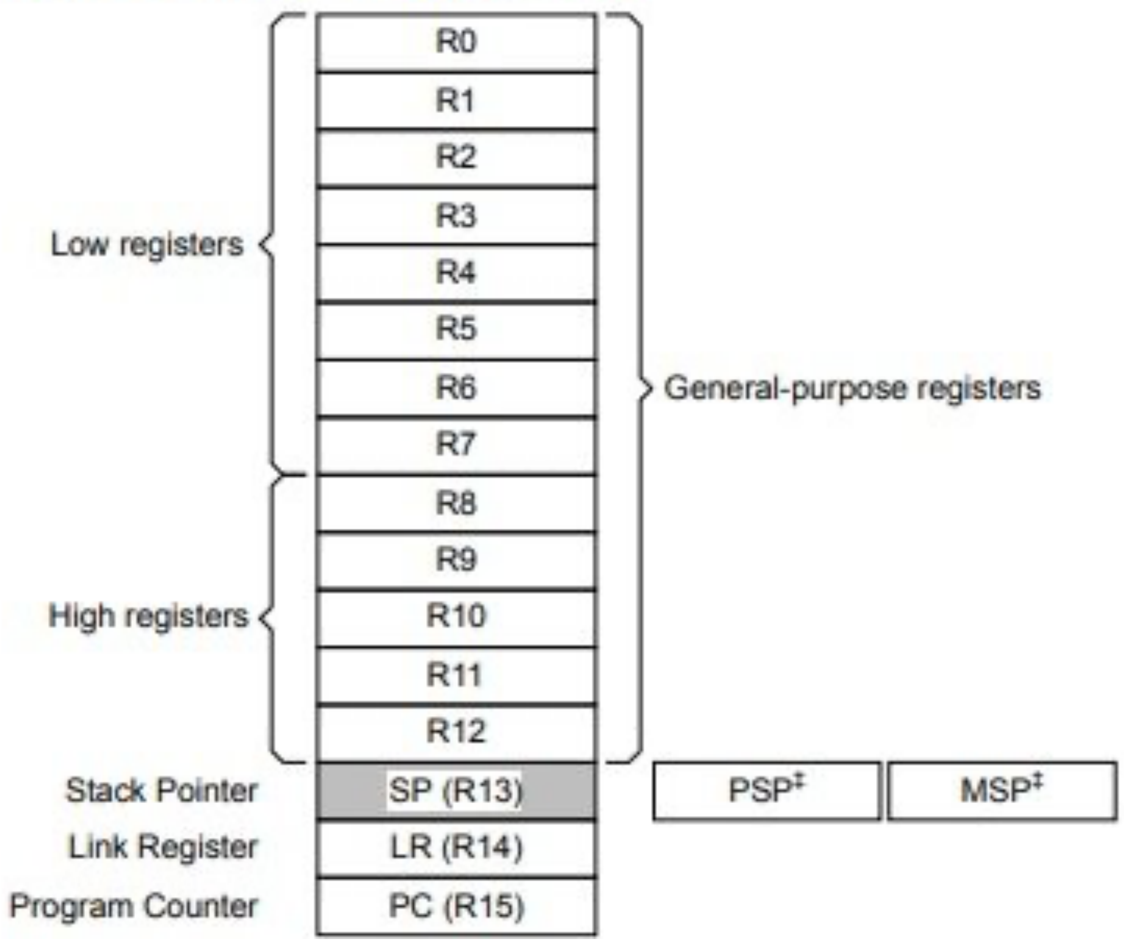
LAB 2: Assembly Language and C Programming

Week 11

NI QINGQING

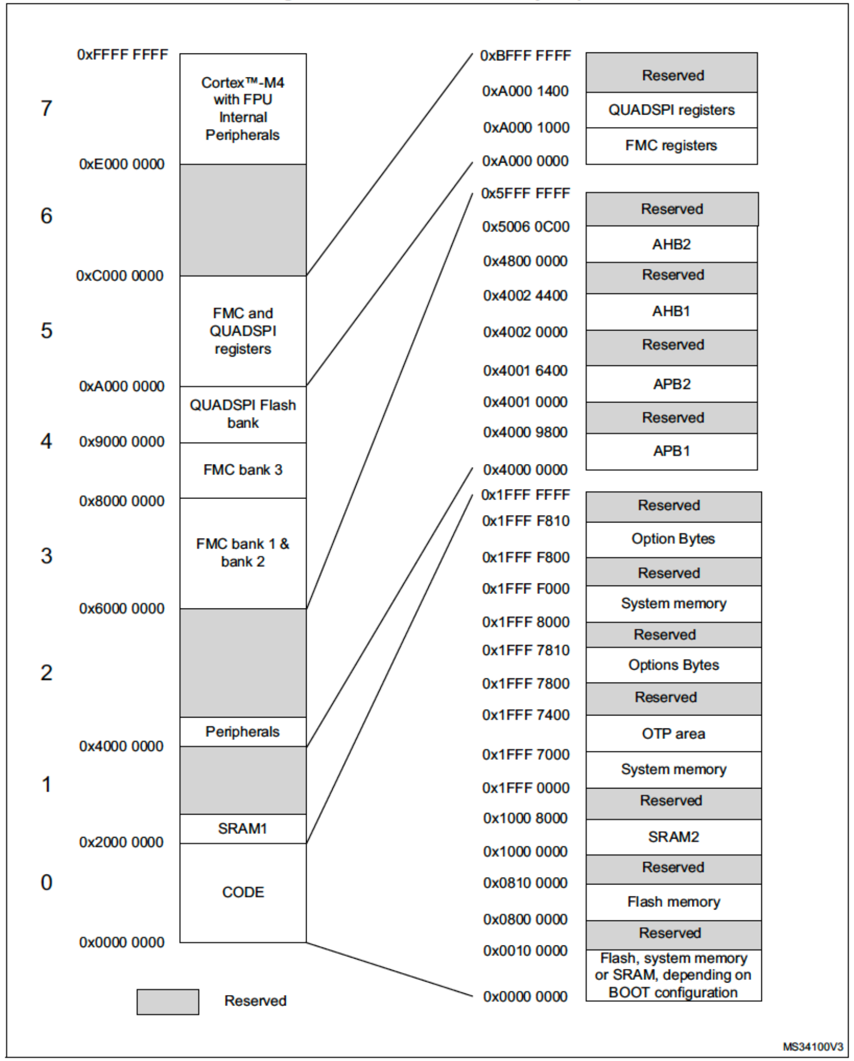
LAB-1 Summary: Registers



- General Purpose Registers **(13 of them, R0-R12)**
- 3 Special Registers:
  - SP
  - LR
  - PC

LAB-1 Summary: Memory

Figure 8. STM32L475xx memory map



- Memory space is segmented
- Each portion has designed to store certain information

LAB-1 Summary: Why ASM?

- ASM instructions are lower-level machine instruction.
- Not commonly used but mostly used for critical part of the program, e.g., core part of operating system were requiring <1ms running time.
- Program in ASM allow full control over microprocessor, the instruction are fully defined by programmer not the compiler.
- Higher level languages like C, Java, Python are being disassembled (translated) from C into ASM, the way of translating is decided by the compiler.
- Fast, full control of processor, low level machine language.

LAB-2 Agenda

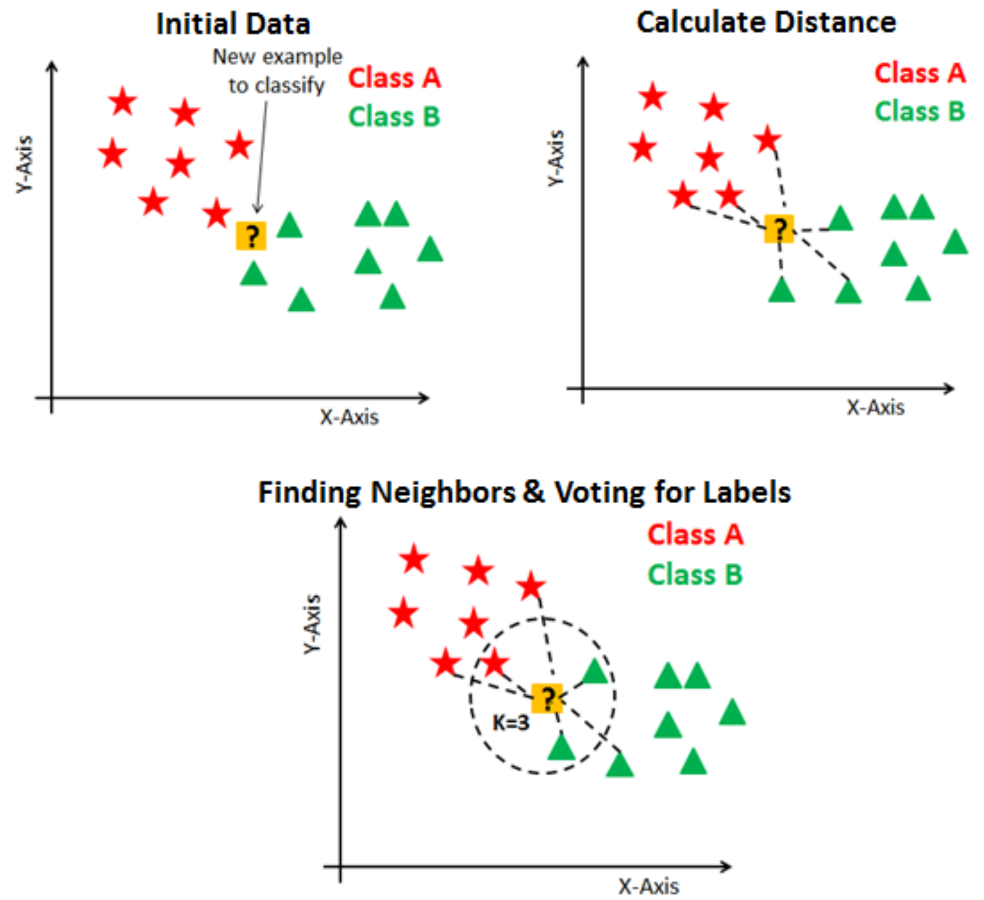
- Assignment introduction:
  - Submission and Assessment
  - Background Concepts: K Nearest Neighbor
  - Objective: develop two ASM function **`classification()`**
  - Walkthrough the Assignment template
- Learning Focus:
  - Pass arguments between C and ASM functions
  - Declare Function in ASM (Optional)
  - Navigates between C and ASM functions - Link Register (LR, R14)
  - Why PUSH and POP? - Stack and Stack Pointer (SP Register, R13)

Submission and Assessment

- Submission: The report and archived workspace have to be uploaded to LumiNUS **on Week 13 Tuesday/Wednesday by 11:59 PM.** Login to the zoom session at least 5 mins before your slot.
- Assessment: **on Week 13 Wednesday/Thursday during lab session.**
- During the Assessment:
  - GA will modify your program slightly
  - GA will ask both students questions related to the code/logic/structure
  - You need to be familiar with all aspects of the assignment
- Other Requirements:
  - Peer Review submit by 11:59 PM 14-Nov (Week 13 Sunday) [LumiNUS > CG2028 > Survey > Peer Review
  - Project (ZIP file only)
  - Report (PDF file only)
  - Prepare your achieve and follow the naming templates in Wiki.NUS

K-Nearest Neighbour (k-NN)

- K-nearest neighbour (k-NN) is one of the popular non-parametric classification methods. It is used for classification and regression.



(image courtesy: https://towardsdatascience.com/knn-visualization-in-just-13-lines-of-code-32820d72c6b6)

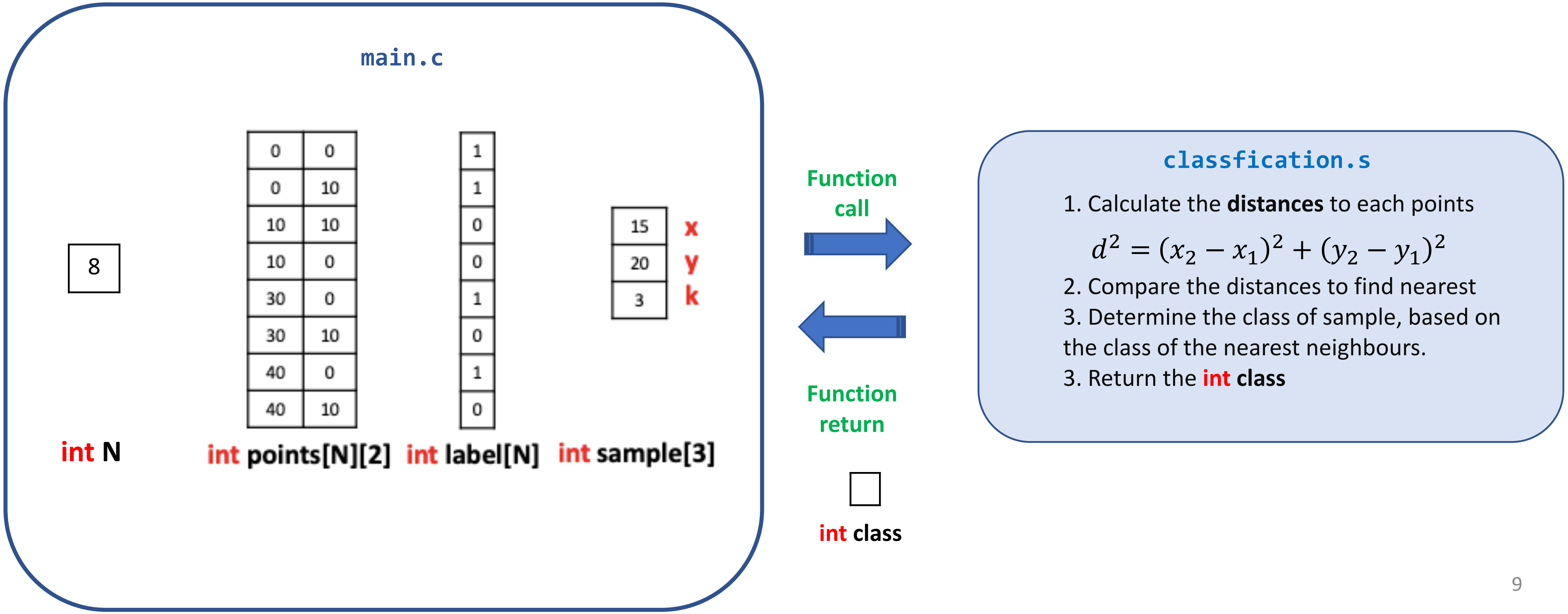
Flow of the Program

Three source files:

- `main.c`:
  - define necessary parameters (not required to edit)
  - calls the assembly function
  - prints out the result on the console pane
- `classfication.s`:
  - write the ASM instructions that implement the **`classification()`** function to classify the data point with 1 nearest neighbour.

K-Nearest Neighbour (k-NN)

- `classfication.s`:
  - classify the data point with k nearest neighbour.
  - Remind you again: int sample in your skeleton code is sample[2] = {15,20}, since we are doing 1-NN, and int class is an integer.



Assignment: Passing Arguments from C to ASM (R0- R3)

```c
#include "stdio.h"
#define k 1


// CG2028 Assignment, Sem 1, AY 2021/22

extern int classification(int N, int* points, int* label, int* sample); // asm implementation
int classification_c(int N, int* points, int* label, int* sample); // reference C implementation

int main(void)
{
    //variables
    int N = 8;
    // think of the values below as numbers of the form x.y
    // (decimal fixed point with 1 fractional decimal digits precision)
    // which are scaled up to allow them to be used integers

    int points[16] = {35, 0, 0, 15, 10, 10, 10, 0, 30, 0, 30, 10, 40, 0, 40, 10};
    int label[8] = {1, 1, 0, 0, 1, 0, 1, 0};
    int sample[2] = {15, 20};

    // Call assembly language function to perform classification
    printf( "asm: class = %d \n", classification(N, points, label, sample) ) ;
    printf( "C  : class = %d \n", classification_c(N, points, label, sample) ) ;

    while (1); //halt
}

```

Function Declaration

Function Call

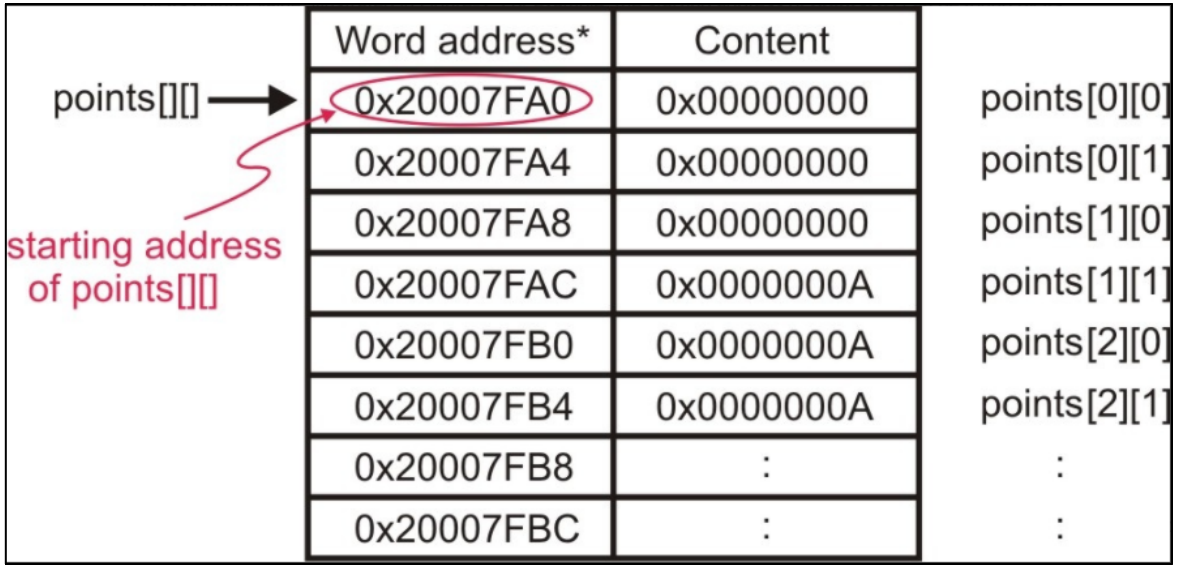
- Function Declaration:
  - **extern void func**(**int*** arg1, **int*** arg2)
  - Why the *?
- When ASM functions being called, we are passing:
  - (**int**) N → R0
  - (**int***) points → R1
  - (**int***) label → R2
  - (**int***) sample → R3
  - What are we passing in exactly?

Assignment: Constant and Variables in Memory

- Question: Knowing the starting address of array points [][], how to find the memory address of the A-th data point, say A ≤ N ?

0	0
0	1
1	1
1	0
3	0
3	1
4	0
4	1

int points[N][2]



- Let’s do a quick demo to exam the values in memories and registers.

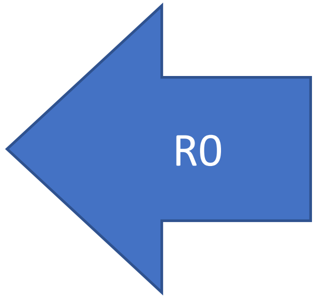
Assignment 1: Passing Arguments from ASM to C (R0)

main.c

```c
#include "stdio.h"
#define k 1


// CG2028 Assignment, Sem 1, AY 2021/22

extern int classification(int N, int* points, int* label, int* sample); // asm implementation
int classification_c(int N, int* points, int* label, int* sample); // reference C implementation
```



asm.s

```
classification:
@ PUSH / save (only those) re
        PUSH {R1-R4,R14}
@ parameter registers need no

@ write asm function body her

@ branch to SUBROUTINE for il
        BL SUBROUTINE
@ prepare value to return (cl
@ the #5 here is an arbitrary
        MOVW R0, #5
@ POP / restore original regi
        POP {R1-R4,R14}
@ return to C program
        BX  LR

@ you could write your code w
SUBROUTINE:

        BX LR
```

- Are we returning any values here?
- If we want to return, how?
- If not allowed to use R0, can you think of several ways to save the result in the memory?

Assignment: Declare Subroutine (Function) in ASM (Optional)

```
24 classification:
25 @ PUSH / save (only those) registers which are modified by your function
26         PUSH {R1-R4,R14}
27 @ parameter registers need not be saved.
28
29 @ write asm function body here
30
31 @ branch to SUBROUTINE for illustration only
32         BL SUBROUTINE
33 @ prepare value to return (class) to C program in R0
34 @ the #5 here is an arbitrary result
35         MOVW R0, #5
36 @ POP / restore original register values. DO NOT save or restore R0. Why?
37         POP {R1-R4,R14}
38 @ return to C program
39         BX  LR
40
41 @ you could write your code without SUBROUTINE
42 SUBROUTINE:
43
44         BX LR
```

Function Call

Function Declaration

- Line 32: Branch to SUBROUTINE
- Execute SUBROUTINE
- Line 44: Branch back to main function in **ASM**
- Execute the rest in main of ASM
- Line 39: Branch back to **main.c**
- **Question1: Why PUSH & POP R14?**
- **Question2: Why not R0?**

Navigates between C and ASM: LR - Link Register (R14)

	Word Address	Instruction Memory
asm_fun.s	e.g. 0x0000 0070	ADD …
	0x0000 0074	SUB …
	…	…
	0x0000 0100	**BX LR**
main.c	…	
	A block of memory location	int i,j;
	e.g. 0x0000 2000 to 0x0000 2008	int points [N][2];
Why a range?	0x0000 2009 to 0x0000 2014	int class[N][2];
	…	…
	…	asm_fun(int* points, int* class);
	…	…
	…	for (i=0; i<M; i++)
	…	…

BX → Branch Indirect (Register)
Format: BX*{cond} Rm*
Performs: branch to location indicated by Rm
PC ← Rm
PC: program counter always points to the next line that should be execute
BX LR: LR serves as a marking to navigate back

Let's take a look at our code.

- Many registers are involved to create the "link" between C and ASM, losing the link will cause problems when navigate back from ASM to C.
- ASM function should not affect C program after its execution so that main.c could continue. **HOW?**

Why PUSH and POP?

- Discussed earlier: PUSH and POP helped us preserve the “marking” we made to navigate back to C.

```
classification:
@ PUSH / save (only those) r
        PUSH {R1-R4,R14}
@ parameter registers need r

@ write asm function body he

@ branch to SUBROUTINE for i
        BL  SUBROUTINE
@ prepare value to return (c
@ the #5 here is an arbitrar
        MOVW R0, #5
@ POP / restore original reg
        POP {R1-R4,R14}
@ return to C program
        BX   LR

@ you could write your code
SUBROUTINE:

        BX LR
```

(int*) N → R0

(int*) points → R1

(int*) label → R2

(int*) sample → R3

- We are to use R0-R3 in asm_fun.s as well, do we need to PUSH and POP them as well? **YES! But not R0!**
- WHY? IDE translates C language into Assembly then implement on the board → Essentially, main.c is relying on registers to do processing.
- We must preserve the status and revert back when we return to C.
- If you used more registers, what should you do?

How PUSH/POP works: Stack & Stack Pointer (R13)

- A very commonly used data structure
- A part of the memory is dedicated as a “Stack”
- Stack Pointer (SP) always pointing to the top of the stack

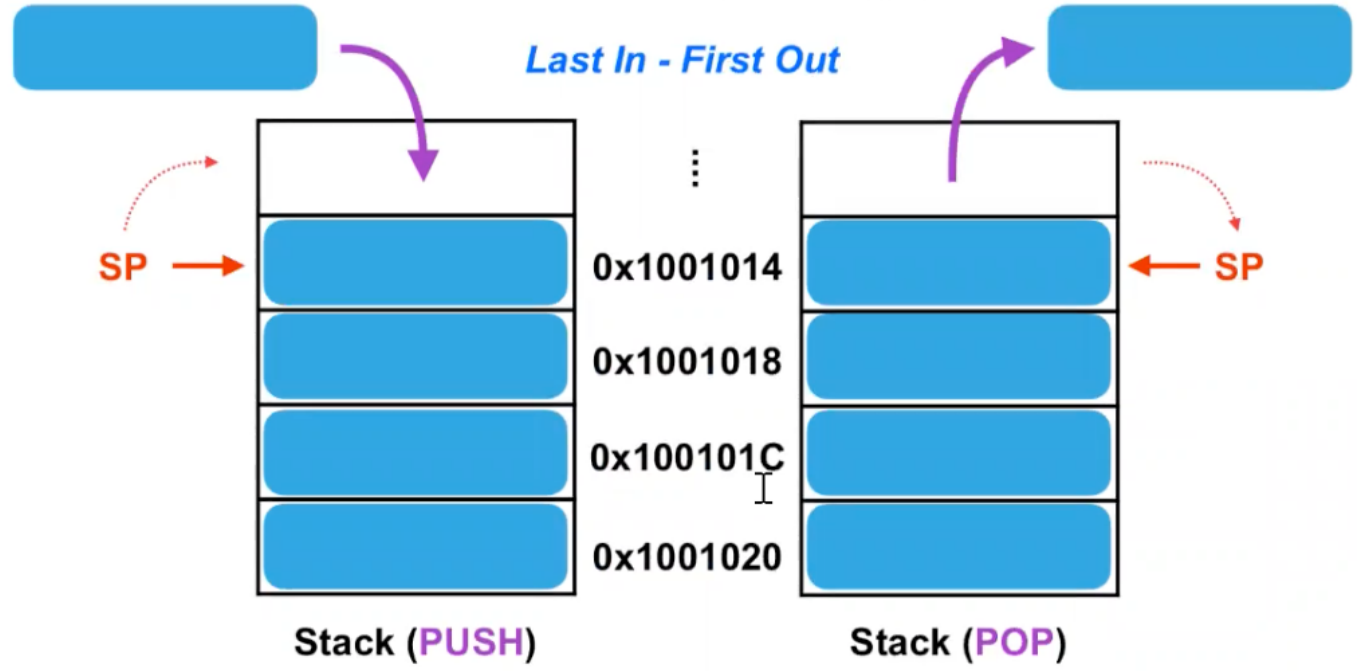


Image curtesy to Dr. GU Jing

Good practices to follow

- Use and re-use registers in a systematic way to reduce the usage of processor.
- Give meaningful comments helps you and your teammate understand each other (also remind yourself if you happen to have fish memory)
- Maintain a register dictionary or table for each asm_fun.s at different time.

```
@Register map
@R0 - N, returns class
@R1 - points
@R2 - label
@R3 - sample
@R4 - <use(s)>
@R5 - <use(s)>
@....
```

- Make use of flowchart to design before attempt to code, include it in your report to better explain your logic.

Exercise to Warm up

- Compile the project and execute the program, **exam your registers and memory** to locate all the predefined parameters.
- Comment the **PUSH {R14}** and **POP {R14}** lines in **classification()**, recompile and execute the program again. Observe the difference.