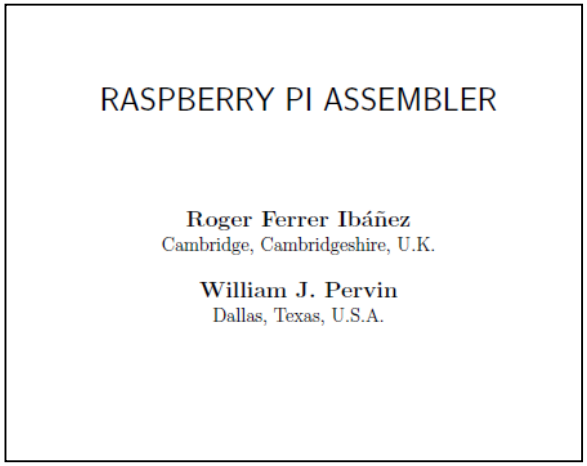
Raspberry Pi Assembler
Functions

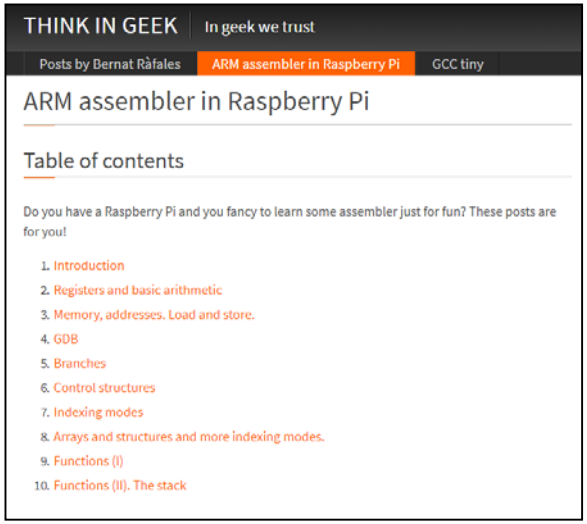

RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

William J. Pervin
Dallas, Texas, U.S.A.

Chapter 9: Raspberry Pi Assembler
“Raspberry Pi Assembler” by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/


THINK IN GEEK In geek we trust

Posts by Bernat Ràfales | ARM assembler in Raspberry Pi | GCC tiny

ARM assembler in Raspberry Pi

Table of contents

Do you have a Raspberry Pi and you fancy to learn some assembler just for fun? These posts are for you!

1. Introduction
2. Registers and basic arithmetic
3. Memory, addresses. Load and store.
4. GDB
5. Branches
6. Control structures
7. Indexing modes
8. Arrays and structures and more indexing modes.
9. Functions (I)
10. Functions (II). The stack

Embedded Systems II – EEE3096S
Y. Abdul Gaffar
S. Winberg

Raspberry Pi Assembler
Functions

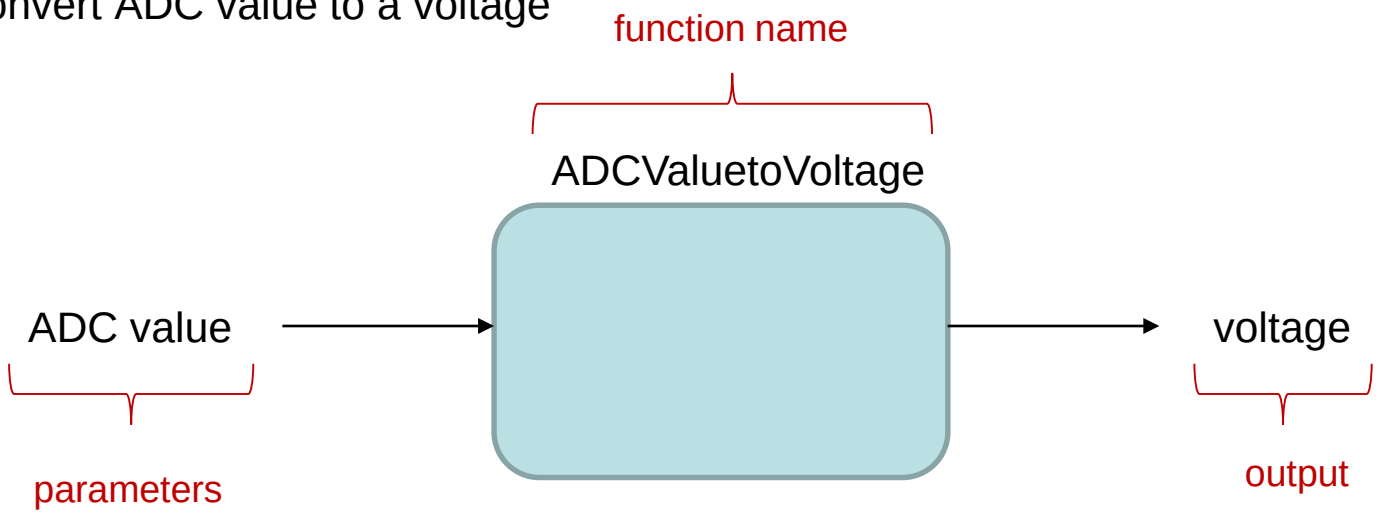
- What is a function?
  - A set of instructions that is typically needed by a program more than once
  - A function receives parameters as inputs and may provide an output

- Examples of functions
  - Convert temperature from °F to °C
  - Convert ADC value to a voltage

Raspberry Pi Assembler
Functions

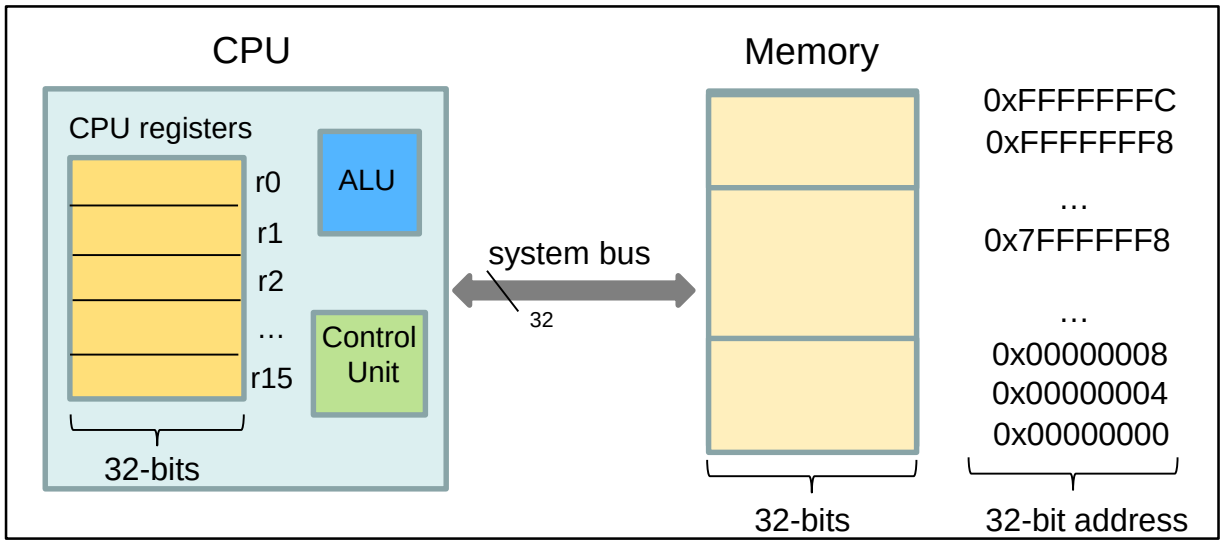
- What is a function?
  - A set of instructions that is typically needed by a program more than once
  - A function receives parameters as inputs and may provide an output

- Examples of functions
  - Convert temperature from °F to °C
  - Convert ADC value to a voltage



Raspberry Pi Assembler
Functions in assembly

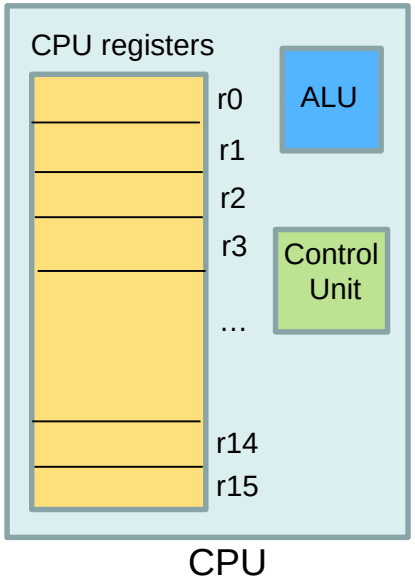
- In assembly, parameters are passed into a function using the CPU registers r0, r1, r2 and r3.
  - The first four parameters of a function are stored in registers r0, r1, r2 and r3
  - More than four parameters may be passed into a function using the stack. The concept of a stack is discussed in the next chapter



Simplified block diagram of a modern computer

Raspberry Pi Assembler
Functions in assembly

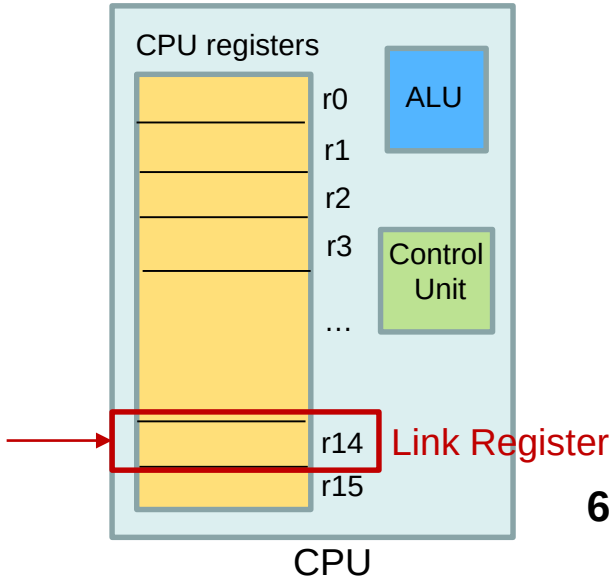
- ARM assembly functions must follow the ARM Architecture Procedure Call Standard (AAPCS)
  - At the start of a function, no assumptions can be made about the contents of the CPSR since the previous instruction that caused the CPSR to update is unknown.
  - Functions may receive input parameters through registers r0, r1, r2 and r3. The contents of these registers can be modified by the function
  - A function can modify r4 to r11. However, before the function exits, each of these registers should be restored to the initial value of the register at the start of the function



Raspberry Pi Assembler
Functions in assembly

- ARM assembly functions must follow the ARM Architecture Procedure Call Standard (AAPCS)
  - At the start of a function, no assumptions can be made about the contents of the CPSR since the previous instruction that caused the CPSR to update is unknown.
  - Functions may receive input parameters through registers r0, r1, r2 and r3. The contents of these registers can be modified by the function
  - A function can modify r4 to r11. However, before the function exits, each of these registers should be restored to the initial value of the register at the start of the function

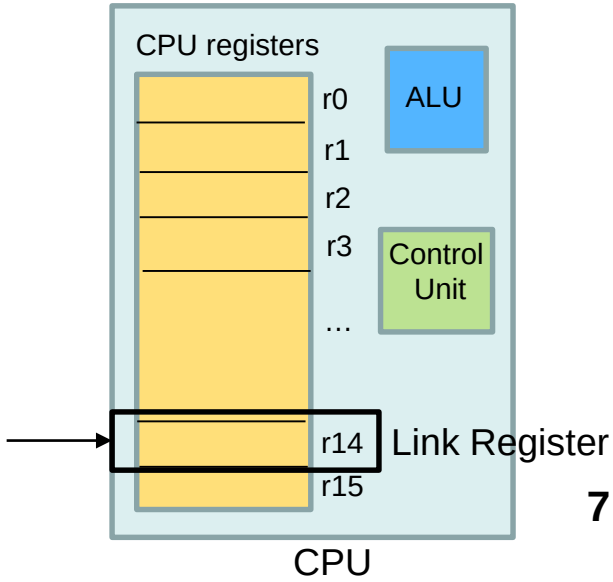
When a program **branches** to a function, the memory address of the next instruction in the main program is stored in CPU register r14, which is called the Link Register



Raspberry Pi Assembler
Functions in assembly

- ARM assembly functions must follow the ARM Architecture Procedure Call Standard (AAPCS)
  - At the start of a function, no assumptions can be made about the contents of the CPSR since the previous instruction that caused the CPSR to update is unknown.
  - Functions may receive input parameters through registers r0, r1, r2 and r3. The contents of these registers can be modified by the function
  - A function can modify r4 to r11. However, before the function exits, each of these registers should be restored to the initial value of the register at the start of the function
  - Similar to 4 to r11, the function may modify the contents of r15 (link register), however before exiting the function, the contents of r15 should be restored to initial value upon entering the function.

When a program **branches** to a function, the memory address of the next instruction in the main program is stored in CPU register r14, which is called the Link Register



Calling a function

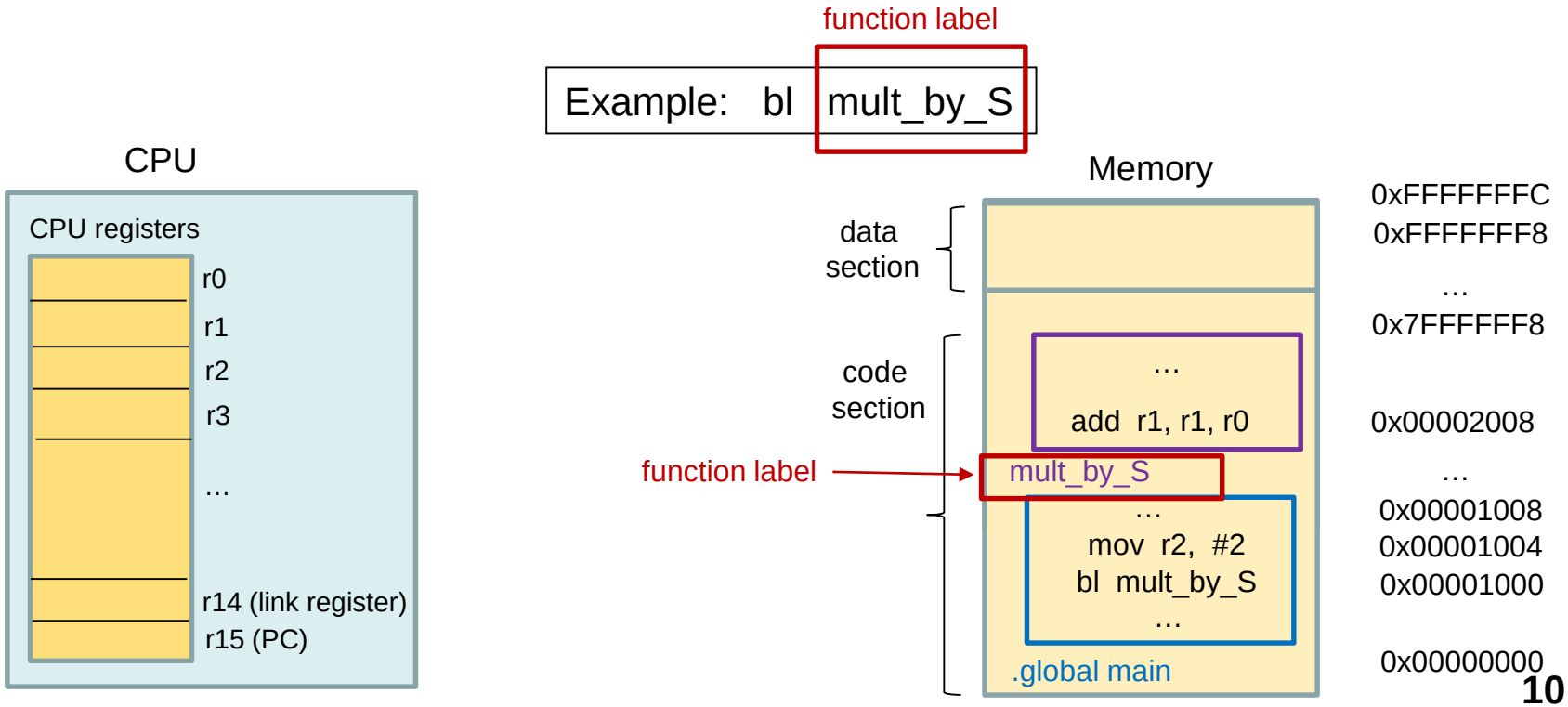
Raspberry Pi Assembler
Calling a function

- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call): Use the **B**ranch with **L**ink (bl) instruction together with the label of the function. The label represents the address of the first instruction of the function and must be defined in the .text section.

```
Example:   bl   mult_by_S
```

Raspberry Pi Assembler
Calling a function

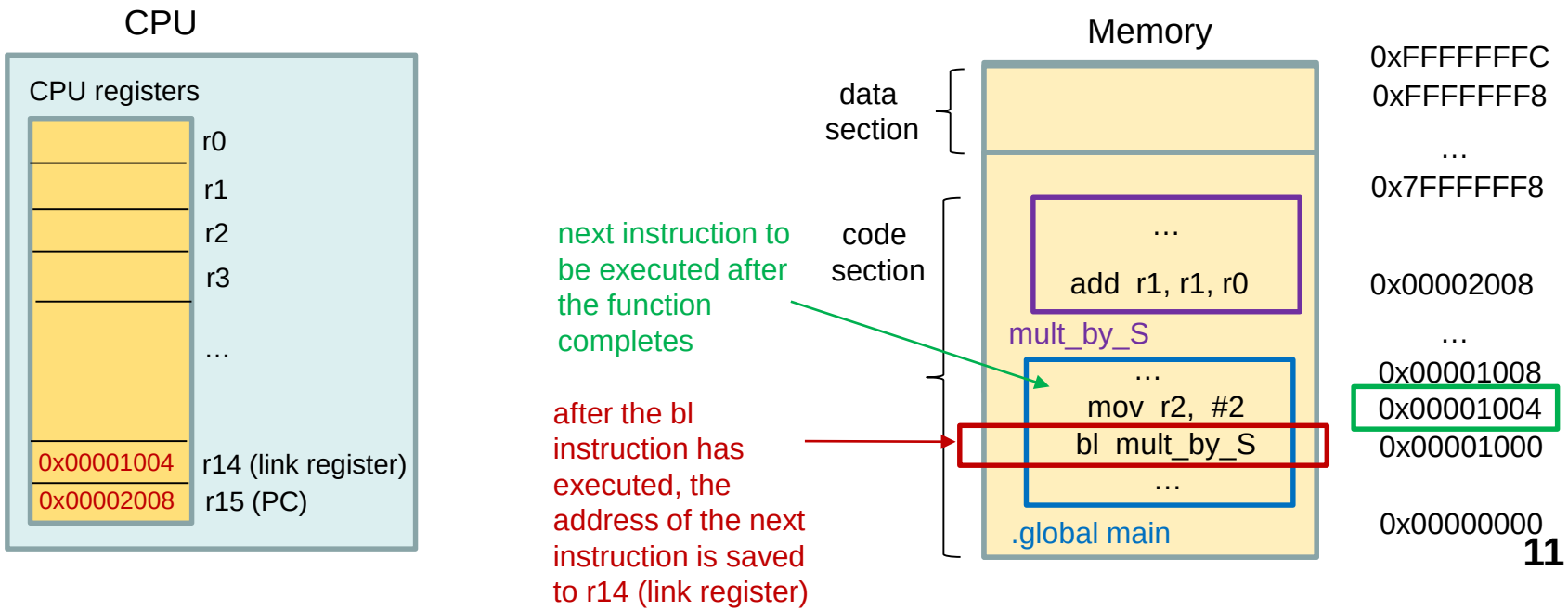
- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call): Use the **B**ranch with **L**ink (bl) instruction together with the label of the function. The label represents the address of the first instruction of the function and must be defined in the .text section.



Raspberry Pi Assembler
Calling a function

- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call): Use the **B**ranch with **L**ink (bl) instruction together with the label of the function. The label represents the address of the first instruction of the function and must be defined in the .text section. After the bl instruction has executed, the address of the next instruction in the main program is saved in the Link Register (R14).

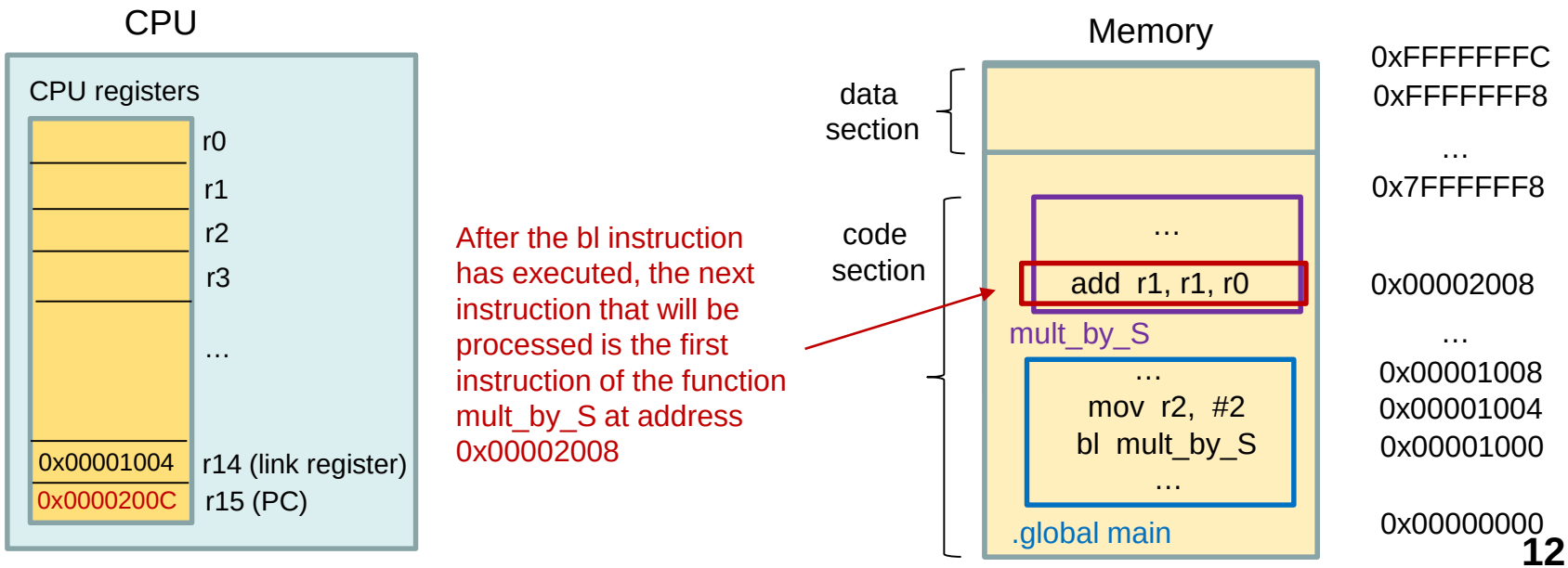
Example: bl mult_by_S



Raspberry Pi Assembler
Calling a function

- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call): Use the **B**ranch with **L**ink (bl) instruction together with the label of the function. The label represents the address of the first instruction of the function and must be defined in the .text section. After the bl instruction has executed, the address of the next instruction in the main program is saved in the Link Register (R14).

Example: bl mult_by_S



Raspberry Pi Assembler
Calling a function

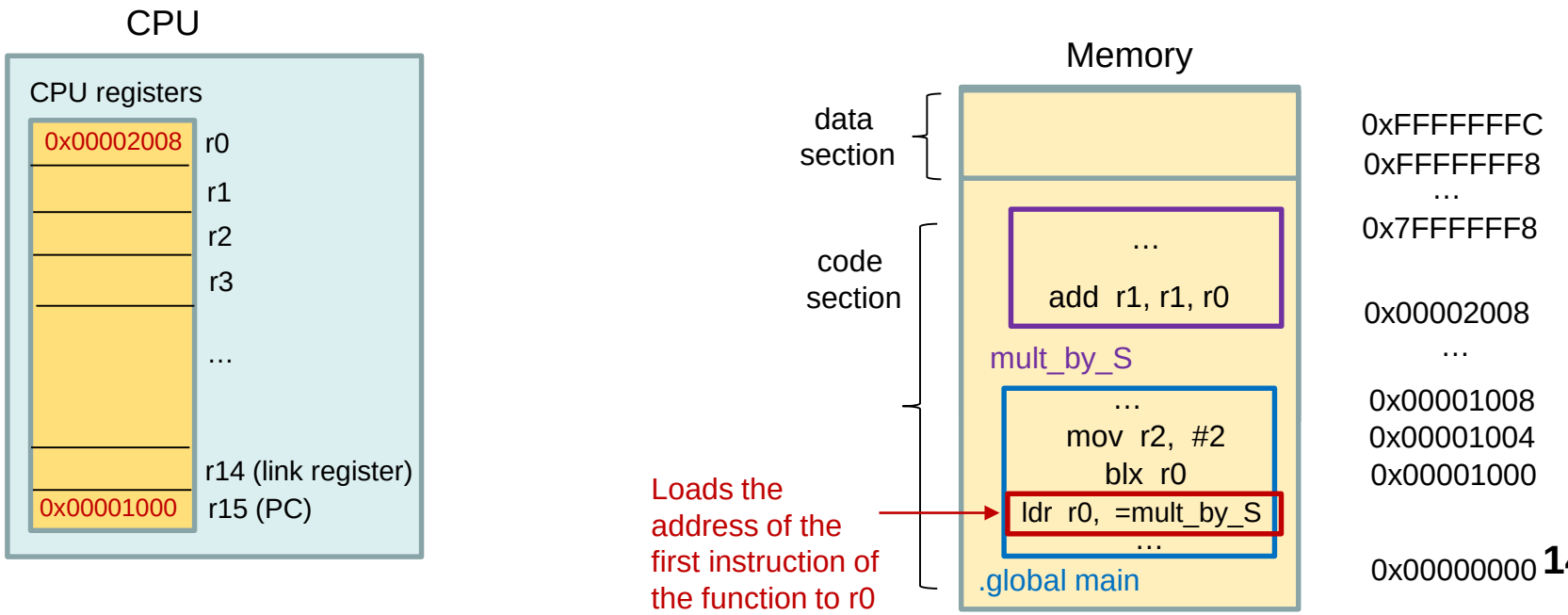
- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call)
  - Approach 2 (indirect call): use the **B**ranch with **L**ink and e**X**change (blx) instruction together with the address of the function stored in a CPU register.

Example: blx r0 ← r0 contains the address of the first instruction of the function

Raspberry Pi Assembler
Calling a function

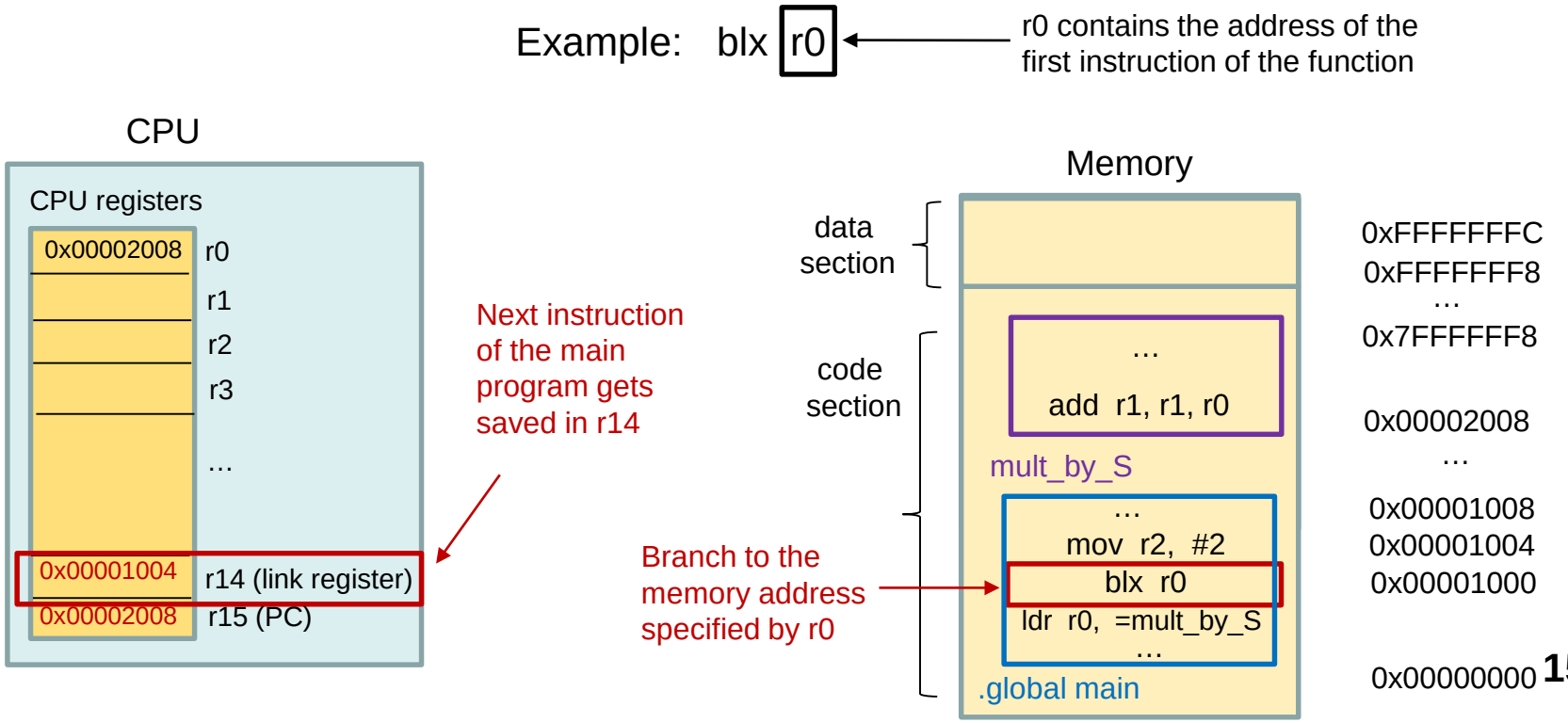
- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call)
  - Approach 2 (indirect call): use the **B**ranch with **L**ink and e**X**change (blx) instruction together with the address of the function stored in a CPU register.

Example: blx r0 ← r0 contains the address of the first instruction of the function



Raspberry Pi Assembler
Calling a function

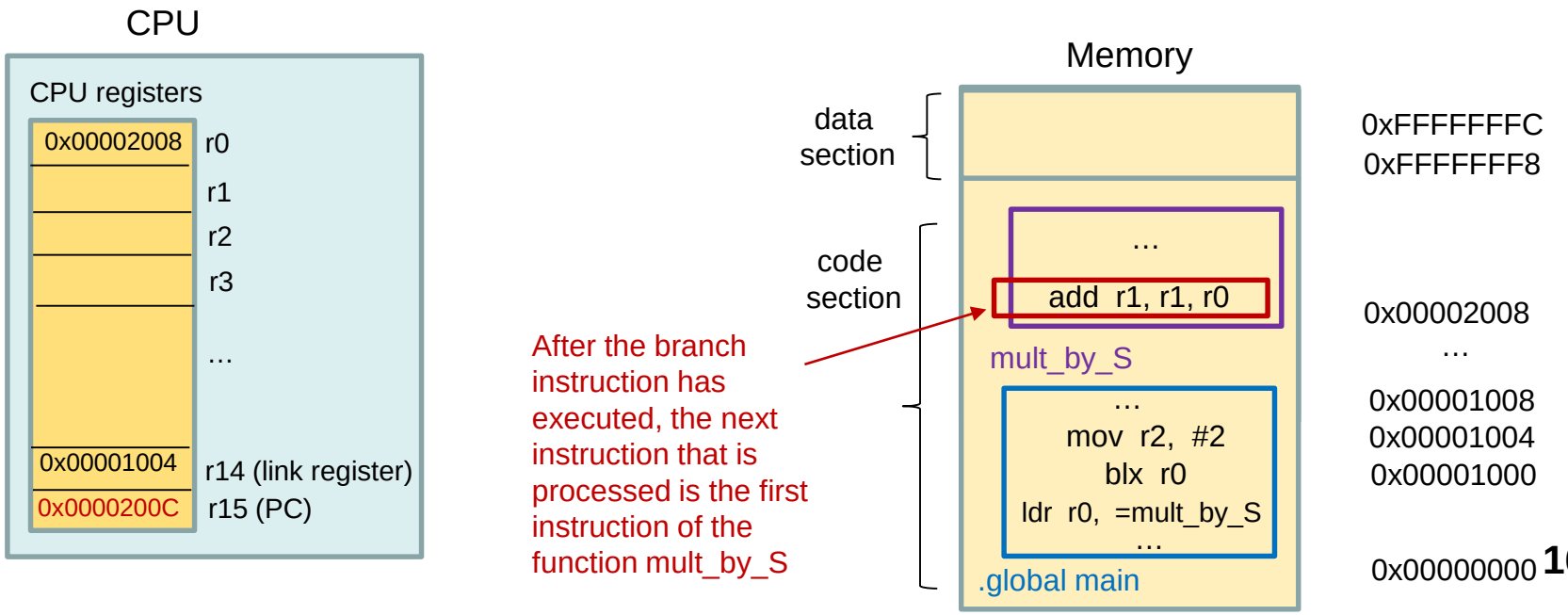
- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call)
  - Approach 2 (indirect call): use the **B**ranch with **L**ink and e**X**change (blx) instruction together with the address of the function stored in a CPU register.



Raspberry Pi Assembler
Calling a function

- How do we call a function from the main program?
- There are two ways to call a function
  - Approach 1 (direct call)
  - Approach 2 (indirect call): use the **B**ranch with **L**ink and e**X**change (blx) instruction together with the address of the function stored in a CPU register.

Example: blx r0 ← r0 contains the address of the first instruction of the function



Leaving a function

Raspberry Pi Assembler
Leaving a function

- How do we leave a function?
  - Firstly, ensure that the current value of the Link Register is the same as the value of the LR at the start of the function

Raspberry Pi Assembler

Leaving a function

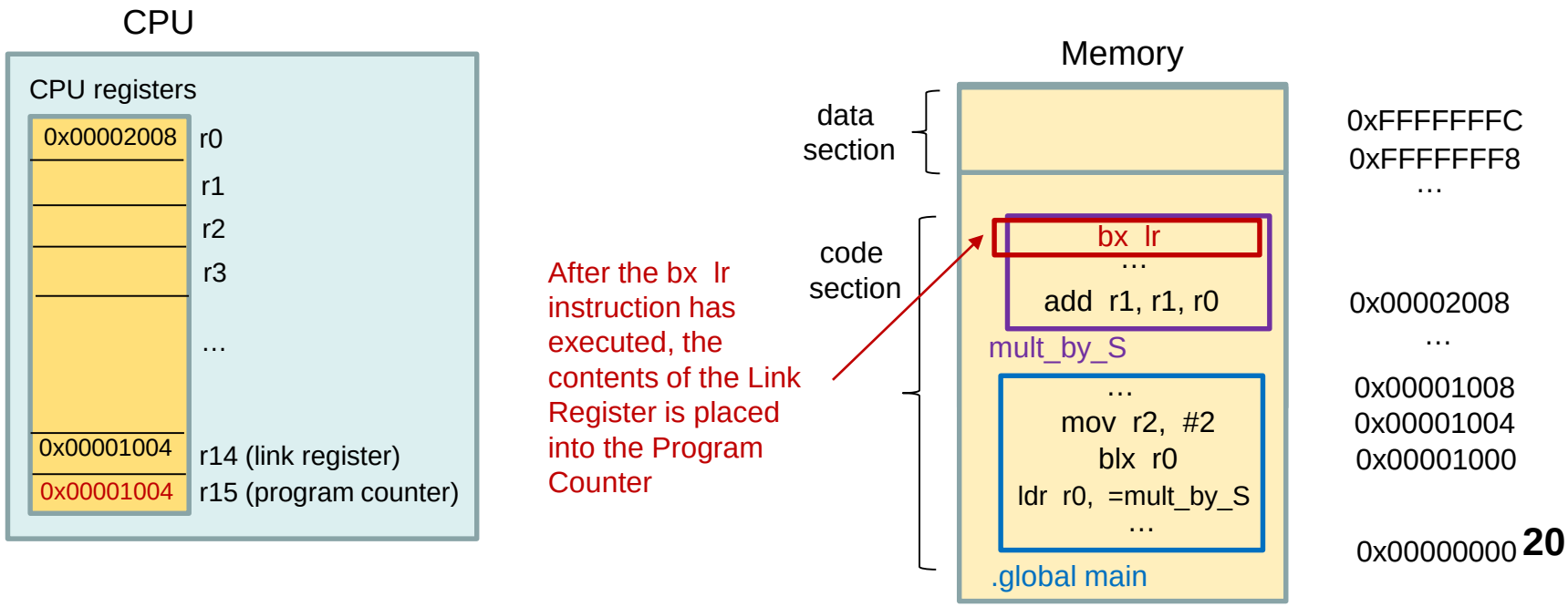
- How do we leave a function?
  - Firstly, ensure that the current value of the Link Register is the same as the value of the LR at the start of the function
  - Use the **B**ranch and e**X**change (bx) together with the link register (lr) to return to the next instruction in the main program, which is after the branch instruction that called the function

Example: bx lr

Raspberry Pi Assembler
Leaving a function

- How do we leave a function?
  - Firstly, ensure that the current value of the Link Register is the same as the value of the LR at the start of the function
  - Use the **B**ranch and e**X**change (bx) together with the link register (lr) to return to the next instruction in the main program, which is after the branch instruction that called the function

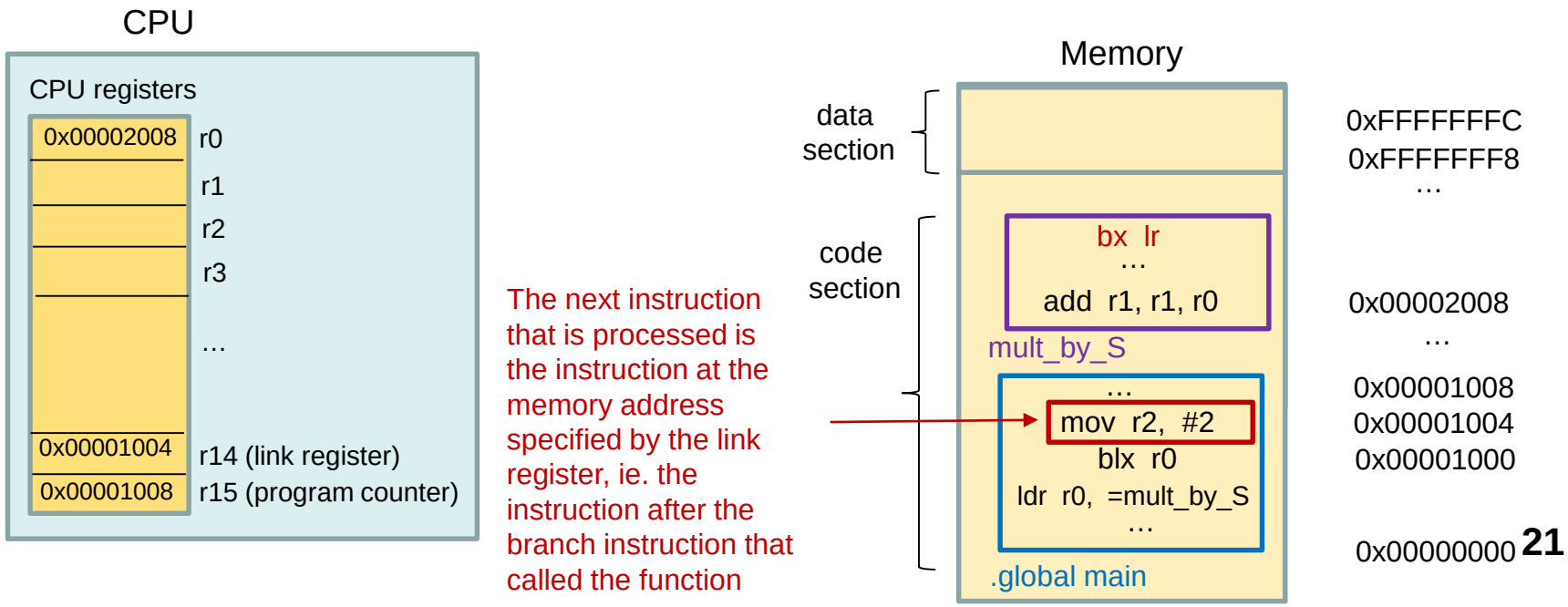
Example: bx lr



Raspberry Pi Assembler
Leaving a function

- How do we leave a function?
  - Firstly, ensure that the current value of the Link Register is the same as the value of the LR at the start of the function
  - Use the **B**ranch and e**X**change (bx) together with the link register (lr) to return to the next instruction in the main program, which is after the branch instruction that called the function

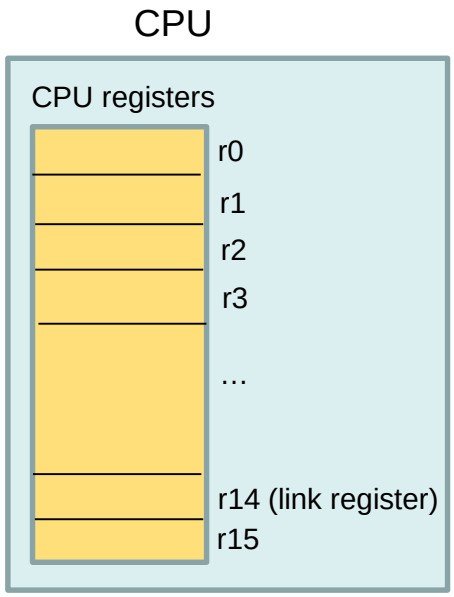
Example: bx lr



Returning data from a function

Raspberry Pi Assembler
Returning data from a function

- How is data returned from a function?
  - This depends on the size of the returned data. If the data is
    - Less than 32-bits, then it is stored in r0
    - Greater than 32-bits but less than 64-bits, then it is stored in r0 and r1
    - Greater than 64-bits, then it is stored in the stack



Example: writing programs

Raspberry Pi Assembler
Writing programs

- Let’s write the following programs
  1. Hello world: Prints the message “Hello World” on the terminal. This program only has a main function. In this program we learn how to store the contents of the Link Register (LR) at the start of the main function. Then, before the main function completes, the LR is restored to the value at the start of the function.

  2. Print back:

  3. First function:

Raspberry Pi Assembler

Writing a program: Hello world

```
 1 /* -- hello01.s */
 2 .data
 3
 4 greeting:
 5  .asciz "Hello world"
 6
 7 .balign 4
 8 return: .word 0
 9
10 .text
11
12 .global main
13 main:
14     ldr r1, =return     @ r1 <- &return
15     str lr, [r1]        @ *r1 <- lr
16
17     ldr r0, =greeting   @ r0 <- &greeting
18                         @ First parameter of puts
19
20     bl  puts            @ Call to puts
21                         @ lr <- address of next instruction
22
23     ldr r1, =return     @ r1 <- &return
24     ldr lr, [r1]        @ lr <- *r1
25     bx  lr              @ return from main
26
27 /* External */
28 .global puts            @ The C function puts
```

- Define a string with the name **greeting**
- The name of the string represents the address of the first character of the string

Raspberry Pi Assembler

Writing a program: Hello world

```
 1 /* -- hello01.s */
 2 .data
 3
 4 greeting:
 5  .asciz "Hello world"
 6
 7 .balign 4
 8 return: .word 0
 9
10 .text
11
12 .global main
13 main:
14     ldr r1, =return     @ r1 <- &return
15     str lr, [r1]        @ *r1 <- lr
16
17     ldr r0, =greeting   @ r0 <- &greeting
18                         @ First parameter of puts
19
20     bl  puts            @ Call to puts
21                         @ lr <- address of next instruction
22
23     ldr r1, =return     @ r1 <- &return
24     ldr lr, [r1]        @ lr <- *r1
25     bx  lr              @ return from main
26
27 /* External */
28 .global puts            @ The C function puts
```

Declare a variable return

- Reserve 32-bits of data for a variable **return** to store the contents of the link register
- Ensure that the variable **return** is aligned to the next memory address that is a multiple of 4

Raspberry Pi Assembler

Writing a program: Hello world

```
 1 /* -- hello01.s */
 2 .data
 3
 4 greeting:
 5  .asciz "Hello world"
 6
 7 .balign 4
 8 return: .word 0
 9
10 .text
11
12 .global main
13 main:
14     ldr r1, =return      @ r1 <- &return
15     str lr, [r1]         @ *r1 <- lr
16
17     ldr r0, =greeting    @ r0 <- &greeting
18                          @ First parameter of puts
19
20     bl  puts             @ Call to puts
21                          @ lr <- address of next instruction
22
23     ldr r1, =return      @ r1 <- &return
24     ldr lr, [r1]         @ lr <- *r1
25     bx  lr               @ return from main
26
27 /* External */
28 .global puts             @ The C function puts
```

Saves the contents of LR to the variable return

- Load the address of the variable **return** into register r1
- Store the contents of the link register to the variable **return**

Raspberry Pi Assembler

Writing a program: Hello world

```
 1 /* -- hello01.s */
 2 .data
 3
 4 greeting:
 5  .asciz "Hello world"
 6
 7 .balign 4
 8 return: .word 0
 9
10 .text
11
12 .global main
13 main:
14     ldr r1, =return      @ r1 <- &return
15     str lr, [r1]         @ *r1 <- lr
16
17     ldr r0, =greeting    @ r0 <- &greeting
18                          @ First parameter of puts
19
20     bl  puts             @ Call to puts
21                          @ lr <- address of next instruction
22
23     ldr r1, =return      @ r1 <- &return
24     ldr lr, [r1]         @ lr <- *r1
25     bx  lr               @ return from main
26
27 /* External */
28 .global puts             @ The C function puts
```

Load the address of the first character of the string **greeting** into r0

Raspberry Pi Assembler

Writing a program: Hello world

```
 1 /* -- hello01.s */
 2 .data
 3
 4 greeting:
 5  .asciz "Hello world"
 6
 7 .balign 4
 8 return: .word 0
 9
10 .text
11
12 .global main
13 main:
14     ldr r1, =return     @ r1 <- &return
15     str lr, [r1]        @ *r1 <- lr
16
17     ldr r0, =greeting   @ r0 <- &greeting
18                         @ First parameter of puts
19
20     bl  puts            @ Call to puts
21                         @ lr <- address of next instruction
22
23     ldr r1, =return     @ r1 <- &return
24     ldr lr, [r1]        @ lr <- *r1
25     bx  lr              @ return from main
26
27 /* External */
28 .global puts            @ The C function puts
```

Call the **puts** functions. This function requires that the address of the first character of the string is in register r0. Thereafter, the string is printed to the terminal

Raspberry Pi Assembler

Writing a program: Hello world

```
 1 /* -- hello01.s */
 2 .data
 3
 4 greeting:
 5  .asciz "Hello world"
 6
 7 .balign 4
 8 return: .word 0
 9
10 .text
11
12 .global main
13 main:
14     ldr r1, =return     @ r1 <- &return
15     str lr, [r1]        @ *r1 <- lr
16
17     ldr r0, =greeting   @ r0 <- &greeting
18                         @ First parameter of puts
19
20     bl  puts            @ Call to puts
21                         @ lr <- address of next instruction
22
23     ldr r1, =return     @ r1 <- &return
24     ldr lr, [r1]        @ lr <- *r1
25     bx  lr              @ return from main
26
27 /* External */
28 .global puts            @ The C function puts
```

Restores the value of the LR

- Loads the address of return to register r1
- Loads the contents of the variable return to the LR

Raspberry Pi Assembler

Writing a program: Hello world

```
 1 /* -- hello01.s */
 2 .data
 3
 4 greeting:
 5  .asciz "Hello world"
 6
 7 .balign 4
 8 return: .word 0
 9
10 .text
11
12 .global main
13 main:
14     ldr r1, =return     @ r1 <- &return
15     str lr, [r1]        @ *r1 <- lr
16
17     ldr r0, =greeting   @ r0 <- &greeting
18                         @ First parameter of puts
19
20     bl  puts            @ Call to puts
21                         @ lr <- address of next instruction
22
23     ldr r1, =return     @ r1 <- &return
24     ldr lr, [r1]        @ lr <- *r1
25     bx  lr              @ return from main
26
27 /* External */
28 .global puts            @ The C function puts
```

Branch to the memory address specified by the LR

Raspberry Pi Assembler
Writing programs

- Let's write the following programs
  1. Hello world: Prints the message "Hello World" on the terminal. This program only has a main function. In this program we learn how to store the contents of the Link Register (LR) at the start of the main function. Then, before the main function completes, the LR is restored to the value at the start of the function.

  2. Print back: Reads a number entered by the user and prints it back to the screen. Also returns the number in the error code to check, for the second time, that everything went as expected. Ensure the number entered is between 0 and 255. Uses both the **printf** and the **scanf** function.

  3. First function:

Raspberry Pi Assembler
Writing programs: Print back

- Let's understand how the **printf** functions operate
  - printf: formats a series of strings and numerical values and builds a string to write to the output stream. The following parameters are required
    - r0 must contain the address of the string
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a
    - r1 must contain the data to be included in the string

Raspberry Pi Assembler

Writing programs: Print back

- Let's understand how the **printf** functions operate
  - printf: formats a series of strings and numerical values and builds a string to write to the output stream. The following parameters are required
    - r0 must contain the address of the string
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a
    - r1 must contain the data to be included in the string

C-language: example of printf

```
int     number = 5


printf("My number is %d\n", number)
```

Raspberry Pi Assembler
Writing programs: Print back

- Let's understand how the **printf** functions operate
  - printf: formats a series of strings and numerical values and builds a string to write to the output stream. The following parameters are required
    - r0 must contain the address of the string
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a
    - r1 must contain the data to be included in the string

C-language: example of printf

```
int     number = 5


printf("My number is %d\n", number)
```

My number is 5

Raspberry Pi Assembler

Writing programs: Print back

- Let's understand how the **printf** functions operate
  - printf: formats a series of strings and numerical values and builds a string to write to the output stream. The following parameters are required
    - r0 must contain the address of the string
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a
    - r1 must contain the data to be included in the string

C-language: example of printf

```
int     number = 5


printf("My number is %d\n", number)
```

My number is 5

Assembly: example of printf

```
message:  .asciz  "My number is %d\n"
number:  .word  5

ldr  r0,  =message
ldr  r1,  =number
ldr  r1,  [r1]

bl  printf
```

Raspberry Pi Assembler
Writing programs: Print back

- Let's understand how the **scanf** operates
  - scanf: reads data from the input stream. The following parameters are required
    - r0 must contain the format of the user entered data
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a whitespace
    - r1 must contain the address of the variable to store user entered data

Raspberry Pi Assembler

Writing programs: Print back

- Let's understand how the **scanf** operates
  - scanf: reads data from the input stream. The following parameters are required
    - r0 must contain the format of the user entered data
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a whitespace
    - r1 must contain the address of the variable to store user entered data

C-language: example of scanf

```
int     number

printf("Enter a number\n")
argsread = scanf("%d", &number)
```

Raspberry Pi Assembler
Writing programs: Print back

- Let's understand how the **scanf** operates
  - scanf: reads data from the input stream. The following parameters are required
    - r0 must contain the format of the user entered data
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a whitespace
    - r1 must contain the address of the variable to store user entered data

C-language: example of scanf

```
int     number

printf("Enter a number\n")
argsread = scanf("%d", &number)
```

stores
- User input into variable **number**,

argsread = number of inputs

Raspberry Pi Assembler

Writing programs: Print back

- Let's understand how the **scanf** operates
  - scanf: reads data from the input stream. The following parameters are required
    - r0 must contain the format of the user entered data
      - %d denotes signed decimal integers
      - %i denotes signed decimal hex numbers
      - %u denotes unsigned decimal numbers
      - %c denotes characters
      - %s denotes a string of characters terminated by a whitespace
    - r1 must contain the address of the variable to store user entered data

C-language: example of scanf

```
int     number

printf("Enter a number\n")
argsread = scanf("%d", &number)
```

Assembly: example of scanf

```
scan_pattern: .asciz  "%d"
number_read:  .word 0

ldr  r0, =scan_pattern
ldr  r1, =number_read

bl  scanf
```

Raspberry Pi Assembler

Writing programs: Print back

```
 1 /* -- printf01.s */
 2 .data
 3
 4 /* First message */
 5 .balign 4
 6 message1: .asciz "Hey, type a number: "
 7
 8 /* Second message */
 9 .balign 4
10 message2: .asciz "I read the number %d\n"
11
12 /* Format pattern for scanf */
13 .balign 4
14 scan_pattern : .asciz "%d"
15
16 /* Where scanf will store the number read */
17 .balign 4
18 number_read: .word 0
19
20 .balign 4
21 return: .word 0
22
```

Defining two strings

- message1
- messsage2

Raspberry Pi Assembler

Writing programs: Print back

```
 1 /* -- printf01.s */
 2 .data
 3
 4 /* First message */
 5 .balign 4
 6 message1: .asciz "Hey, type a number: "
 7
 8 /* Second message */
 9 .balign 4
10 message2: .asciz "I read the number %d\n"
11
12 /* Format pattern for scanf */
13 .balign 4
14 scan_pattern : .asciz "%d"
15
16 /* Where scanf will store the number read */
17 .balign 4
18 number_read: .word 0
19
20 .balign 4
21 return: .word 0
22
```

Define a variable **scan_pattern** to save the format of the user entered data for scanf function

- %d indicates user will enter signed integers

Raspberry Pi Assembler

Writing programs: Print back

```
 1 /* -- printf01.s */
 2 .data
 3
 4 /* First message */
 5 .balign 4
 6 message1: .asciz "Hey, type a number: "
 7
 8 /* Second message */
 9 .balign 4
10 message2: .asciz "I read the number %d\n"
11
12 /* Format pattern for scanf */
13 .balign 4
14 scan_pattern : .asciz "%d"
15
16 /* Where scanf will store the number read */
17 .balign 4
18 number_read: .word 0
19
20 .balign 4
21 return: .word 0
22
```

Define a variable **number_read** to hold the contents of the user entered data

Raspberry Pi Assembler

Writing programs: Print back

```
 1 /* -- printf01.s */
 2 .data
 3
 4 /* First message */
 5 .balign 4
 6 message1: .asciz "Hey, type a number: "
 7
 8 /* Second message */
 9 .balign 4
10 message2: .asciz "I read the number %d\n"
11
12 /* Format pattern for scanf */
13 .balign 4
14 scan_pattern : .asciz "%d"
15
16 /* Where scanf will store the number read */
17 .balign 4
18 number_read: .word 0
19
20 .balign 4
21 return: .word 0
22
```

Define a variable **return** to save the contents of the Link Register at the start of the function, so it can be restored to this value before the main function exits

Raspberry Pi Assembler

Writing programs: Print back

```
25 .global main
26 main:
27     ldr r1, =return             @ r1 <- &return
28     str lr, [r1]                @ *r1 <- lr ; save return address
29
30     ldr r0, =message1           @ r0 <- &message1
31     bl  printf                  @ call to printf
32
33     ldr r0, =scan_pattern       @ r0 <- &scan_pattern
34     ldr r1, =number_read        @ r1 <- &number_read
35     bl  scanf                   @ call to scanf
36
37     ldr r0, =message2           @ r0 <- &message2
38     ldr r1, =number_read        @ r1 <- &number_read
39     ldr r1, [r1]                @ r1 <- *r1
40     bl  printf                  @ call to printf
41
42     ldr r0, =number_read        @ r0 <- &number_read
43     ldr r0, [r0]                @ r0 <- *r0
44
45     ldr lr, =return             @ lr <- &return
46     ldr lr, [lr]                @ lr <- *lr
47     bx  lr                      @ return from main using lr
48
49 /* External */
50 .global printf
51 .global scanf
```

Store the contents of the Link Register in the variable **return**

Note: we need to restore the LR to this value before exiting the main function

Raspberry Pi Assembler

Writing programs: Print back

```
25 .global main
26 main:
27     ldr r1, =return             @ r1 <- &return
28     str lr, [r1]                @ *r1 <- lr ; save return address
29
30     ldr r0, =message1           @ r0 <- &message1
31     bl  printf                  @ call to printf
32
33     ldr r0, =scan_pattern       @ r0 <- &scan_pattern
34     ldr r1, =number_read        @ r1 <- &number_read
35     bl  scanf                   @ call to scanf
36
37     ldr r0, =message2           @ r0 <- &message2
38     ldr r1, =number_read        @ r1 <- &number_read
39     ldr r1, [r1]                @ r1 <- *r1
40     bl  printf                  @ call to printf
41
42     ldr r0, =number_read        @ r0 <- &number_read
43     ldr r0, [r0]                @ r0 <- *r0
44
45     ldr lr, =return             @ lr <- &return
46     ldr lr, [lr]                @ lr <- *lr
47     bx  lr                      @ return from main using lr
48
49 /* External */
50 .global printf
51 .global scanf
```

prepare the parameters for printf instruction:

- r0 must be the address of the first character of the string

Raspberry Pi Assembler

Writing programs: Print back

```
25 .global main
26 main:
27     ldr r1, =return             @ r1 <- &return
28     str lr, [r1]                @ *r1 <- lr ; save return address
29
30     ldr r0, =message1           @ r0 <- &message1
31     bl  printf                  @ call to printf
32
33     ldr r0, =scan_pattern       @ r0 <- &scan_pattern
34     ldr r1, =number_read        @ r1 <- &number_read
35     bl  scanf                   @ call to scanf
36
37     ldr r0, =message2           @ r0 <- &message2
38     ldr r1, =number_read        @ r1 <- &number_read
39     ldr r1, [r1]                @ r1 <- *r1
40     bl  printf                  @ call to printf
41
42     ldr r0, =number_read        @ r0 <- &number_read
43     ldr r0, [r0]                @ r0 <- *r0
44
45     ldr lr, =return             @ lr <- &return
46     ldr lr, [lr]                @ lr <- *lr
47     bx  lr                      @ return from main using lr
48
49 /* External */
50 .global printf
51 .global scanf
```

prepare the parameters for scanf instruction:

- r0 must be the format of the user entered data
- r1 must be the address of the variable to store the user entered data

Raspberry Pi Assembler

Writing programs: Print back

```
25 .global main
26 main:
27     ldr r1, =return             @ r1 <- &return
28     str lr, [r1]                @ *r1 <- lr ; save return address
29
30     ldr r0, =message1           @ r0 <- &message1
31     bl  printf                  @ call to printf
32
33     ldr r0, =scan_pattern       @ r0 <- &scan_pattern
34     ldr r1, =number_read        @ r1 <- &number_read
35     bl  scanf                   @ call to scanf
36
37     ldr r0, =message2           @ r0 <- &message2
38     ldr r1, =number_read        @ r1 <- &number_read
39     ldr r1, [r1]                @ r1 <- *r1
40     bl  printf                  @ call to printf
41
42     ldr r0, =number_read        @ r0 <- &number_read
43     ldr r0, [r0]                @ r0 <- *r0
44
45     ldr lr, =return             @ lr <- &return
46     ldr lr, [lr]                @ lr <- *lr
47     bx  lr                      @ return from main using lr
48
49 /* External */
50 .global printf
51 .global scanf
```

prepare the parameters for printf instruction:

- r0 must be the address of the first character of the string
- r1 must be the data to be included in the string

Raspberry Pi Assembler

Writing programs: Print back

```
25 .global main
26 main:
27     ldr r1, =return             @ r1 <- &return
28     str lr, [r1]                @ *r1 <- lr ; save return address
29
30     ldr r0, =message1           @ r0 <- &message1
31     bl  printf                  @ call to printf
32
33     ldr r0, =scan_pattern       @ r0 <- &scan_pattern
34     ldr r1, =number_read        @ r1 <- &number_read
35     bl  scanf                   @ call to scanf
36
37     ldr r0, =message2           @ r0 <- &message2
38     ldr r1, =number_read        @ r1 <- &number_read
39     ldr r1, [r1]                @ r1 <- *r1
40     bl  printf                  @ call to printf
41
42     ldr r0, =number_read        @ r0 <- &number_read
43     ldr r0, [r0]                @ r0 <- *r0
44
45     ldr lr, =return             @ lr <- &return
46     ldr lr, [lr]                @ lr <- *lr
47     bx  lr                      @ return from main using lr
48
49 /* External */
50 .global printf
51 .global scanf
```

transfer the number read from the user to register r0, so that this value can be displayed to the user when the echo $? command is executed

Raspberry Pi Assembler

Writing programs: Print back

```
25 .global main
26 main:
27     ldr r1, =return             @ r1 <- &return
28     str lr, [r1]                @ *r1 <- lr ; save return address
29
30     ldr r0, =message1           @ r0 <- &message1
31     bl  printf                  @ call to printf
32
33     ldr r0, =scan_pattern       @ r0 <- &scan_pattern
34     ldr r1, =number_read        @ r1 <- &number_read
35     bl  scanf                   @ call to scanf
36
37     ldr r0, =message2           @ r0 <- &message2
38     ldr r1, =number_read        @ r1 <- &number_read
39     ldr r1, [r1]                @ r1 <- *r1
40     bl  printf                  @ call to printf
41
42     ldr r0, =number_read        @ r0 <- &number_read
43     ldr r0, [r0]                @ r0 <- *r0
44
45     ldr lr, =return             @ lr <- &return
46     ldr lr, [lr]                @ lr <- *lr
47     bx  lr                      @ return from main using lr
48
49 /* External */
50 .global printf
51 .global scanf
```

restore the link register to the value that it was at the start of the main function

Raspberry Pi Assembler

Writing programs: Print back

```
25 .global main
26 main:
27     ldr r1, =return             @ r1 <- &return
28     str lr, [r1]                @ *r1 <- lr ; save return address
29
30     ldr r0, =message1           @ r0 <- &message1
31     bl  printf                  @ call to printf
32
33     ldr r0, =scan_pattern       @ r0 <- &scan_pattern
34     ldr r1, =number_read        @ r1 <- &number_read
35     bl  scanf                   @ call to scanf
36
37     ldr r0, =message2           @ r0 <- &message2
38     ldr r1, =number_read        @ r1 <- &number_read
39     ldr r1, [r1]                @ r1 <- *r1
40     bl  printf                  @ call to printf
41
42     ldr r0, =number_read        @ r0 <- &number_read
43     ldr r0, [r0]                @ r0 <- *r0
44
45     ldr lr, =return             @ lr <- &return
46     ldr lr, [lr]                @ lr <- *lr
47     bx  lr                      @ return from main using lr
48
49 /* External */
50 .global printf
51 .global scanf
```

return from main

Raspberry Pi Assembler
Writing programs: Print back

- Run the program

```
$ ./printf01 ; echo $?
Hey, type a number: 124<CR>
I read the number 124
124
```

Raspberry Pi Assembler
Writing programs

- Let's write the following programs
  1. Hello world: Prints the message "Hello World" on the terminal. This program only has a main function. In this program we learn how to store the contents of the Link Register (LR) at the start of the main function. Then, before the main function completes, the LR is restored to the value at the start of the function.

  2. Print back: Reads a number entered by the user and prints it back to the screen. Also returns the number in the error code to check, for the second time, that everything went as expected. Ensure the number entered is between 0 and 255. Uses both the **printf** and the **scanf** function.

  3. First function: extend the "Print back" program by multiplying the number by 5 and display to the user.

Raspberry Pi Assembler

Writing programs: First Function

- Develop a function to multiply a number by 5
  - Input parameter, r0: number to multiply
  - Output, r0: number multiplied by 5

```
14 .balign 4
15 return2: .word 0
16
17 .text
18
19 /* mult_by_5 function */
20 mult_by_5:
21     ldr r1, =return2            @ r1 <- &return2
22     str lr, [r1]                @ *r1 <- lr
23
24     add r0, r0, r0, LSL #2      @ r0 <- r0 + 4*r0
25
26     ldr lr, =return2            @ lr <- &return2
27     ldr lr, [lr]                @ lr <- *lr
28     bx  lr                      @ return to main using lr
```

store the contents of the Link Register in the variable **return2**

Note: we need to restore the LR to this value before exiting the main function

Raspberry Pi Assembler

Writing programs: First Function

- Develop a function to multiply a number by 5
  - Input parameter, r0: number to multiply
  - Output, r0: number multiplied by 5

```
14 .balign 4
15 return2: .word 0
16
17 .text
18
19 /* mult_by_5 function */
20 mult_by_5:
21     ldr r1, =return2            @ r1 <- &return2
22     str lr, [r1]                @ *r1 <- lr
23
24     add r0, r0, r0, LSL #2      @ r0 <- r0 + 4*r0
25
26     ldr lr, =return2            @ lr <- &return2
27     ldr lr, [lr]                @ lr <- *lr
28     bx  lr                      @ return to main using lr
```

multiply the contents of r0 by 5 and put the result into r0

Raspberry Pi Assembler

Writing programs: First Function

- Develop a function to multiply a number by 5
  - Input parameter, r0: number to multiply
  - Output, r0: number multiplied by 5

```
14 .balign 4
15 return2: .word 0
16
17 .text
18
19 /* mult_by_5 function */
20 mult_by_5:
21     ldr r1, =return2            @ r1 <- &return2
22     str lr, [r1]                @ *r1 <- lr
23
24     add r0, r0, r0, LSL #2      @ r0 <- r0 + 4*r0
25
26     ldr lr, =return2            @ lr <- &return2
27     ldr lr, [lr]                @ lr <- *lr
28     bx  lr                      @ return to main using lr
```

restore the link register to the value that it was at the start of the main function

Raspberry Pi Assembler
Writing programs: First Function

- Develop a function to multiply a number by 5
  - Input parameter, r0: number to multiply
  - Output, r0: number multiplied by 5

```
14 .balign 4
15 return2: .word 0
16
17 .text
18
19 /* mult_by_5 function */
20 mult_by_5:
21     ldr r1, =return2            @ r1 <- &return2
22     str lr, [r1]                @ *r1 <- lr
23
24     add r0, r0, r0, LSL #2      @ r0 <- r0 + 4*r0
25
26     ldr lr, =return2            @ lr <- &return2
27     ldr lr, [lr]                @ lr <- *lr
28     bx  lr                      @ return to main using lr
```

- put the contents of the Link Register into the PC
- The next instruction that will execute is the instruction after the branch instruction that called the function in the main program

Raspberry Pi Assembler
Writing programs: First Function

- First function code

```
 1 /* -- printf02.s */
 2 .data
 3
 4 .balign 4                 @ First message
 5 message1: .asciz "Hey, type a number: "
 6 .balign 4                 @ Second message
 7 message2: .asciz "%d times 5 is %d\n"
 8 .balign 4                 @ Format pattern for scanf
 9 scan_pattern: .asciz "%d"
10 .balign 4                 @ Where scanf will store the number read
11 number_read: .word 0
12 .balign 4
13 return: .word 0
14 .balign 4
15 return2: .word 0
```

Raspberry Pi Assembler

Writing programs: First Function

- First function code: function not shown

```
30 .global main
31 main:
32     ldr r1, =return              @ r1 <- &return
33     str lr, [r1]                 @ *r1 <- lr
34
35     ldr r0, =message1            @ r0 <- &message1
36     bl  printf                   @ call to printf
37
38     ldr r0, =scan_pattern        @ r0 <- &scan_pattern
39     ldr r1, =number_read         @ r1 <- &number_read
40     bl  scanf                    @ call to scanf
41
42     ldr r0, =number_read         @ r0 <- &number_read
43     ldr r0, [r0]                 @ r0 <- *r0
44     bl  mult_by_5
45
46     mov r2, r0                   @ r2 <- r0
47     ldr r1, =number_read         @ r1 <- &number_read
48     ldr r1, [r1]                 @ r1 <- *r1
49     ldr r0, =message2            @ r0 <- &message2
50     bl  printf                   @ call to printf
51
52     ldr lr, =return              @ lr <- &return
53     ldr lr, [lr]                 @ lr <- *lr
54     bx  lr                       @ return from main using lr
55
56 /* External */
57 .global printf
58 .global scanf
```

- calls the function mult_by_5
- Before the function executes, r0 has the value entered by the user
- After the function executes, r0 has the value entered by the user multiplied by 5

Contents of r0 gets overwritten

Raspberry Pi Assembler
Writing programs: First Function

- Result obtained when the code is run

```
$ ./printf02
Hey, type a number: 1234<CR>
1234 times 5 is 6170
```