Lecture 6: C Programming I

ME/AE 6705

Introduction to Mechatronics

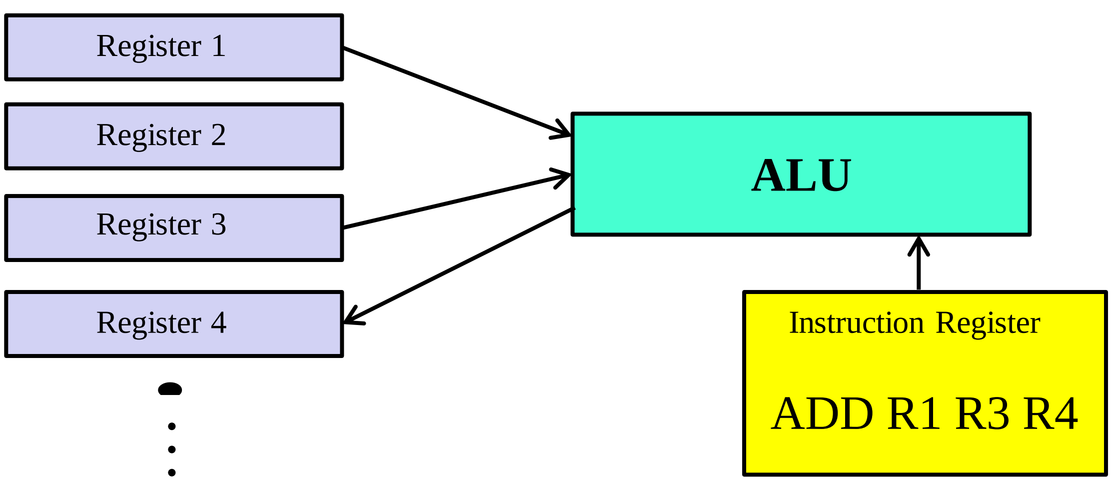
Dr. Jonathan Rogers

Lesson Objectives

- Understand hierarchy of programming languages/methods
  - Machine language, assembly, C/C++
- Understand basic structure of C program
- Be comfortable with use of C data types
- Be able to decompose program into functions and use appropriate flow control
- Demonstrate concepts through multiple examples

Fundamentals of Processor Execution

- Arithmetic & Logic Unit of microprocessor is responsible for taking data and performing some (simple) operation
  - More complex operations built from series of simple ones
  - ALU made up of series of logic gates



Configuration Registers

- Registers are particular 32-bit memory locations
- Processor uses registers for configuration purposes
  - These registers are fixed memory locations that serve a particular purpose
  - Example: Flash size register

Other Important Registers:

- Instruction register
- Data registers on ALU
- ADC configuration registers
- PWM configuration registers
- …

This register reflects the size of flash main memory available on the device.

Figure 6-6. SYS_FLASH_SIZE Register

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

Table 6-3. SYS_FLASH_SIZE Register Description

BIT	FIELD	TYPE	RESET	DESCRIPTION
31-0	SIZE	R	Variable	Indicates the size (in bytes) of the flash main memory on the device. This is divided equally between the two banks.

Data Registers

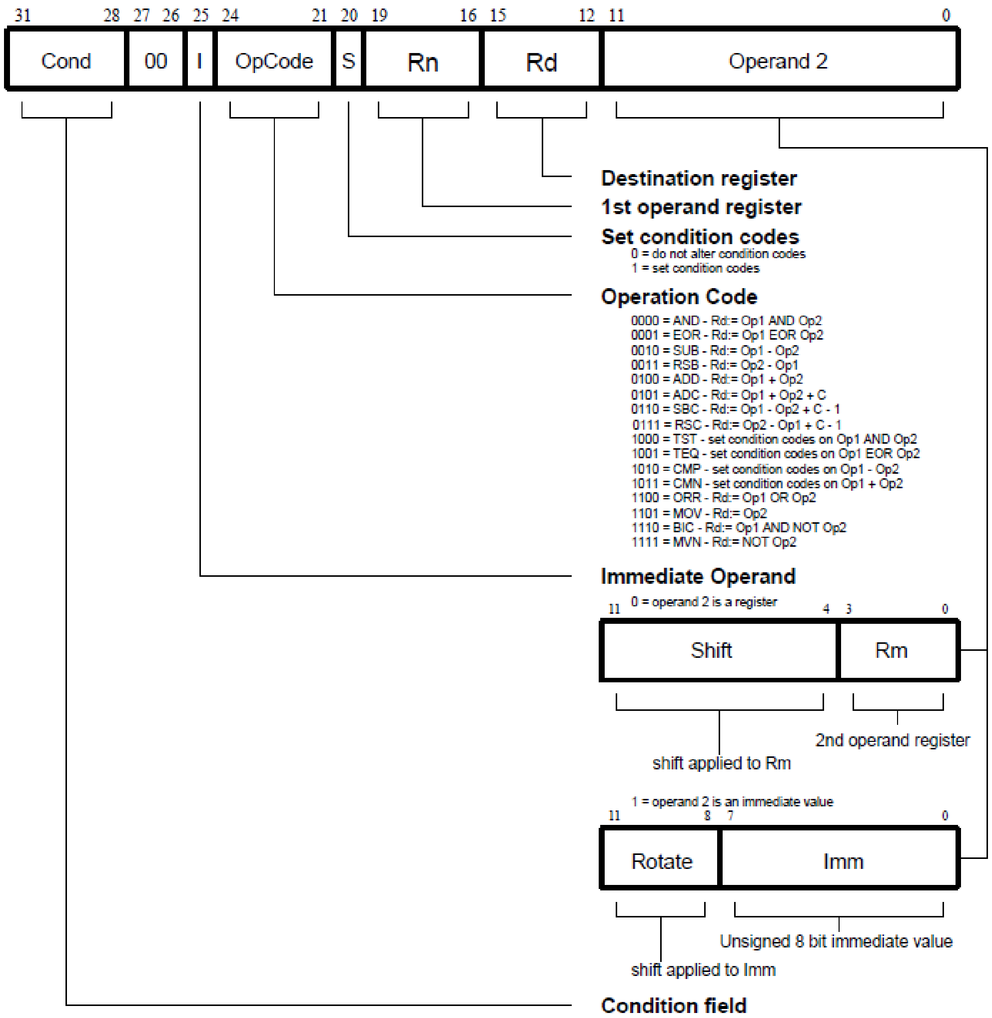
- Processor also contains registers used to hold data during processing
- These registers are the size of one WORD
  - 32-bit on 32-bit processors
  - 8-bit on 8-bit processors
- These registers hold intermediate data
  - Either taken from memory to be operated on
  - Or staged in registers before being shipped to memory

Instruction Register

- Holds the instruction that ALU is executing
- An *instruction* is a 32-bit value that determines what operation is to be performed
- An *instruction set* is defined by the processor manufacturer
  - ALU can only perform operations in its instruction set
- Instructions also can have arguments associated with them
  - Tell ALU pieces of data on which instruction should be performed

MSPM0 uses ARM Cortex Instruction Set.

MSPM0 Instructions



- Instructions are 32-bit binary values
- Contain fields specifying operation to perform, register addresses with relevant data, and configuration flags
- Instruction format located in ARM Cortex datasheet
- Instruction format to the left applies to all “data processing” instructions (add, subtract, etc)

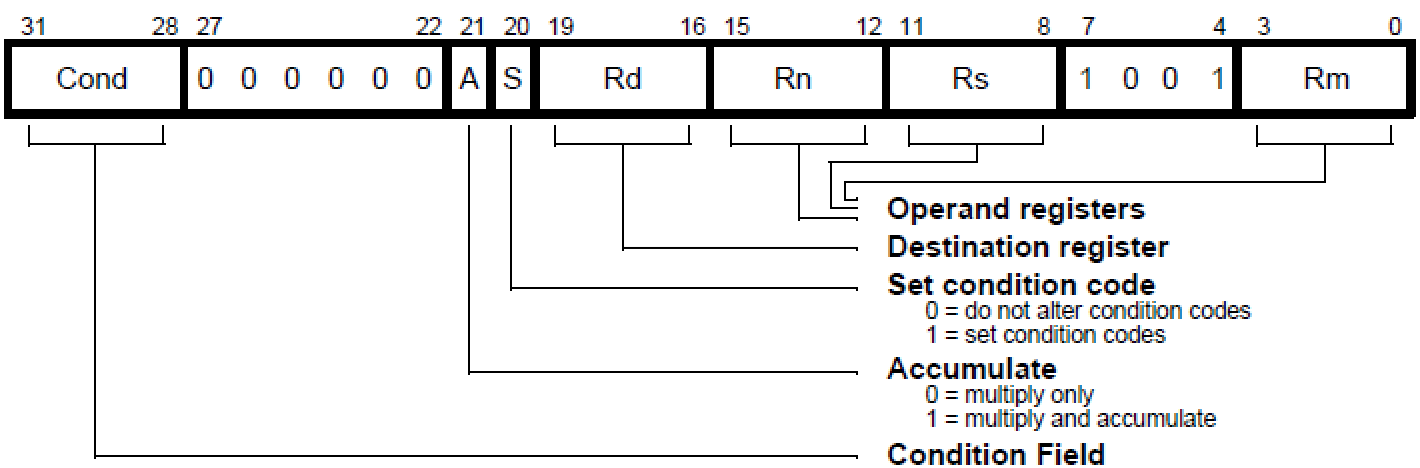
MSPM0 Instructions

31–28	27–26	25	24–21	20	19–16	15–12	11–0
Cond	00	I	OpCode	S	Rn	Rd	Operand 2

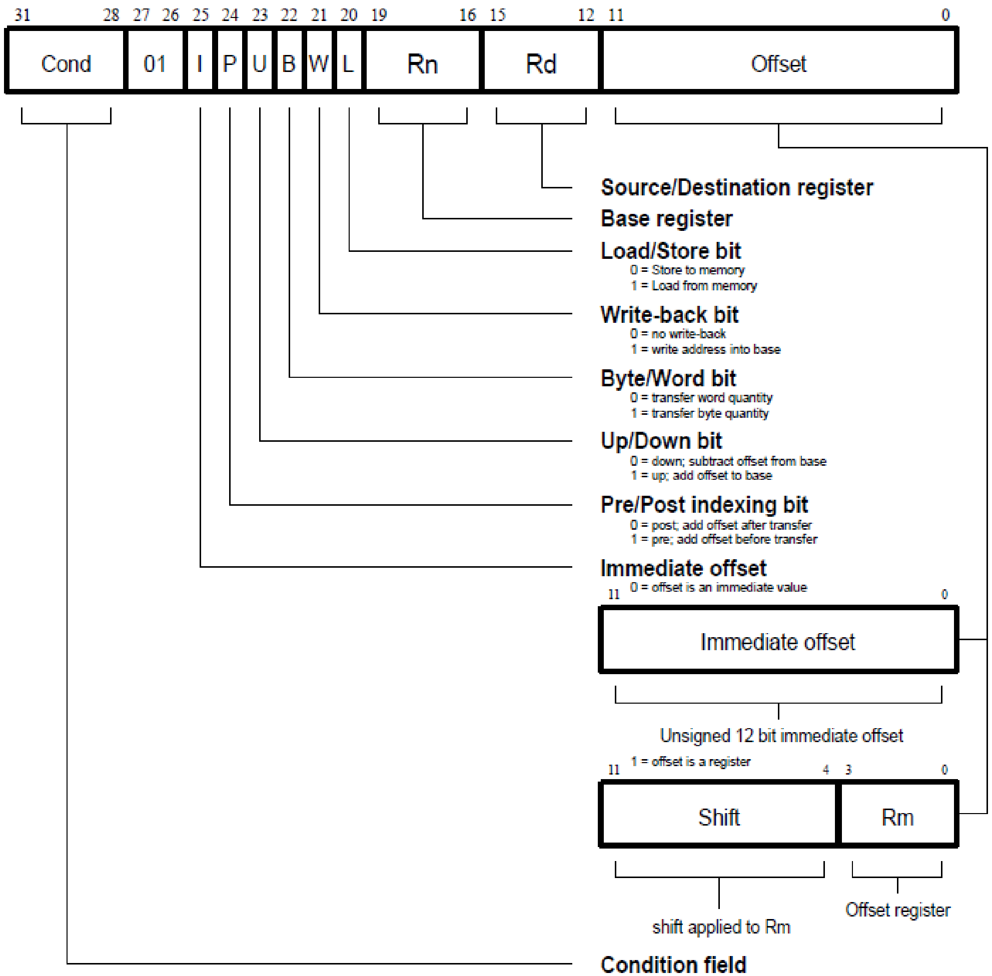
- Example: Add instruction (ADD)
  – OpCode 0100
  – Rd = destination (result) register
  – Rn = register containing first argument
  – Operand2 = register containing second argument
    - *Or can contain second argument itself (depends on I bit)*
  – Executes Rd = Rn + Operand2
  – Takes 1 clock cycle for ALU to execute

MSPM0 Instructions

- Example: Multiply instruction (MUL)
  - Multiplies Rd = Rm*Rs (Rn field ignored)
  - May take several clock cycles to complete depending on instruction configuration



MSPM0 Instructions



- Load/store instruction
- Like ADD instruction, LDR/STR instruction is configured by the bits within the 32-bit instruction
- For instance, bit L is used to determine whether it is a Store instruction (1) or Load instruction (0)
- Store means send data from Rn to Rd
- Load means send data from Rd to Rn
- Takes 3 clock cycles to execute

Writing a Program

- Theoretically you can write a program for MSPM0 completely in machine language by typing in the hex values of each instruction sequentially
  - Not feasible
- Alternatively, you can write a program in assembly language by typing in each instruction and associated arguments in assembly format
  - Assembly language can then be converted to machine language by a computer program (assembler)

Assembly Language

- Assembly language is low level programming language consisting of individual instructions
  - Essentially abstracts "binary" instructions above into more readable language
- Examples of assembly instructions for MSPM0

```
ADD     R2,  R1,  R3    ; R2 = R1+R3
ADDS   R4,  R4,  #100  ; R4 = R4 + 100
```

Add instructions

```
AND     R9,  R2, #0xFF00 ; R9 = R2&0xFF00
```

And instruction

```
MUL     R10,  R2, R5    ; R10 = R2*R5
```

Multiply instruction

```
STR     R2, [R9, R12]   ; Store 32-bit value in R2 into
                          memory address R9+R12
```

Store instruction

Assembly Language

- ARM Cortex M processor has over 150 instructions

Operation		§	Assembler	S updates	Action	Notes
Add	Add		`ADD{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2	N
	with carry		`ADC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2 + Carry	N
	wide	T2	`ADD Rd, Rn, #<imm12>`		Rd := Rn + imm12, imm12 range 0-4095	T, P
	saturating {doubled}	5E	`Q{D}ADD Rd, Rm, Rn`		Rd := SAT(Rm + Rn) doubled: Rd := SAT(Rm + SAT(Rn * 2))	Q
Address	Form PC-relative address		`ADR Rd, <label>`		Rd := <label>, for <label> range from current instruction see Note L	N, L
Subtract	Subtract		`SUB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2	N
	with carry		`SBC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2 – NOT(Carry)	N
	wide	T2	`SUB Rd, Rn, #<imm12>`	N Z C V	Rd := Rn – imm12, imm12 range 0-4095	T, P
	reverse subtract		`RSB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn	N
	reverse subtract with carry		`RSC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn – NOT(Carry)	A
	saturating {doubled}	5E	`Q{D}SUB Rd, Rm, Rn`		Rd := SAT(Rm – Rn) doubled: Rd := SAT(Rm – SAT(Rn * 2))	Q
	Exception return without stack		`SUBS PC, LR, #<imm8>`		PC = LR – imm8, CPSR = SPSR(current mode), imm8 range 0-255.	T

Branch	Branch		`B <label>`	PC := label. label is this instruction ±32MB (T2: ±16MB, T: –252 - +256B)	N, B
	with link		`BL <label>`	LR := address of next instruction, PC := label. label is this instruction ±32MB (T2: ±16MB).	
	and exchange	4T	`BX Rm`	PC := Rm. Target is Thumb if Rm[0] is 1, ARM if Rm[0] is 0.	N
	with link and exchange (1)	5T	`BLX <label>`	LR := address of next instruction, PC := label, Change instruction set. label is this instruction ±32MB (T2: ±16MB).	C
	with link and exchange (2)	5	`BLX Rm`	LR := address of next instruction, PC := Rm[31:1]. Change to Thumb if Rm[0] is 1, to ARM if Rm[0] is 0.	N
	and change to Jazelle state	5J	`BXJ Rm`	Change to Jazelle state if available	
	Compare, branch if (non) zero	T2	`CB{N}Z Rn,<label>`	If Rn {== or !=} 0 then PC := label. label is (this instruction + 4-130).	N T U
	Table Branch Byte	T2	`TBB [Rn, Rm]`	PC = PC + ZeroExtend(Memory(Rn + Rm, 1) << 1). Branch range 4-512. Rn can be PC.	T U
	Table Branch Halfword	T2	`TBH [Rn, Rm, LSL #1]`	PC = PC + ZeroExtend(Memory(Rn + Rm << 1, 2) << 1). Branch range 4-131072. Rn can be PC.	T U

Move data	Move		`MOV{S} Rd, <Operand2>`	N Z C	Rd := Operand2 See also Shift instructions	N
	NOT		`MVN{S} Rd, <Operand2>`	N Z C	Rd := 0xFFFFFFFF EOR Operand2	N
	top	T2	`MOVT Rd, #<imm16>`		Rd[31:16] := imm16, Rd[15:0] unaffected, imm16 range 0-65535	
	wide	T2	`MOV Rd, #<imm16>`		Rd[15:0] := imm16, Rd[31:16] = 0, imm16 range 0-65535	
	40-bit accumulator to register	XS	`MRA RdLo, RdHi, Ac`		RdLo := Ac[31:0], RdHi := Ac[39:32]	
	register to 40-bit accumulator	XS	`MAR Ac, RdLo, RdHi`		Ac[31:0] := RdLo, Ac[39:32] := RdHi	

Assembly Language

- Example ARM Cortex assembly code
  - For function my_asm

```
my_asm
; ARM Assembly language function to set LED1 bit to a value passed from C
; LED1 gets value (passed from C compiler in R0) ; LED1 is on GPIO port 1 bit 18
; See Chapter 9 in the LPC1768 User Manual
; for all of the GPIO register info and addresses
; Pinnames.h has the mbed modules pin port and bit connections ;
; Load GPIO Port 1 base address in register R1
      LDR     R1, =0x2009C020 ; 0x2009C020 = GPIO port 1 base address
; Move bit mask in register R2 for bit 18 only
     MOV.W   R2, #0x040000   ; 0x040000 = 1<<18 all "0"s with a "1" in bit 18
; value passed from C compiler code is in R0 - compare to a "0"
      CMP     R0, #0          ; value == 0 ?
; (If-Then-Else) on next two instructions using equal cond from the zero flag
     ITE EQ
; STORE if EQ - clear led 1 port bit using GPIO FIOCLR register and mask
     STREQ   R2, [R1,#0x1C]  ; if==0, clear LED1 bit
; STORE if NE - set led 1 port bit using GPIO FIOSET register and mask
     STRNE   R2, [R1,#0x18]  ; if==1, set LED1 bit
; Return to C using link register (Branch indirect using LR - a return)
     BX      LR
     END
```

Assembly Language

- Example assembly language program (not ARM):

```
_vector_add:
mov.u32 %r1, %tid.x;
mov.u32 %r2, %envreg3;
add.s32 %r3, %r1, %r2;
mov.u32 %r4, %ctaid.x;
mov.u32 %r5, %ntid.x;
mad.lo.s32 %r6, %r4, %r5, %r3;
shl.b32 %r7, %r6, 2;
ld.param.u32 %r8, [vector_add_param_1];
ld.param.u32 %r9, [vector_add_param_0];
add.s32 %r10, %r9, %r7;
add.s32 %r11, %r8, %r7;
ld.param.u32 %r12, [vector_add_param_2];
ld.global.u32 %r13, [%r10];
ld.global.u32 %r14, [%r11];
add.s32 %r15, %r14, %r13;
add.s32 %r16, %r12, %r7;
st.global.u32 [%r16], %r15;
ret;
```

Function name

Move unsigned 32-bit number

Add signed 32-bit number

Load unsigned 32-bit number

Writing a Program

- MCU’s used to be programmable only using assembly language
  - All complex functionality had to be expressed in simple instructions
- Problem: To implement complex functionality, writing assembly programs from scratch extremely time consuming
  - Programs can be very long – simple line of C code becomes 10’s of lines of assembly

Writing a Program

- For instance, previous 18-line assembly program was assembly language version of this C function:

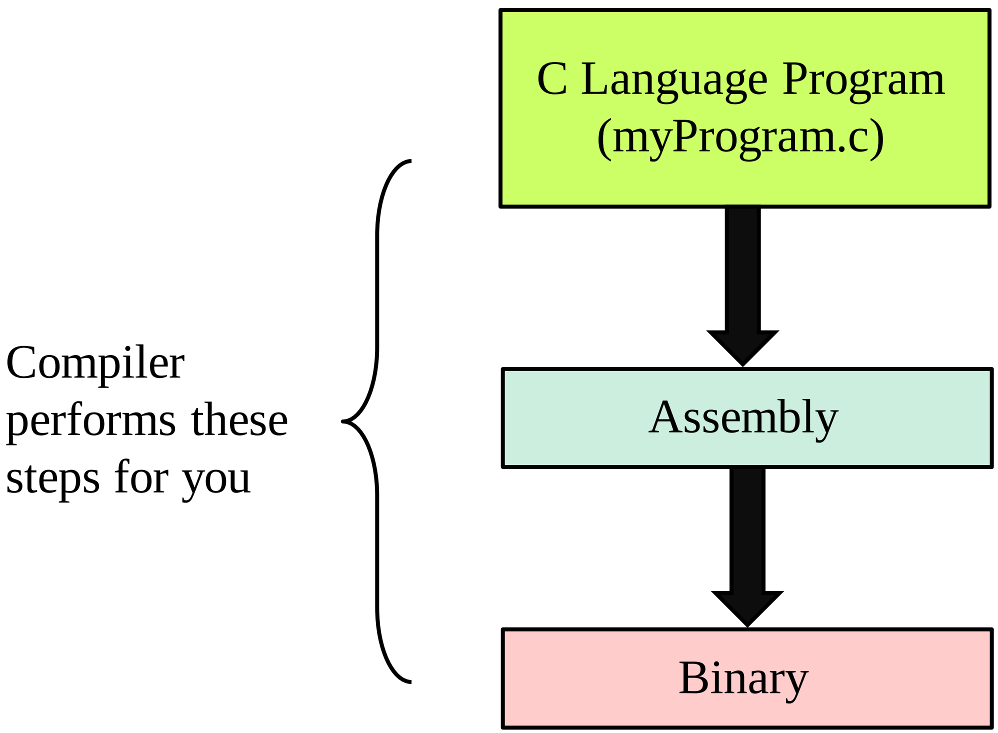
```
void vector_add(int *A, int *B, int *C) {

    // Get the index of the current element to be processed
    int i = get_global_id(0);

    // Perform addition
    C[i] = A[i] + B[i];
}
```

C Language Basics

- Thus higher level languages like C were developed to limit developer effort to reasonable level
- C language is built on top of assembly code
- Facts about C language:
  - Developed in 1970 at Bell Laboratories
    - *Replaced earlier languages such as FORTRAN*
  - Very flexible and elegant language, used extremely widely
  - C++ written on top of C language to enable object-oriented programming
  - MSPM0 supports variant of C called C99

C Language Basics

- Compiling workflow



- C Compiler is a computer program that translates C language to machine instructions (binary)
- Running program successfully results in executable file
- Since different processors use different instruction sets, processor manufacturer has to provide compiler for their device

C Language Basics

- Simple C program

```
// Simple C program

#include <stdio.h>

int main(){

         int miles, yards ;
         float kilometers ;

         miles = 26 ;
         yards = 385 ;
         kilometers = 1.069 * (miles + yards/1760.0) ;

         printf("\nA marathon is %f kilometers\n\n",kilometers) ;

         return 0 ;
}
```

C Language Basics

- Simple C program

```
// Simple C program

#include <stdio.h>

int main(){

        int miles, yards ;
        float kilometers ;

        miles = 26 ;
        yards = 385 ;
        kilometers = 1.069 * (miles + yards/1760.0) ;

        printf("\nA marathon is %f kilometers\n\n",kilometers) ;

        return 0 ;
}
```

Comment

Header file include

Main function definition

Variable declarations

Assignments

Print statement

Return statement

C Language Basics

- In C programs, data is stored in “variables”
- Variables are declared by the programmer
  - When a variable is declared, memory is allocated to store the data for that variable
- All variables must have an associated <u>data type</u>
  - In MSPM0 C, can be declared as int, float, double, etc.
  - Can also use int32_t, int16_t, uint32_t, etc.

C Language Basics

- In C programs, data is stored in “variables”
- Variables are declared by the programmer
  - When a variable is declared, memory is allocated to store the data for that variable
- All variables must have an associated <u>data type</u>
  - In MSPM0 C, can be declared as int, float, double, etc.
  - Can also use int32_t, int16_t, uint32_t, etc.
- Data type tells compiler:
  - How much memory should be allocated for this variable
  - How to interpret data in memory location (signed, unsigned, float, etc.)

C Language Basics

- All programs in C must contain a “main” function
  - This is where program execution always begins
  - Other functions can be called from main function
  - Main function can never be called by another function

```
// Simple C program
#include <stdio.h>

int main(){

        int miles, yards ;
        float kilometers ;
        ...

        return 0 ;
}
```

C Language Basics

- Assignment statements assign the value of one variable (or number) to another variable
  - = sign used for assignment
- All declarations and assignment statements in C end with a semicolon (“;”)
- To the right of = sign in assignment operations, can add any operations or function calls you like

```
kilometers = 1.069 * (miles + yards/1760.0) ;
```

C Language Basics

- Basic mathematical operations

  + add two numbers

  - subtract two numbers

  * multiply two numbers

  / divide two numbers

- Standard order of operations used when multiple operations performed on same line
  – Parentheses used to define desired order of operations

```
kilometers = 1.069 * (miles + yards/1760.0) ;
```

Printf Statements

- Recall our discussion of printf from last class
- printf prints a string of characters formatted according to our specification in code

```
float res = 15.7 ;

printf("res = %f\n",res) ;
```

→ prints: res = 15.7000

```
int32_t res = 15 ;

printf("res = %d\n",res) ;
```

→ prints: res = 15

Preprocessor

- Compiler actually composed of two programs:
  - Preprocessor: Goes through code before compilation and makes modifications
  - Compiler: Actually translates code to machine language
- Preprocessor directives:
  - All start with a "#" sign
  - Tell the preprocessor to do something prior to compilation
  - Two used often: #include and #define

#include Statements

- Oftentimes we define lots of variables (or functions) in separate files to make our code more modular and readable

```
// Simple C program
#include <stdio.h>

int main(){

        ...

        return 0 ;
}
```

#include statement says to take all code in file “stdio.h” and place it in this part of code prior to compilation

Header Files

- Most of the time, #include is used to include header files in your code
- Header files contain common functions, variable declarations, etc.
  - Always end in “.h”
- For instance, printf(…) is a function that is defined in stdio.h
  - Without including this, compiler doesn’t know what printf(…) refers to
- ARM codebase comes with many header files (math.h, stdio.h, ...) but you can write your own

#define Statements

- Another common preprocessor directive is #define
- This says to replace all instances of [first argument] in code with [second argument]
  – Allows you to change something in multiple places easily

```
#define C 325

int main(){
        int newVal = C ;
        ...
        newVal = log(C)/log(2);
        return 0 ;
}
```

During compile time, all values of C *will get populated with 325.*

Functions

- Functions allow you to decompose program into discrete modules
- Functions are declared in C using following structure:

```
type  function_name ( argument list ){

        variable definitions

        statements …

        return …
}
```

Functions

- All functions return something
  - Thus they are declared with a data type specifying what type of data they will return
- Example: Consider function “square” which computes square of two values

```
// Returns value of x^2
int square (int x){

        int ans = x*x ;

        return ans ;
}
```

Functions

- A function can return any data type you want
- If you don't want a function to return anything, you can declare it as a type <u>void</u> function
- Example: Function that prints "hello!"

```
// Prints "hello!" to screen
void print_hello(){

        printf("hello!") ;

        return ;
}
```

Functions

- When return statement is encountered, execution of the function is terminated and control goes back to environment which called function
  - Return statements may contain an expression:

```
// Returns value of x^2
int square (int x){

        return x*x ;
}
```

  - Functions of type void do not need to include return statement

```
void print_hello(){

        printf("hello!") ;
}
```

Functions

- Function arguments are sent to function by calling environment
  - These are variables that can be used by function
  - Some, all, or none of the function’s variables can be sent in as arguments

```
// Sums x, y, and then multiplies result by 2.5
float do_math (int x, int y){

        float z = (float)(x+y)
        return z*2.5 ;
}
```

Functions

- When variables are passed in as an argument, a copy of them is actually being passed
  - A variable (passed in as argument) which is changed inside function is not also changed outside

```
// Sums x, y, and then multiplies result by 2
int do_math (int x, int y){

        x = x+y
        return x*2 ;
}

int main (){

        int x = 4, y = 7 ;

        printf("x=%d\n", do_math(x,y)) ;   // Will print "x = 22"
        printf("x=%d\n", x) ;              // Will print "x = 4"
}
```

Functions

- Functions may of course be called from other functions
- Compiler compiles your code sequentially
  - If a function is called before it is defined, compiler will not know what that function call refers to

```
int main (){

        int x = 4, y = 7 ;
        printf(“x=%d\n”, do_math(x,y)) ;   // Will print “x = 22”
        printf(“x=%d\n”, x) ;              // Will print “x = 4”
}

int do_math (int x, int y){

        x = x+y
        return x*2 ;
}
```

Compiling will fail – cannot find function do_math.

Functions

- Instead, need to add <u>function prototype</u> before function is called
  - Usually all function prototypes located at top of file

```
int do_math(int x, int y) ;

int main (){

        int x = 4, y = 7 ;
        printf("x=%d\n", do_math(x,y)) ;   // Will print "x = 22"
        printf("x=%d\n", x) ;              // Will print "x = 4"
}

int do_math (int x, int y){

        x = x+y
        return x*2 ;
}
```

Function prototype contains return type, name, and argument list of function.

Main Function

- All programs require definition of a function called “main”
- main function in traditional C:
  - Takes arguments which are provided in command line
  - Always returns int value (specifying program return value)

```
// Traditional C main function
int main (int argc, char *argv[]){
        ...
        return 0 ;
}
```

- main function for MSPM0
  - Takes no arguments (void) and returns void

```
// MSPM0 main function
void main(void){
        ...
        return ;
}
```

Flow Control

- As program is executing, you oftentimes will require conditional statements, loops, or both
- C provides various flow control mechanisms:
  - if-else statement
  - while loop
  - for loop
  - switch statement
- if-else and switch statements provide conditional execution
- while and for loops provide repetitive execution

Flow Control

- Relational, equality, and logical operators

<table>
<tr><th>Type</th><th>Meaning</th><th>Symbol</th></tr>
<tr><td>Relational Operators</td><td>Less than
Greater than
Less than or equal to
Greater than or equal to</td><td><
>
<=
>=</td></tr>
<tr><td>Equality Operators</td><td>Equal to
Not equal to</td><td>==
!=</td></tr>
<tr><td>Logical Operators</td><td>Negation
Logical and
Logical or</td><td>!
&&
||</td></tr>
</table>

Flow Control

- If statement:

```
if(expression){

        ...

}
```

  - Code in {} brackets only gets executed if [expression] evaluates to true
  - Same exact functionality as in MATLAB

Flow Control

- Example:

```
if((a > 5) && (b != 7)){

        ...

}
```

  - Statements inside if statement will execute only if a is greater than 5 and b is not equal 7
  - Otherwise these statements will be skipped

Flow Control

- Else statement can be used in conjunction to define code that will be executed if if statement is not entered
- Example:

```
if((a > 5) && (b != 7)){

        c = 54 ;

}
else{
        c = 0 ;
}
```

Flow Control

- If-else statements can be grouped together to define series of alternative execution paths
- Example:

```
if(a < 5){
        c = 54 ;                    // Will execute only when a < 5
}
else if((a >=5) && (a < 14)){
        c = 55 ;                    // Will execute only when 5 <= a < 14
}
else{
        c = 0 ;                     // Will execute otherwise
}
```

Flow Control

- While loop will repeat execution of code within {} while a certain condition remains true
- Example:

```
while(i < 100){
         printf("i = %d\n", i)
         i++ ;
}
```

- While loops often used when the number of times the loop is performed is not known a priori

Flow Control

- While loops can be used to create infinite loops
  - Loops which never end

```
while(1){
          printf("I'm still running.\n")
}
```

  - In C, logical expressions return either 1 (true) or 0 (false)
    - *0 is always treated as "false"*
    - *Anything not zero is considered "true"*
- Infinite loops used often in microcontroller programming to wrap around main processing loop

Flow Control

- for loop used to execute code iteratively
  - Similar to while loop
- With for loop, you specify
  1. Variable initialization at start of loop
  2. End condition of loop
  3. Action to take every time loop is repeated
- Example:

```
for(i = 0; i < 12; i++){
        printf("i = %d\n", i) ;
}
```

Flow Control

- Example of factorial function using while loop

```
// Computes n factorial
int computeFactorial(int n){

        int factorial = 1 ;

        for(i = 1; i <= n; i++){
                factorial = factorial*i ;
        }

        return factorial ;
}
```

- Any while loop <u>can always</u> be made into a for loop

Flow Control

- Use the “break” keyword to jump out of a loop when a certain condition is met

```
while(1){
         printf(“I’m still running.\n”)
         i++ ;
         if(i > 22){
                  break;
         }
}
```

- break will only jump out of lowest level loop

```
while(1){
         for(i = 0; i < 40; i++){
         if(i > 22){
                  break;
         }
}
```

Jumps back into while loop

Flow Control

- Switch statement is generalization of if-else
  - Used when there are many possible else-if's
- Example:

```
switch (c){
        case 'a':
                a_cnt++ ;
                break ;
        case 'b':
                b_cnt++ ;
                break ;
        default:
                other_cnt++ ;
}
```

- Evaluate whether c is equal to a, b, or something else
- Execute logic after the appropriate case statement
- break terminates switch statement

Comments

- Your code should *always* be commented well
- Essential when building codes for real mechatronic systems
  - Real codes may be 1000’s of lines
  - Important if you ever want someone else to see your code
  - Should comment approximately every line
- Comments in C:
  - Any line preceded by two slashes //
  - Between /* and */ mark

Comments

- Examples

```
// Example switch statement
switch (c){
        case 'a':
                // Increment a counter
                a_cnt++ ;
                break ;
        case 'b':
                // Increment b counter
                b_cnt++ ;
                break ;
        default:
                // Increment other counter
                other_cnt++ ;
}
/* This is an example of another comment, used to
       comment out multiple lines at once.  */
```