PCI-2: Assembly Language and C Programming

CG2028 COMPUTER ORGANIZATION

Hou Linxin (TA), Week 4

Lab 1 Homework

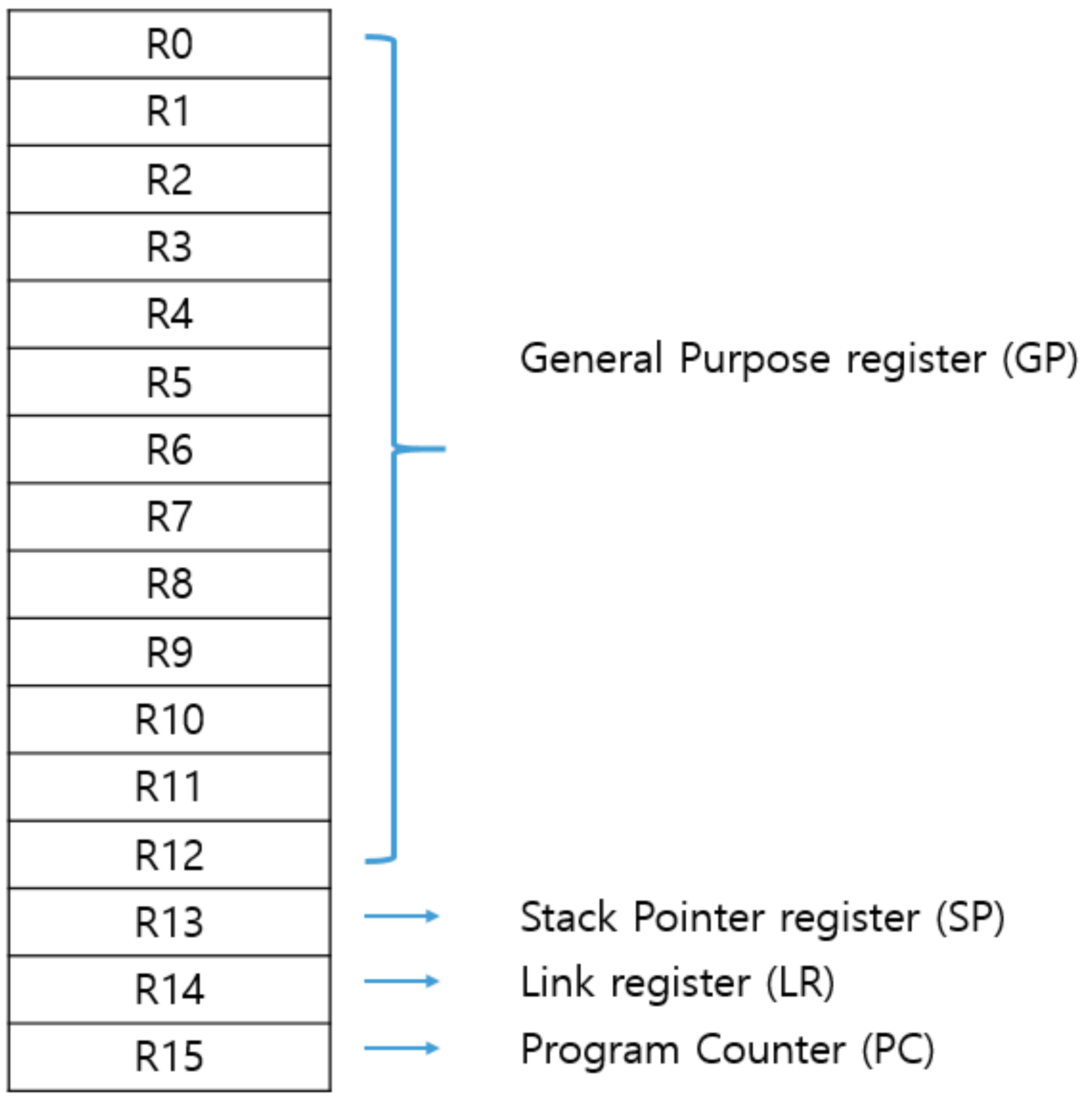
Possible Solution

- Modify on the **main.s** of **asm_basic,** to store all the odd numbers between 50 and 100 into the memory, starting from the address right after where **ANSWER** is.
- **Hint 1:** If the address of constant **ANSWER** is **ADDR**, the address of next memory location right **ANSWER** is **ADDR+4**.
- **Hint 2:** A loop needs to be created to recurring store values into the memory.

```
          MOV R5, #51
          MOV R6, #101
homework_loop:
          STR R5, [R3, #4]!
          ADD R5, #2
          CMP R5, R6
          BEQ HALT
          B homework_loop
```

Lecture Summary

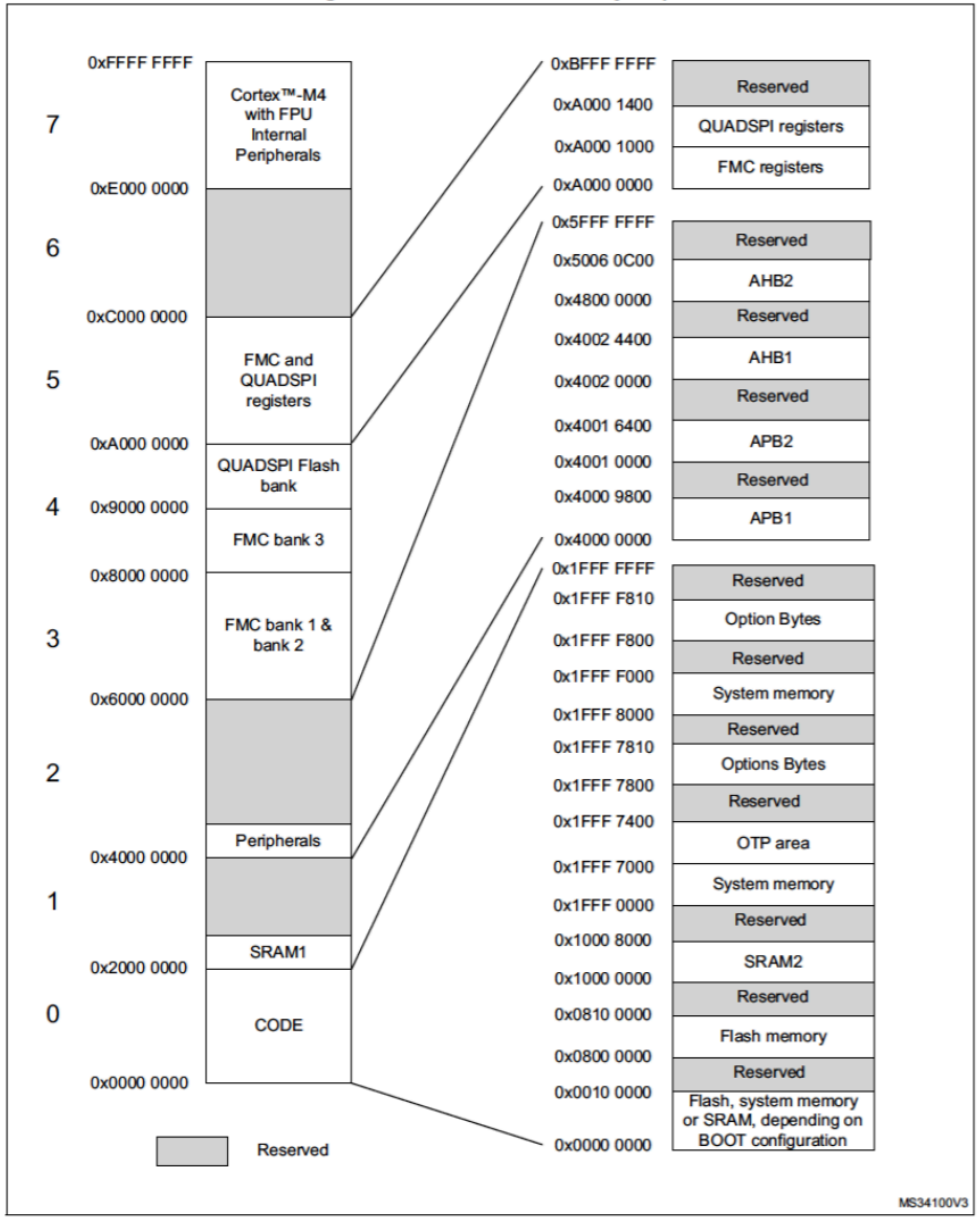
- 13 General Purpose Registers (R0-R12)
- 3 Special Registers: SP, LR, PC



Lecture Summary

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

Figure 8. STM32L475xx memory map



Lecture Summary

- Assembly: low-level programming language that is communicate directly with a computer's hardware

- Higher level languages like C, Java, Python are being disassembled (translated) from C into ASM, the way of translating is decided by the compiler.

- Memory Assessing; Data Processing; Branching

Objectives of Today

- Assignment introduction:
  - Background Concepts: Optimisation & Gradient Descent
  - Objective: develop ASM function optimize()
  - Walkthrough Assignment 1 template
  - Submission Requirement
- Learning Focus:
  - Pass arguments between C and ASM functions
  - Declare Function in ASM
  - Navigates between C and ASM functions - Link Register (LR, R14)
  - Why PUSH and POP? - Stack and Stack Pointer (SP, R13)

Optimization - Gradient Descent

- A function or subroutine can be programmed in assembly language and called from a C program. In this assignment, you will write an assembly language function that performs optimization using gradient descent.
- https://www.youtube.com/watch?v=0kFydRfswU8
- (Watch video until 6:55)
- In this assignment, we shall use gradient descent to find the solution x^* that minimizes a **quadratic** cost function so that it is easy for us to check the correctness of the answer.

Flow of the Programme

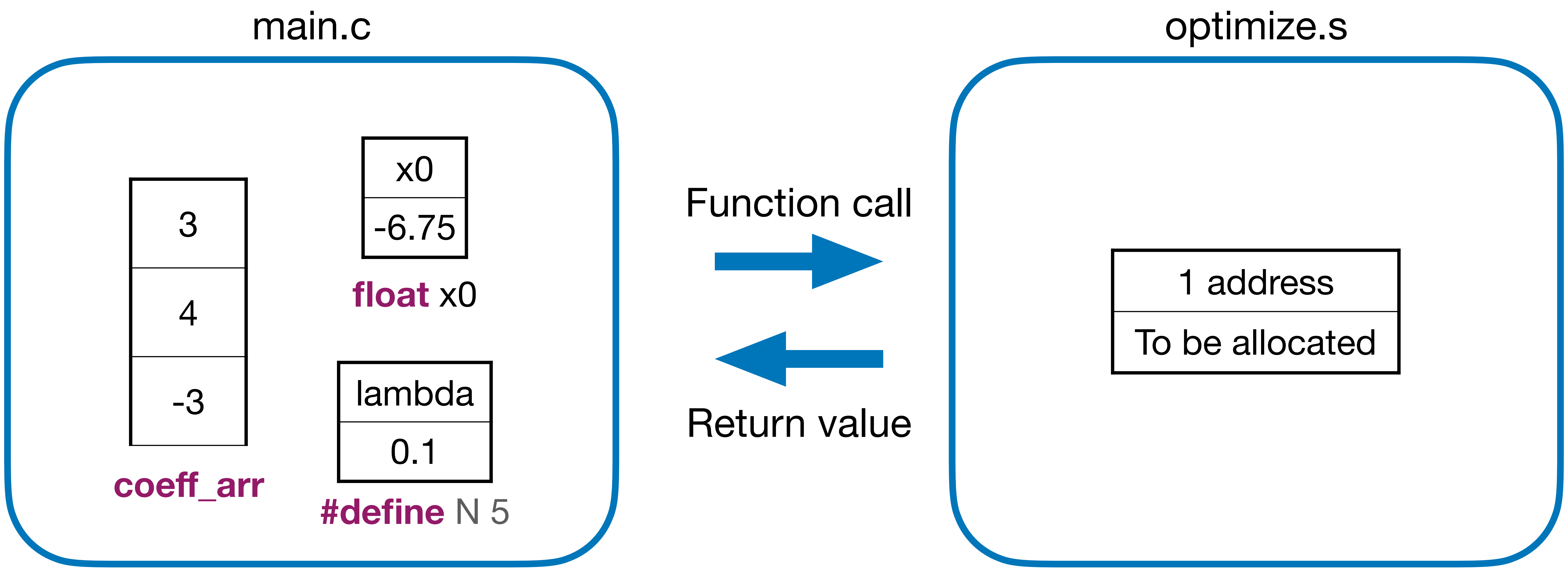
- Two source files:
  - **main.c**:
    1. Defines necessary parameters (not required to edit)
    2. Calls the assembly function
    3. Prints out the result on the console pane
  - **optimize.s**:
    1. write the ASM instructions that implement the optimize() function to produce xsol and the number of rounds.

Assignment 1

Debug Configurations

- Follow the Canvas pages instructions…

Programme Structure



Assignment 1 Breakdown (main.c)

```c
#include "stdio.h"

// Necessary function to enable printf() using semihosting
extern void initialise_monitor_handles(void);

// Functions to be written in assembly
extern int* optimize(int* coeff_arr, int x0_int, int lambda_int);

// Optimization implementation in C
void optimize_c(int a, int b, float x0, float lambda)
{
    float fp, xprev, change, x=x0;
    int round = 0;
    while (1)
    {
        fp = 2*a*x + b;
        xprev = x;
        change = -lambda*fp;
        x = x + change;
        round = round + 1;

//      printf("x: %f, fp: %f, change: %f\n", x, fp, change); //uncomment to see each step
        if (x==xprev) break;
    }
    printf("xsol : %.1f No. of rounds : %d \n\n", x, round);
    return;
}
```

extern functions pass parameters between C and assembly programs through the ARM Cortex-M4 registers.

Function declaration (assembly)

Function declaration (C)

Optimisation in C

(You may use the same logic for the assembly function. You can also use a new logic.)

Assignment 1 Breakdown (main.c)

```c
int main(void)
{
    // Necessary function to enable printf() using semihosting
    initialise_monitor_handles();

    // modify the following lines for different test cases
    int a=3, b=4, c=-3;    // Polynomial coefficients
    float x0 = -6.7;       // Starting point
    float lambda = 0.1;    // Learning Rate

    // NOTE: DO NOT modify the code below

    /*
     * Multiply by 10 to convert floats (1 decimal place) to integers for assembly,
     * Divide by 10 to get the true float result back from assembly.
     */
    int arr[3] = {a*10, b*10, c*10};  //array to pack scaled coefficients
    int x0_int = x0*10;               //Scale starting point
    int lambda_int = lambda*10;       //Scale learning rate

    // call optimize.s
    printf("ASM version:\n");
    int *xsol = optimize((int*)arr, x0_int, lambda_int);
    float xsol_float = xsol[0] / 10;
    int xsol_round = xsol[1];
    printf("xsol : %.1f No. of rounds : %d \n\n", xsol_float, xsol_round);

    // call optimize.c
    printf("C version:\n");
    optimize_c(a, b, x0, lambda);
}
```

Define variables

Function call (assembly)

Function call (C)

Assignment 1 C to ASM

Pointer: passing the address into the function

```
// call optimize.s
printf("ASM version:\n");
int *xsol = optimize((int*)arr, x0_int, lambda_int);
float xsol_float = xsol[0] / 10;
int xsol_round = xsol[1];
printf("xsol : %.1f No. of rounds : %d \n\n", xsol_float, xsol_round);
```

When ASM function is called in C program, starting address of **arr** and value of **x0_int and lanbda_int** are passed into the registers:

(**int***) arr —> R0; (**int**) x0_int —> R1;
(**int**) lambda_int —> R2

A maximum of **4** parameters can be passed, but we are passing 5 elements. How are the values in this array known by assembly? 🧐

Writing the parameters in an array and passing the address of the first element in that array

Assignment 1: Constant and Variables in Memory

- Question 1: how to access the elements in the array from a assembly function? Given the n-th element's address in x[N] is <u>*ADDR*</u>, what would be the address of the (n+k)-th element in x[N]? Given that n and k are two constant decimal integers, and <u>*ADDR*</u> is a constant hexadecimal integer.

Example:

```c
int arr[M] = {22, 5, 2, 32, 66, 10};
```

22
5
2
32
66
10

int arr[M]

starting address of Arr[]

Word address*	Content	
0x20007FA0	0x00000000	Arr[0]
0x20007FA4	0x00000000	Arr[1]
0x20007FA8	0x00000000	Arr[2]
0x20007FAC	0x0000000A	Arr[3]
0x20007FB0	0x0000000A	:
0x20007FB4	0x0000000A	:
0x20007FB8	:	
0x20007FBC	:	

Assignment 1 ASM to C

```c
// Functions to be written in assembly
extern int* optimize(int* coeff_arr, int x0_int, int lambda_int);
```

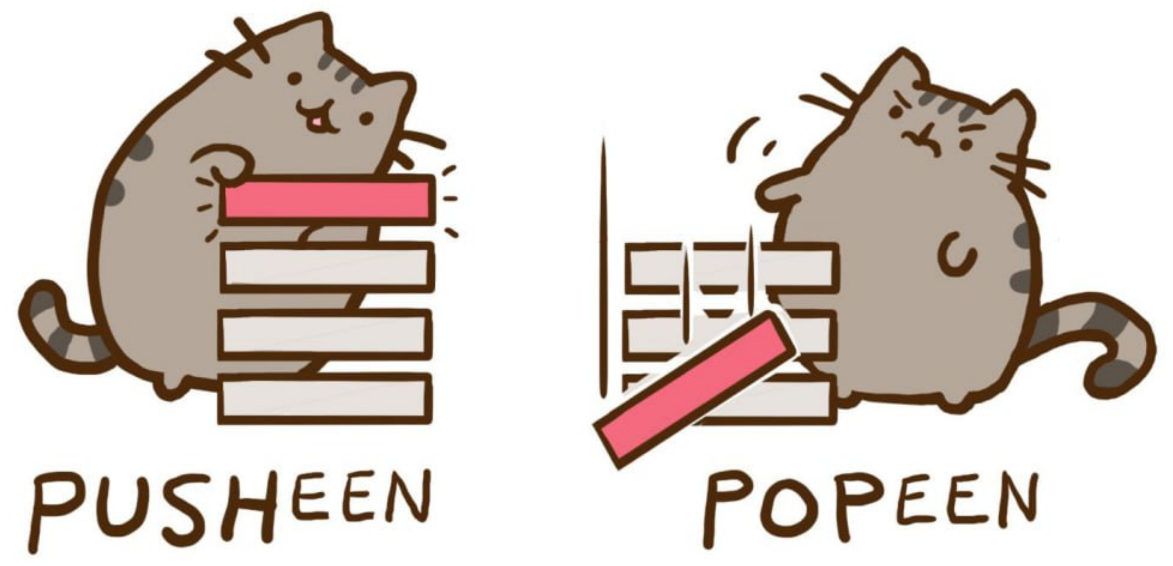
```c
// call optimize.s
printf("ASM version:\n");
int *xsol = optimize((int*)arr, x0_int, lambda_int);
```

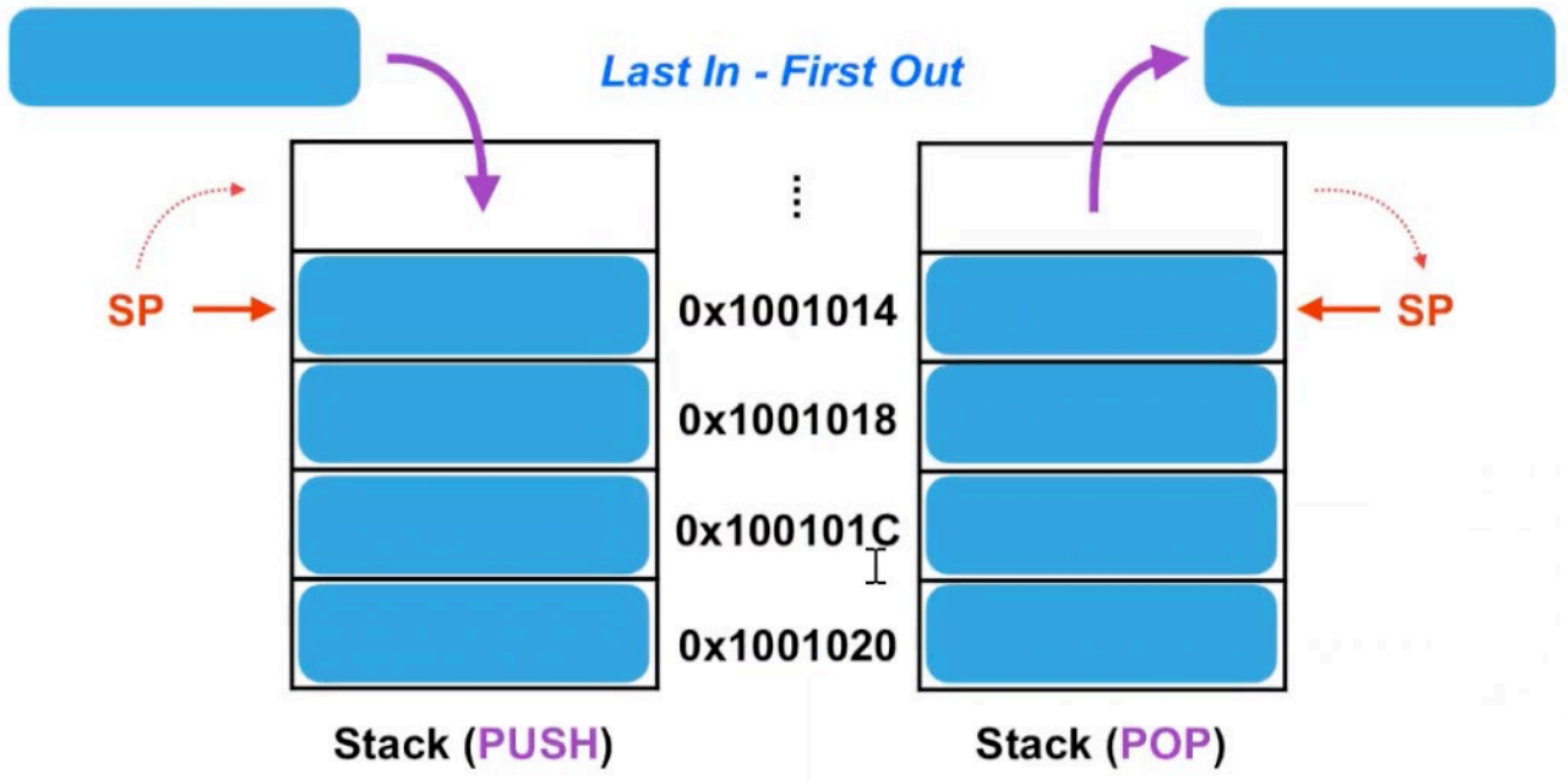
ASM function returns a integer pointer that points to the first element of the resulting array

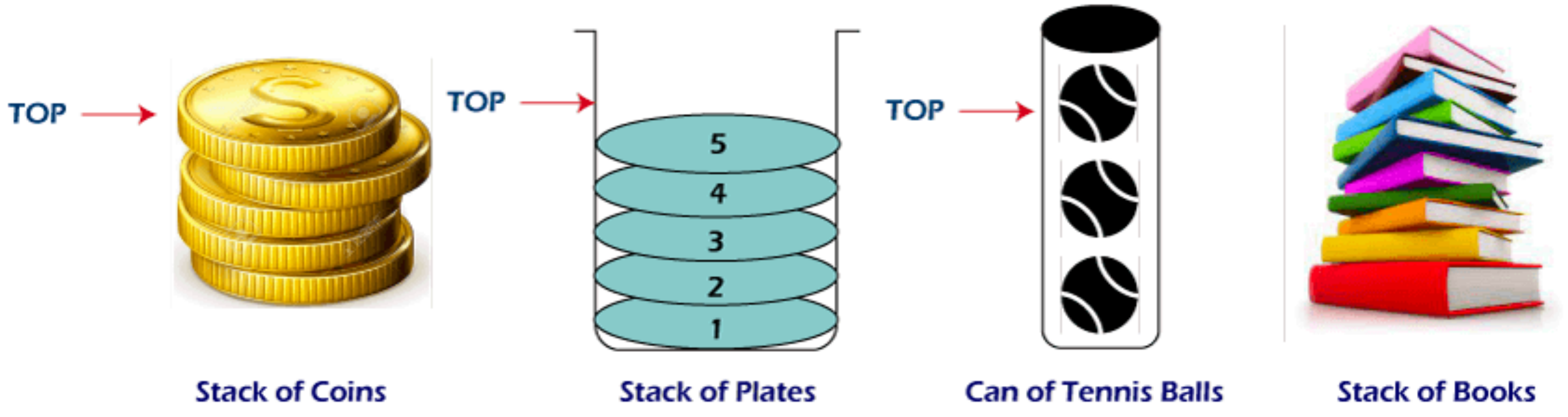
Return value from ASM to C should be put in R0

Assignment 1: 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







Assignment 1: PUSH and POP

`ASM_FUNC:`	
`    PUSH {R14}`	
`    BL  SUBROUTINE`	Branch to SUBROUTINE Execute to SUBROUTINE
`    POP {R14}`	
`    BX   LR`	Branch back to main.c
`SUBROUTINE:`	
`    BX LR`	Branch back to main function in ASM Execute the rest in main of ASM

Assignment 1: PUSH and POP

- Question 2:
- Compile the “Assign1” project and execute the program.
- Comment the **PUSH {R14}** and **POP {R14}** lines in optimize**()**, recompile and execute the program again.
- Observe the difference in (i) and (ii).

```
ASM_FUNC:
        PUSH {R14}

        BL  SUBROUTINE          Function call

        POP {R14}
        BX   LR

SUBROUTINE:                     Function declaration

        BX  LR
```

Assignment 1: PUSH and POP

	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 arr[M] = {20, 12, 10, 15, 2};
	…	…
	…	…
	…	asm_fun(int* a, int b);
	…	…
	…	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

- 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.

Assignment 1: PUSH and POP

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

```
ASM_FUNC:
    PUSH {R14}

    BL SUBROUTINE

    POP {R14}
    BX  LR

SUBROUTINE:

    BX LR
```

- 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 we are to use R0-R3 in **asm_fun.s** as well, do we need to PUSH and POP them as well, how about R4-R11? 🧐

R0-R3, R12 do not need to be PUSH and POP, (caller saved)
R4-R11, R14 need to be PUSH and POP (callee saved)

Submission

- You only need to submit your "optimize.s" code as a .txt file and your report as a .pdf file. In your report:
  - answers to the 5 questions asked in the assignment manual,
  - microarchitecture design that supports MLA and MUL instructions
  - discussions of your program logic (overall logic flow, do not explain line by line),
  - discussions of the improvements you have made that enhance your program efficiency (reusing registers, more efficient algorithms, etc.),
  - and an Appendix that declares every member's joint and specific individual contributions towards this assignment.

Assignment 1 Breakout

- 50% towards final grade, 50 marks in total
  - Code: 20 marks
    - 4 given test cases - 2 marks per each case
    - 2 hidden test cases - 2 marks per each case
    - 3 marks for coding optimisation and style
    - 5 marks for machine codes
  - Report: 30 marks
    - 16 marks for Q&A
    - 5 marks for microarchitecture design
    - 4 marks for program logic
    - 5 marks for discussions of the improvements
  - Peer evaluation: only if necessary

Assessment Tips

- You only need to edit the optimize.s program, other programs shall remain unchanged.
- You can change your array to validate your program

Programming Tips

- 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.

```
classification:
    @ R0:   points10
    @ R1:   centroids10
    @ R2:   class
    @ R3:   new_centroids10

    PUSH {R14} //Preserve marking to C
```

- More ASM commands in MPUs programming manual pg.50 onwards Reading data sheet/manual is a very important part of EE2028 lab. (Some self-learning required)