs. Shift in MPLAB® C18

Example: Divide by 2

```
int x = 20;
int y;
y = x / 2;
```

Example: Right Shift by 1

```
int x = 20;
int y;
y = x >> 1;
```

10:               y = x / 2;
0132 C08C     MOVFF 0x8c, 0x8a
0134 F08A     NOP
0136 C08D     MOVFF 0x8d, 0x8b
0138 F08B     NOP
013A 0802     MOV.0 0x2
013C 6800     MOVWF 0xd, ACCESS
013E 6A0E     CLR.F 0xe, ACCESS
0140 C08A     MOVFF 0x8a, 0x8
0142 F008     NOP
0144 C08B     MOVFF 0x8b, 0x9
0146 F009     NOP
0148 EC6B     CALL 0x6d, 0
014A F000     NOP
014C C008     MOVFF 0x8, 0x8a
014E F08A     NOP
0150 C009     MOVFF 0x9, 0x8b
0152 F08B     NOP

9:               y = x >> 1;
0122 C0BC     MOVFF 0x8c, 0x8a
0124 F08A     NOP
0126 C0BD     MOVFF 0x8d, 0x8b
0128 F08B     NOP
012A 0100     MOVLR 0
012C 9008     BCF 0xf68, 0, ACCESS
012E 338B     RRCF 0x8b, F, BANKED
0130 338A     RRCF 0x8a, F, BANKED

16-Bit Shift on 8-Bit Architecture
# Operators

## Memory Addressing

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Address of</td>
<td>x</td>
<td>Pointer to x</td>
</tr>
<tr>
<td>*</td>
<td>Indirection</td>
<td>p</td>
<td>The object or function that p points to</td>
</tr>
<tr>
<td>[]</td>
<td>Subscripting</td>
<td>x[y]</td>
<td>The y&lt;sup&gt;th&lt;/sup&gt; element of array x</td>
</tr>
<tr>
<td>.</td>
<td>Struct / Union Member</td>
<td>x.y</td>
<td>The member named y in the structure or union x</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Struct / Union Member by Reference</td>
<td>p-&gt;y</td>
<td>The member named y in the structure or union that p points to</td>
</tr>
</tbody>
</table>

These operators will be discussed later in the sections on arrays, pointers, structures, and unions. They are included here for reference and completeness.

## Other

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Function Call</td>
<td>foo(x)</td>
<td>Passes control to the function with the specified arguments</td>
</tr>
<tr>
<td>sizeof</td>
<td>Size of an object or type in bytes</td>
<td>sizeof x</td>
<td>The number of bytes x occupies in memory</td>
</tr>
<tr>
<td>(type)</td>
<td>Explicit type cast</td>
<td>(short) x</td>
<td>Converts the value of x to the specified type</td>
</tr>
<tr>
<td>?:</td>
<td>Conditional expression</td>
<td>x ? y : z</td>
<td>The value of y if x is true, else value of z</td>
</tr>
<tr>
<td>,</td>
<td>Sequential evaluation</td>
<td>x, y</td>
<td>Evaluates x then y, else result is value of y</td>
</tr>
</tbody>
</table>