Chapter 7

Stack and Subroutines

Updated: 3/23/15
Basic Idea

- Large programs are hard to handle
  - We can break them to smaller programs
  - They are called subroutines
- Subroutines are called from the main program
- Writing subroutines
  - When should we jump? (use CALL)
  - Where do we return to? (use RETURN)
A subroutine is a block of code that is called from different places from within a main program or other subroutines.

Saves code space in that the subroutine code does not have to be repeated in the program areas that need it;
  - Only the code for the subroutine call is repeated.

A subroutine can have
  - parameters that control its operation
  - local variables for computation.

A subroutine may pass a return value back to the caller.

Space in data memory must be reserved for parameters, local variables, and the return value.
Subroutine

Without Subroutines

instr_1
instr_2

instr_a1
instr_a2
...
instr_an

instr_3
instr_4
instr_5

instr_a1
instr_a2
...
instr_an

instr_6
instr_7

......

Replicated instruction sequence

With Subroutines

Caller

instr_1
instr_2
call sub
instr_3
instr_4
instr_5
call sub

......

Callee

instr_a1
instr_a2
...
instr_an
return

Replicated instruction sequence as a subroutine
Using Subroutines

- When using subroutines we need to know the following:
  - Where is the NEXT instruction’s address
  - How to remember the RETURN address
- Subroutines are based on MPU instructions and use STACK
Stack

- Temporary memory storage space used during the execution of a program
- Used by MPU
- Stack Pointer (SP)
  - The MPU uses a register called the stack pointer, similar to the program counter (PC), to keep track of available stack locations.
Data Storage via the Stack

- The word ‘stack’ is used because storage/retrieval of words in the stack memory area is the same as accessing items from a stack of items.

- Visualize a stack of boxes. To build a stack, you place box A, then box B, then box C
  - Notice that you only have access to the last item placed on the stack (the Top of Stack – TOS). You retrieve the boxes from the stack in reverse order (C then B then A). A stack is also called a LIFO (last-in-first-out) buffer (similar to a Queue)
PIC18 Microcontroller Stack

- Consists of 31 registers-21-bit wide, called the hardware stack
  - Starting with 1 to 31
  - Stack is neither a part of program memory or data registers.
  - To identify these 31 registers, 5-bit address is needed
  - PIC18 uses one of the special function registers called STKPTR (Stack Pointer) to keep track of the available stack locations (registers).
STKPTR (Stack Pointer) Register

- SP4-SP0: Stack Address
- STKOF: Stack overflow
  - When the user attempts to use more than 31 registers to store information (data bytes) on the stack, BIT7 in the STKPTR register is set to indicate an overflow.
- STKUNF: Stack underflow
  - When the user attempts to retrieve more information than what is stored previously on the stack, BIT6 in the STKPTR register is set to indicate an underflow.
Instructions to Store and Retrieve Information from the Stack

- **PUSH**
  - Increment the memory address in the stack pointer (by one) and stores the contents of the counter \((PC+2)\) on the top of the stack

- **POP**
  - **Discards** the address of the top of the stack and decrement the stack pointer by one

- The contents of the stack (21-bit address), pointed by the stack pointer, are copied into three special function registers
  - **TOSU** (Top-of-Stack Upper), **TOSH** (High), and **TOSL** (Low)
Instructions to Store and Retrieve Information from the Stack

- The PIC18 stack has limited capability compared to other µPs. It resides within its memory, and is limited to 31 locations.

- For a CALL, address of next instruction (nPC) is pushed onto the stack
  - A **push** means to increment STKPTR, then store nPC (Next PC or PC+2) at location [STKPTR].
  - **STKPTR++; [STKPTR] ← nPC**

- A return instruction pops the PC off the stack.
  - A **pop** means read [STKPTR] and store to the PC, then decrement
  - **STKPTR (PC ←[STKPTR], STKPTR--)**
What is the value of PC, TOSU/H/L and STKPTR as you execute each line?

<table>
<thead>
<tr>
<th>nPC</th>
<th>TOS</th>
<th>STKPTR</th>
<th>W</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>movlw 0x20</td>
<td>0x20</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>1</td>
<td>20</td>
<td>push</td>
<td>0x00</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>2</td>
<td>20</td>
<td>push</td>
<td>0x00</td>
</tr>
<tr>
<td>2A</td>
<td>26</td>
<td>1</td>
<td>20</td>
<td>pop</td>
<td>0x00</td>
</tr>
<tr>
<td>2C</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>pop</td>
<td>0x00</td>
</tr>
</tbody>
</table>
In the PIC18F, the stack is used to store the **return address** of a **subroutine** call.

The return address is the place in the calling program that is returned to when subroutine exits.

On the PIC18Fxx, the return address is PC+4, if PC is the location of the **call** instruction.

The return address is PC+2 if it is a **rcall** instruction.
CALL Label, S (0/1) ; Call subroutine located at Label

CALL Label, FAST ; FAST is equivalent to S = 1

- If S = 0: Increment the stack pointer and store the contents of the program counter (PC+4) on the top of the stack (TOS) and branch to the subroutine address located at Label.

- If S = 1: Increment the stack pointer and store the contents of the program counter (PC+4) on the top of the stack (TOS) and the contents of W, STATUS, and BSR registers in their respective shadow registers, and branch to the subroutine address located at Label.

*Remember: CALL is a 2-word Instruction!*
RETURN

- RETURN,0 → gets the address from TOS and moves it to PC, decrements stack pointer
- RETURN,1 → gets the address from TOS and moves it to PC, decrements stack pointer; retrieves all shadow registers (WREG, STATUS, BSR)*
- RETLW → gets the address from TOS and moves it to PC; returns literal to WREG, decrements stack pointer

* 1 or FAST
RCALL, n ;Relative call to subroutine within n = ± 512 ;words (or ± 1 Kbyte)

;Increments the stack pointer and stores the contents of the program counter (PC+2) on the top of the stack (TOS) and branch to the location Label within n = ± 512 words (or ± 1 ;Kbyte)
Example

Program Listing with Memory Addresses

Main Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0020</td>
<td>0EFE</td>
<td>START: MOV LW</td>
</tr>
<tr>
<td>0022</td>
<td>6E94</td>
<td>MOVWF TRISC</td>
</tr>
<tr>
<td>0024</td>
<td>6E01</td>
<td>MOVWF REG1</td>
</tr>
<tr>
<td>0026</td>
<td>C001</td>
<td>MOVWF REG1, PORTC</td>
</tr>
<tr>
<td>002A</td>
<td>E200</td>
<td>CALL DELAY50MC</td>
</tr>
<tr>
<td>0030</td>
<td>D7FA</td>
<td>BRA ONOFF</td>
</tr>
</tbody>
</table>

Subroutine

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040</td>
<td>0EA6</td>
<td>MOV LW D'166'</td>
</tr>
<tr>
<td>0042</td>
<td>6E10</td>
<td>MOVWF REG10</td>
</tr>
<tr>
<td>0044</td>
<td>0610</td>
<td>DECF REG10,1</td>
</tr>
<tr>
<td>0046</td>
<td>E1FE</td>
<td>BNZ LOOP1</td>
</tr>
<tr>
<td>0048</td>
<td>0012</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

Org 0x40

2-Word Instructions

Note:
2-Word 
Inst. 
PC+4 (2A→2E)

After CALL:
- TOS=00 00 2E
- PC=00 00 40
- STKPTR=01

After RETURN:
- PC=2E
- STKPTR=00

END ; will be at the end of the program!
Subroutine Architecture

How do we write a subroutine?

Parameter Passing

Inputs

Basic Functionality
Register Modifications

Outputs

List of Subroutines used
Macros and Software Stack

- **Macro**
  - Group of assembly language instructions that can be labeled with name
  - Short cut provided by assembler
  - Format includes three parts

```plaintext
Push_macro       macro      arg
movff           arg,POSTINC1
endm
```

**USE:**

```
Push_macro       WREG
```

A *push* means to increment STKPTR, then store nPC (Next PC or PC+2) at location [STKPTR].

```
STKPTR++; [STKPTR] ← nPC
```
Macro Description - Example

- See how FSR is loaded and POSTDEC works.
- How a MACRO is being called!
MACRO Application

- Note COUNT is not defined in the MARCO
  - It is the "arg" of the MACRO
- MACRO is assembled after every instance it is called
So what if MACRO is called multiple times?

- A MACRO is assembled after every instance it is called.
Subroutine versus Macro

- **Subroutine (by MPU)**
  - Requires instructions such as CALL and RETURN, and the STACK (overhead)
  - Memory space required by a subroutine does not depend on how many times it is called
  - It is less efficient in terms of execution than that of a macro because it includes overhead instructions such as Call and Return

- **Macro (by assembler)**
  - Based on assembler
  - **Shortcut** in writing assembly code
  - Memory space required depends on how many times it is called
  - In terms of execution it is more efficient because it does not have overhead instructions
More about subroutines…

- Remember subroutines can call other subroutines
- This is referred as **structured** code
Using Table Pointers

- Reading/writing values from/into the program memory one byte at a time

TBLPTRU/H/L (21-bit) → Program Memory (16-bit) → Table Latch (8-bit)

TBLRD* / TBLRD*+ / TBLRD*-

TBLWT* / TBLWT*+ / TBLWT*-
Table Example

Program Memory

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>0028</td>
<td>FFFF</td>
</tr>
<tr>
<td>22</td>
<td>002A</td>
<td>FFFF</td>
</tr>
<tr>
<td>23</td>
<td>002C</td>
<td>FFFF</td>
</tr>
<tr>
<td>24</td>
<td>002E</td>
<td>FFFF</td>
</tr>
<tr>
<td>25</td>
<td>0030</td>
<td>FFFF</td>
</tr>
<tr>
<td>26</td>
<td>0032</td>
<td>FFFF</td>
</tr>
<tr>
<td>27</td>
<td>0034</td>
<td>FFFF</td>
</tr>
<tr>
<td>28</td>
<td>0036</td>
<td>FFFF</td>
</tr>
<tr>
<td>29</td>
<td>0038</td>
<td>FFFF</td>
</tr>
<tr>
<td>30</td>
<td>003A</td>
<td>FFFF</td>
</tr>
<tr>
<td>31</td>
<td>003C</td>
<td>FFFF</td>
</tr>
<tr>
<td>32</td>
<td>003E</td>
<td>FFFF</td>
</tr>
</tbody>
</table>

TBLPTR(U/H/L) = 00 00 40

Pointing to the address

REG10 = 0x02

Save in Prog Memory; Label the value as BUFFER

Rd the register content into Table Latch

0x0002

Program Memory (16-bit)

W = 0x02

TABLAT = 0x02

REG10 = 0x02
LAB: Modify this program such that values 0xaa, 0xbb, 00cc stored in the program memory are copied into REG60,61,62, respectively.

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE8</td>
<td>WREG</td>
<td>0x02</td>
</tr>
<tr>
<td>010</td>
<td>REG10</td>
<td>0x02</td>
</tr>
<tr>
<td>FF5</td>
<td>TABLAT</td>
<td>0x02</td>
</tr>
<tr>
<td>FF6</td>
<td>TBLPTR</td>
<td>0x000040</td>
</tr>
</tbody>
</table>

Program Memory (16-bit)

TBLPTR = 00 00 40

Pointing to the address

W=0x02

REG10=0x02
Examine this code:

```assembly
;//** INITIALIZATION ************/

;//** Constants
BYTECOPY EQU  0x70
BLOCKNUM  EQU  0x05

ORG 0x60
BUFFER DB 0x01, 0x02, 0x01, 0x3, 0x00, 0x00

;/** MICROCODE ************/
GOTO MAIN

BYTECP MACRO STARTHERE
LFSR FSR1, STARTHERE ; SET THE POINTER
MOVLW UPPER BUFFER
MOVLW TBLPTRU
MOVLW HIGH BUFFER
MOVLW TBLPTRH
MOVLW LOW BUFFER
MOVLW TBLPTRL

NEXT_BYTE
TBLRD**
MOVLW 0x0
XORWF TABLAT
BZ ENDOLOOP

MOVFF TABLAT,POSTINC1
GOTO NEXT_BYTE

ENDLOOP
ENDM

;/// Clearing a block of registers
CLEARME MACRO STARTCLEAR
LFSR FSR1, STARTCLEAR
MOVLW 0x0

NEXT_CLEAR
MOVFF WREG, PostINC1
DECF BLOCKNUM
BZ ENDOLOOP_CLEAR
GOTO NEXT_CLEAR

ENDLOOP_CLEAR ENDM

;/// MAIN CODE ************/
org 0x80
MAIN

| MOVLW 0x0
| MOVLW 0x0
| CLEARME 0x60
| BYTECP BYTECOPY
| MOVLW 0x0

END
```

See next slide.....
Answer the following:

- At what location in the program memory CLEARME is built? Explain.
- What are the contents of register 0x60, 0x61, etc. in the program memory?
- Where is the location of FSR1 when the MARCO is called?
- Where exactly does CLEARME macro does? How many registers are effected?
- Where exactly does BYTECP macro does? How many registers are effected?
- Modify the program using MACROs such that you perform the following tasks:
  - Copy NINE values already stored in PROGRAM MEMORY locations, starting with locations 0x80, into RAM locations starting with register 0x80. Assume the numbers are 1-9.
  - Copy NINE values already stored in PROGRAM MEMORY locations, starting with locations 0x80, into RAM locations starting with register 0x90. Assume the numbers are 1-9.
  - Take the sum of all the values and locate the SUM in RAM location 0x100.
  - Delete all the RAM registers starting with with register 0x90 - 0x9F
Modify program 7.5.3 such that you can correctly take the average of any sum.
The number of inputs can be up to 20 non-zero unsigned values
The PIC18 MCU does not provide any divide instruction. Therefore, a divide operation must be synthesized by other instructions. A simple but popular divide algorithm in use today is the repeated subtraction method. This method performs unsigned divide operation. The hardware required for implementing the repeated subtraction method is shown in Figure below.

Before performing the repeated subtraction operation, one needs to load 0, the dividend, and the divisor into registers R, 0, and N, respectively. The carry flag is used to indicate whether the subtraction result is negative. The ALU can perform n-bit unsigned addition and subtraction operations. The repeated subtraction method consists of n steps. Each division step consists of three parts:

- **Step 1**
  - Shift the register pair (R, 0) one place to the left.

- **Step 2**
  - Subtract the contents of N from R and put the result back in R if the result is positive.

- **Step 3**
  - If the result of Step 2 is negative, then set the least significant bit of 0 to 0. Otherwise, set the least significant bit of 0 to 1.

Reference: Huang
Division By Subtraction

```assembly
;** MAIN CODE ************/
org 0x20

MAIN

<table>
<thead>
<tr>
<th>MOVLP</th>
<th>0x84</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVWF</td>
<td>reg10; LO_SUM</td>
</tr>
<tr>
<td>MOVLP</td>
<td>0x4</td>
</tr>
<tr>
<td>MOVWF</td>
<td>reg11; HI_SUM</td>
</tr>
<tr>
<td>CLRPF</td>
<td>QUOTIENT</td>
</tr>
</tbody>
</table>

DIVIDE

<table>
<thead>
<tr>
<th>MOVLP</th>
<th>0xF0; Note that any number can be here!</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCF</td>
<td>QUOTIENT</td>
</tr>
<tr>
<td>SUBWP</td>
<td>reg10</td>
</tr>
<tr>
<td>BTFSC</td>
<td>STATUS, 0; In case of F-W; only when F&lt;W ----&gt; C = 0</td>
</tr>
<tr>
<td>GOTO</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>DECF</td>
<td>reg11</td>
</tr>
</tbody>
</table>

HI_SUM2

<table>
<thead>
<tr>
<th>BTFSC</th>
<th>STATUS, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>DECF</td>
<td>QUOTIENT</td>
</tr>
</tbody>
</table>

END
```
References

Subroutine Documentation and Parameter Passing

- Parameter passing
  - Information exchanged between a calling program and a subroutine
- Subroutine documentation should include:
  - Function of a subroutine: Brief description of what it does
  - Input parameters: Information that should be provided by calling program to subroutine
  - Output parameters: Information or results provided by subroutine to calling program
  - Registers modified: List of registers changed by the subroutine
  - List of subroutines called: List of other subroutines called by the called subroutine