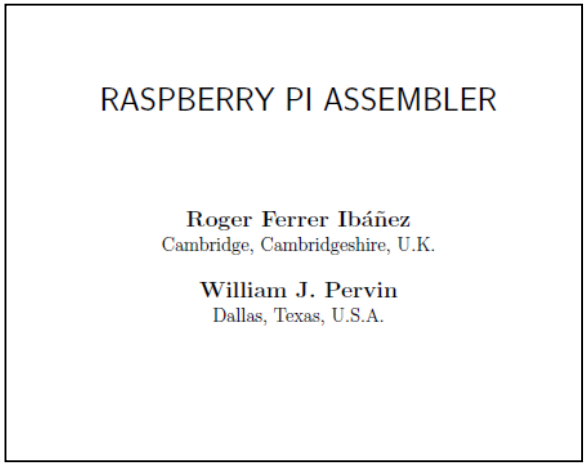
Raspberry Pi Assembler
The Stack

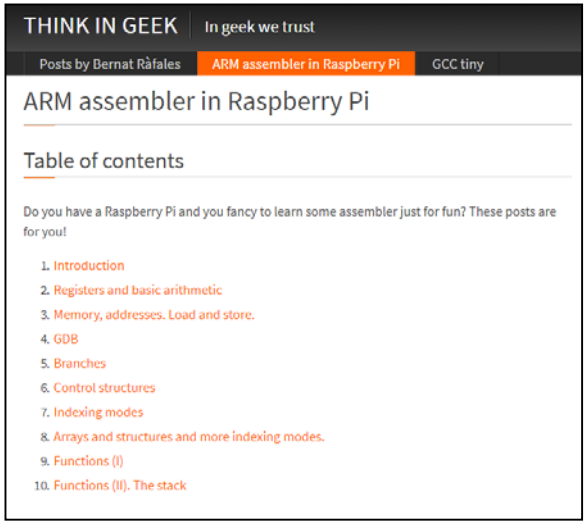

RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

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

Chapter 11: Raspberry Pi Assembler
"Raspberry Pi Assembler" by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/


THINK IN GEEK In geek we trust

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

ARM assembler in Raspberry Pi

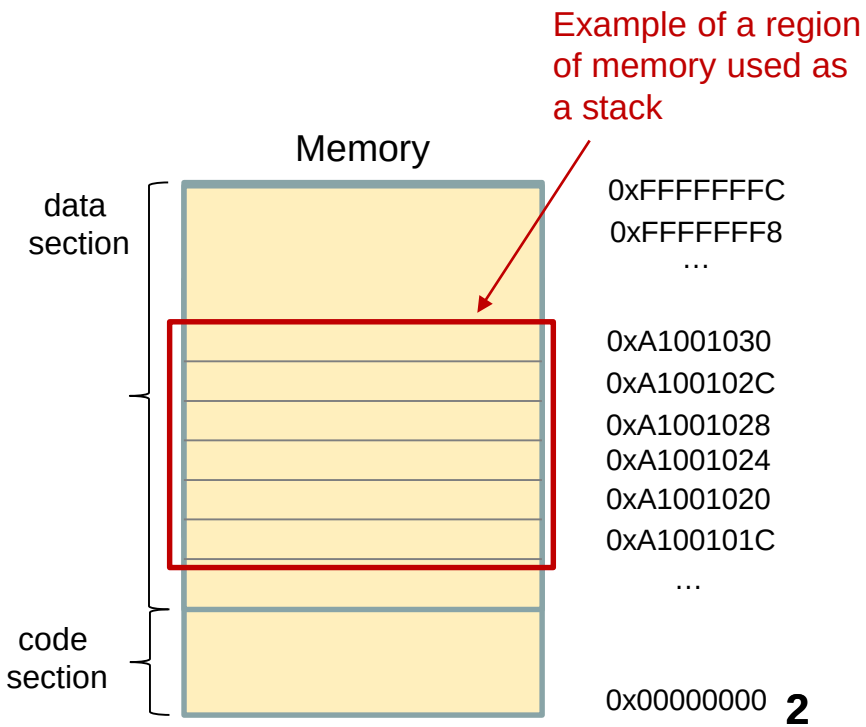
Table of contents

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

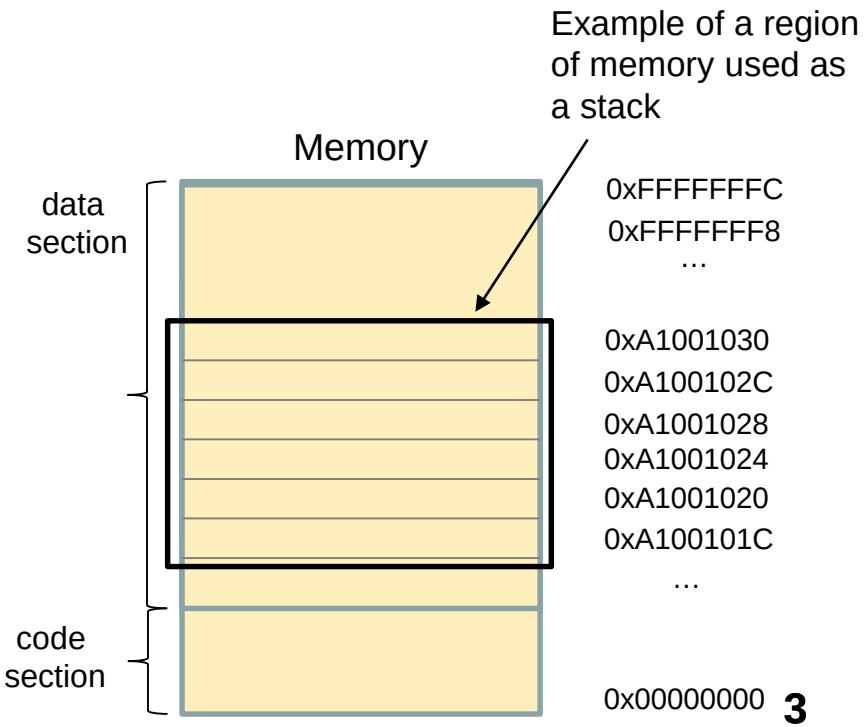
Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data



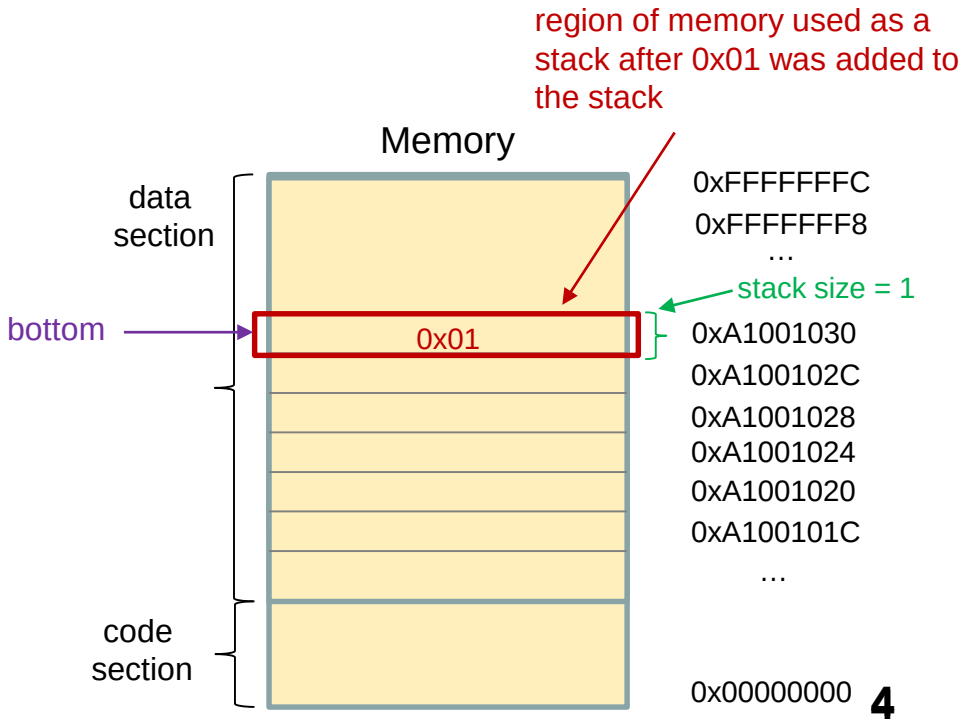
Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer



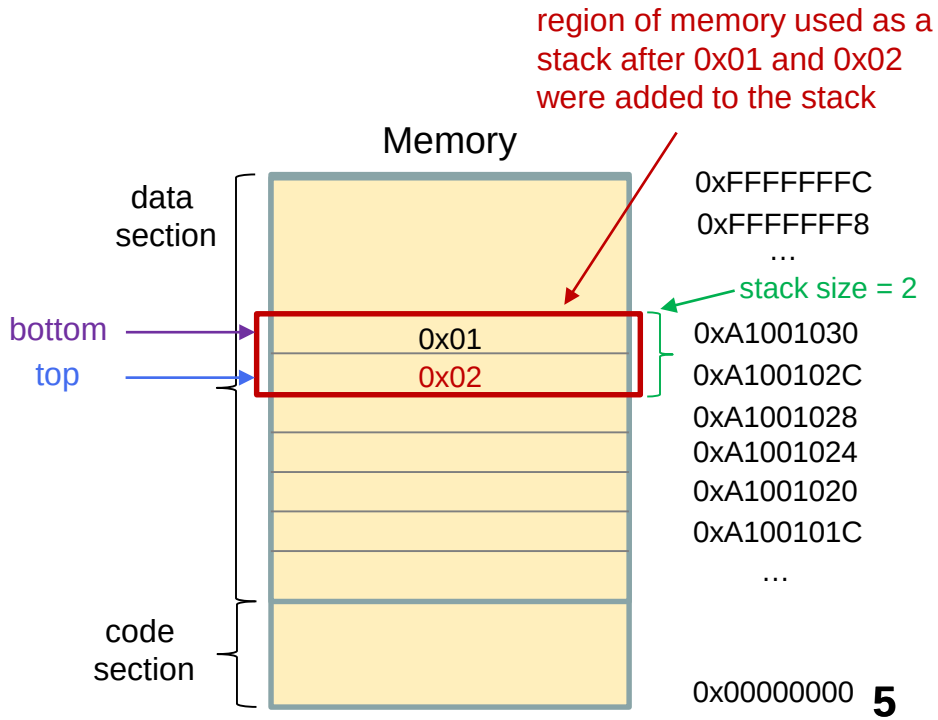
Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack.



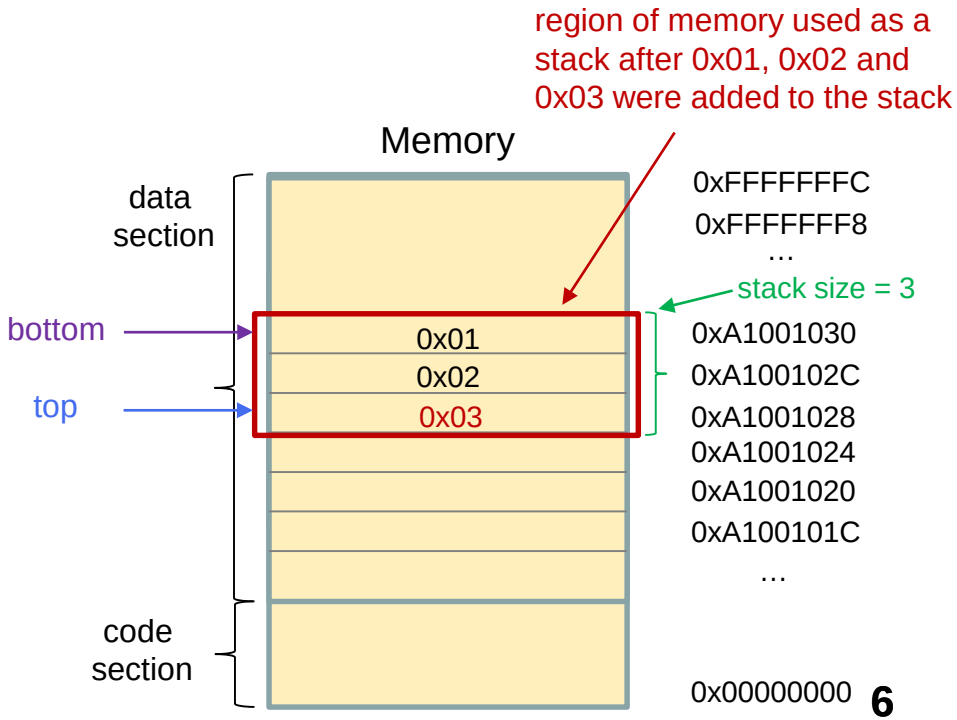
Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack.



Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.



Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.

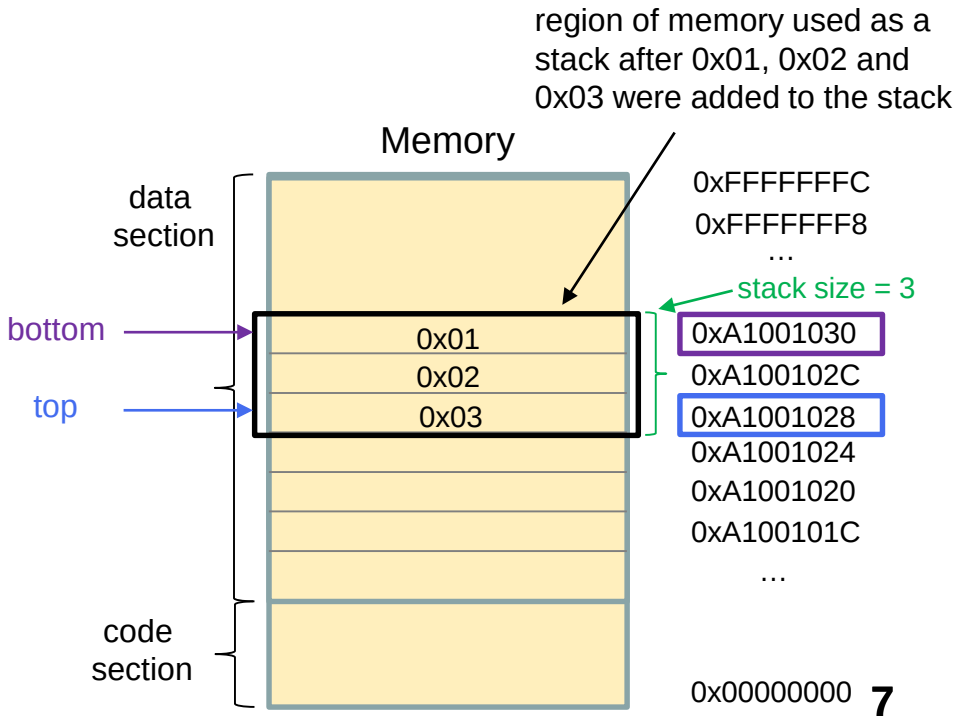
Observations of the stack

• The stack is not a fixed size. As data is added to the stack, it grows in size

•The bottom of the stack corresponds to the first data added to the stack and has the biggest memory address of the stack

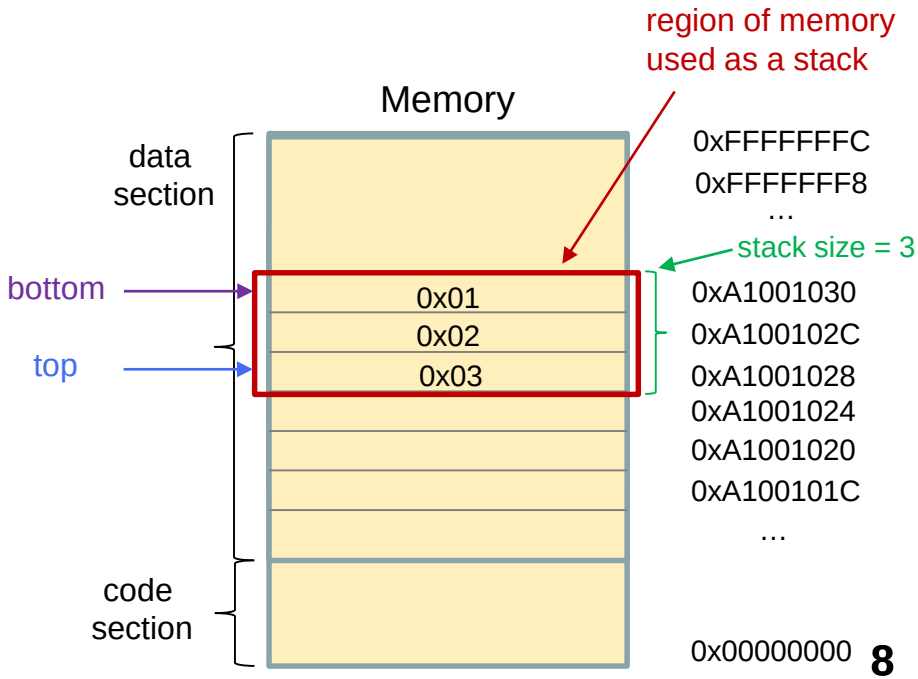
• The top of the stack corresponds to the last data added to the stack and has the smallest memory address

• When adding new data, it is placed at the top of the existing stack



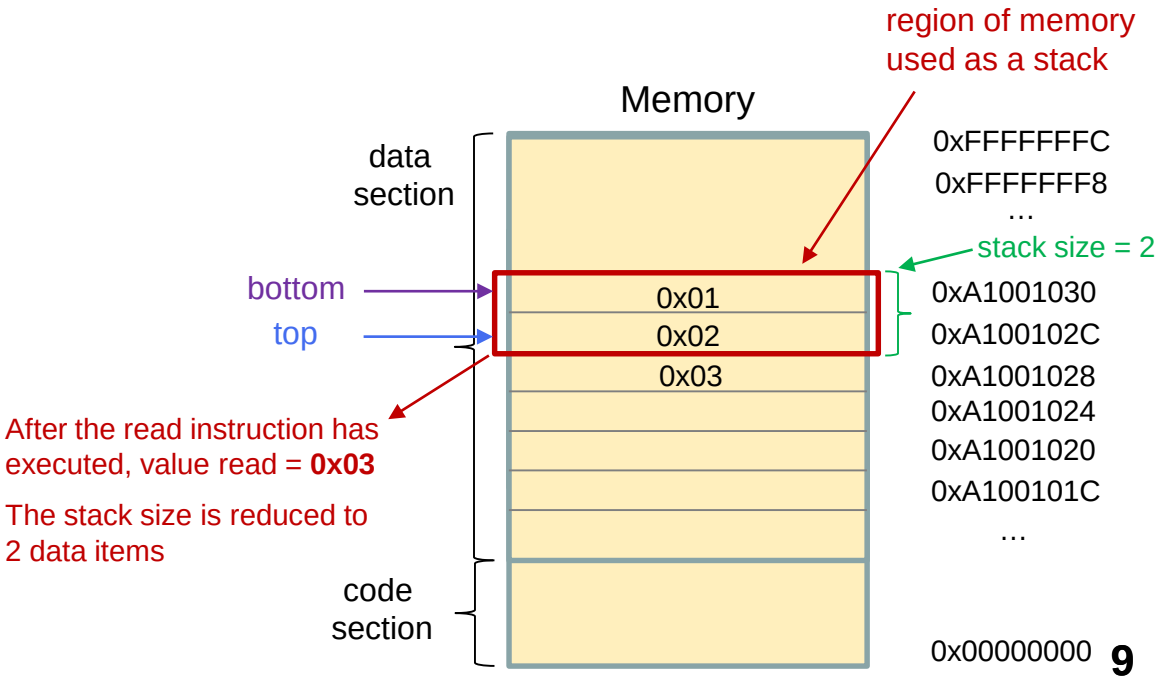
Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.
    - Reading data from the stack:



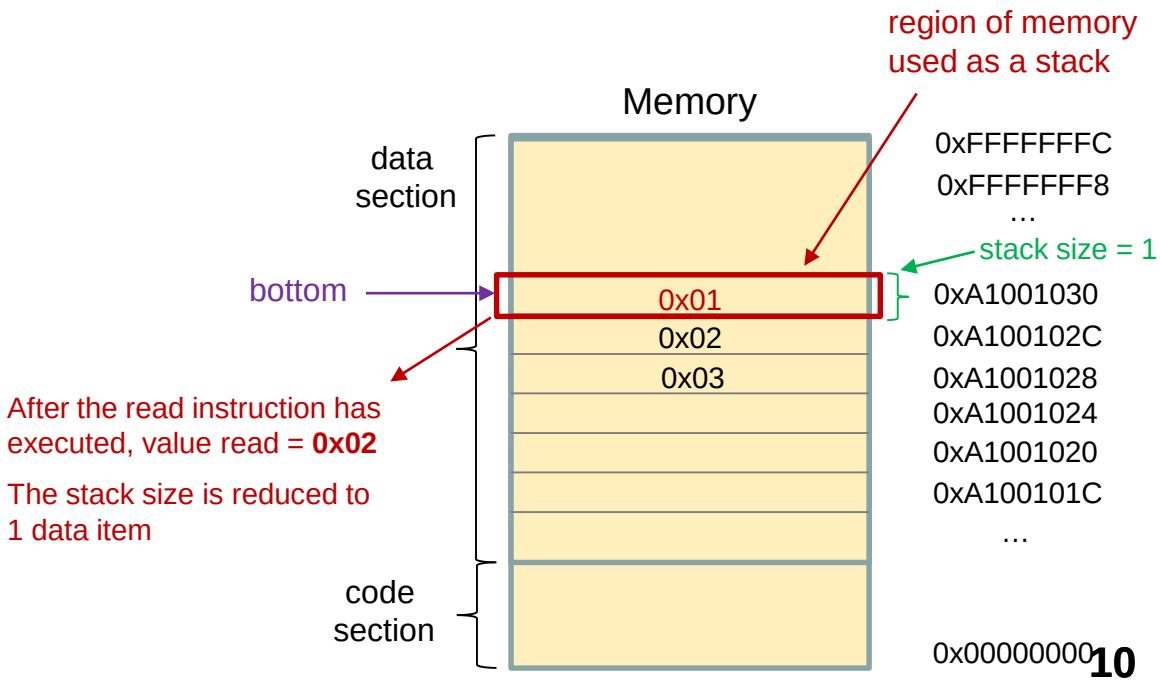
Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.
    - Reading data from the stack: read one data item from the stack.



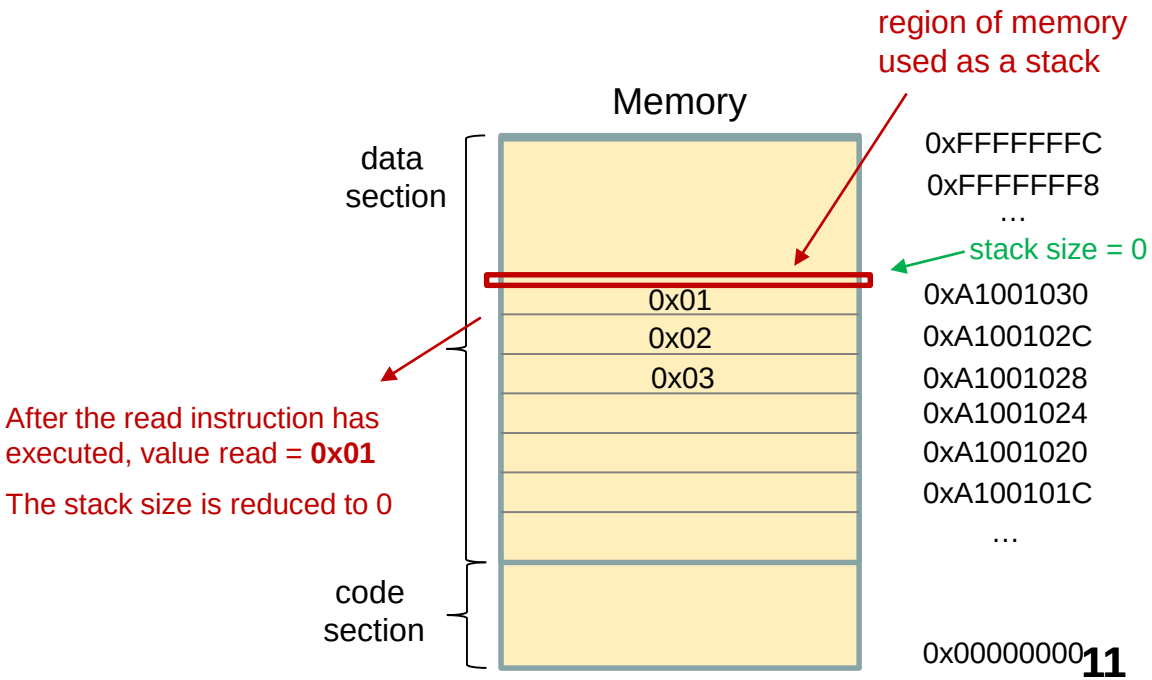
Raspberry Pi Assembler
The Stack

- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.
    - Reading data from the stack: read one data item from the stack. Read another data item from the stack.



Raspberry Pi Assembler
The Stack

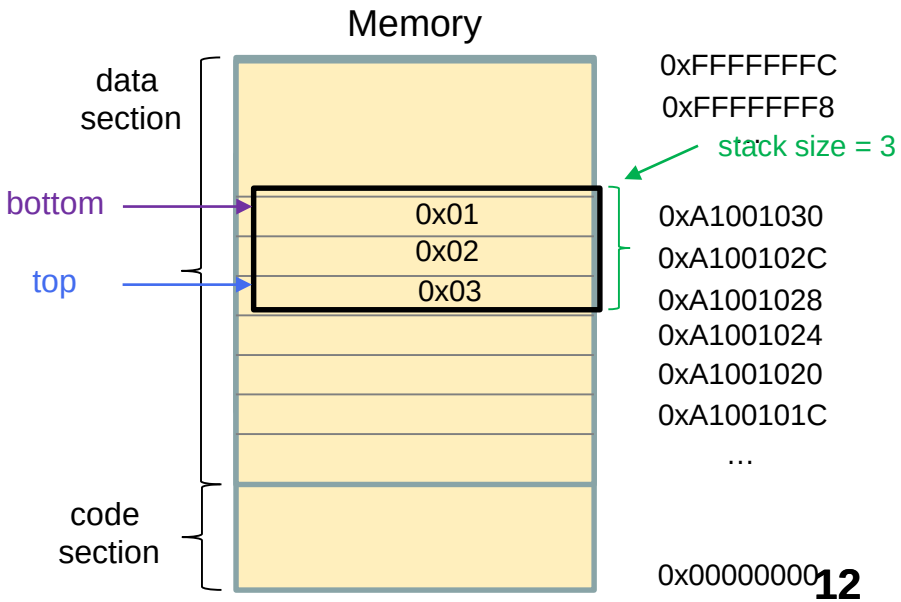
- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.
    - Reading data from the stack: read one data item from the stack. Read another data item from the stack. Read another data item from the stack.



Raspberry Pi Assembler
The Stack

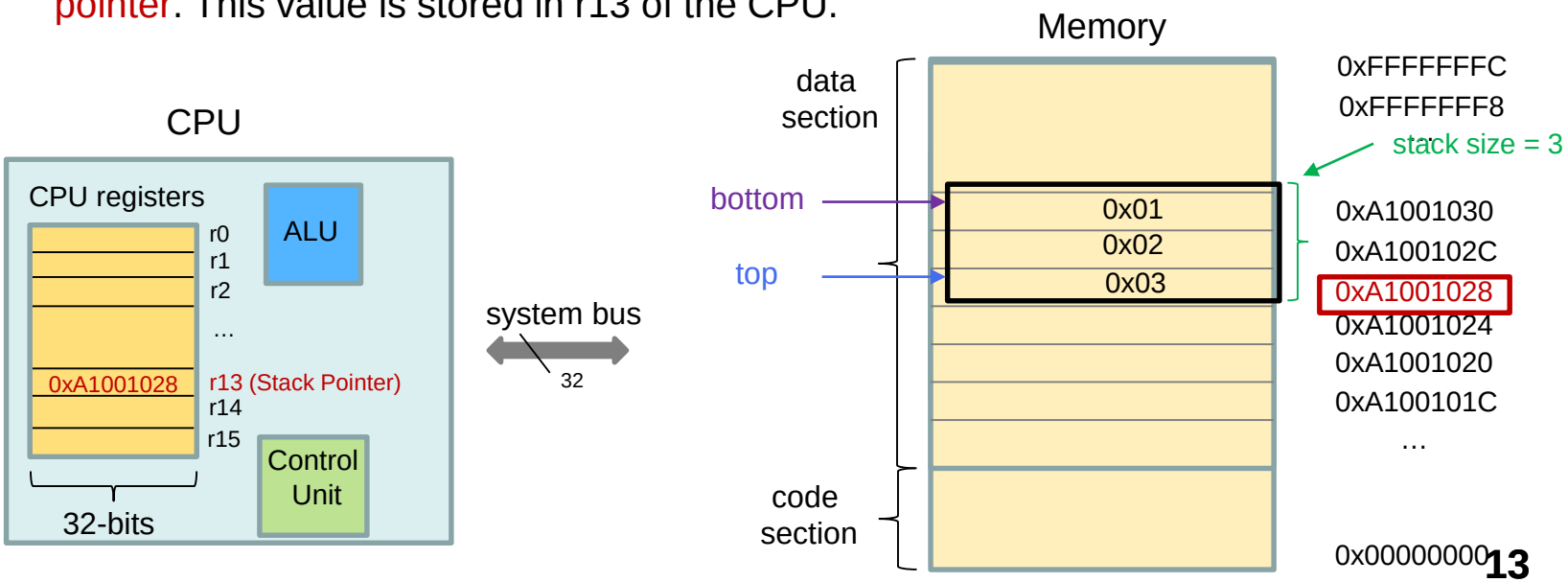
- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.
    - Reading data from the stack: read one data item from the stack. Read another data item from the stack. Read another data item from the stack.

Observations: reading from the stack
• The data item at the top of the stack is read
• The data item remains in memory, however the size of the stack decreases



Raspberry Pi Assembler
The Stack

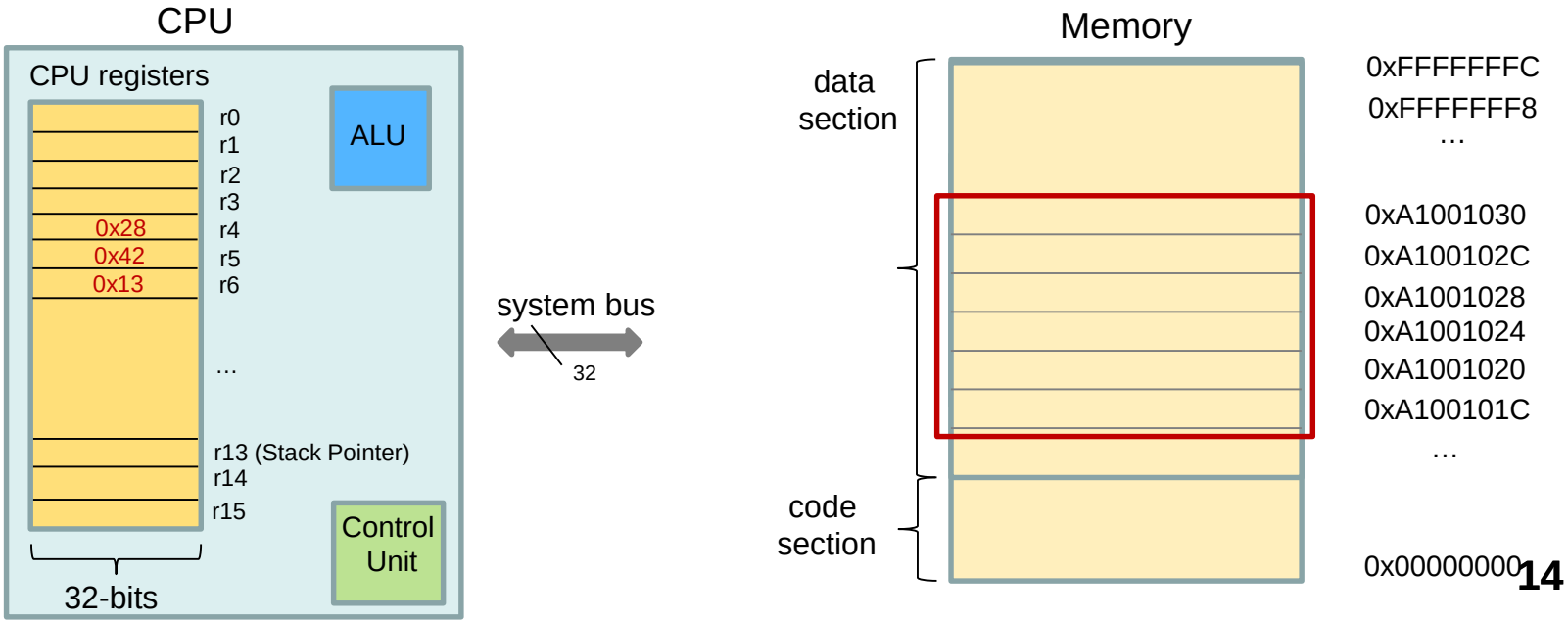
- What is a stack?
  - A stack is a region of memory used by functions to store temporary data
  - Data is stored using a Last-in-first-out (LIFO) buffer
    - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to the stack.
    - Reading data from the stack: read one data item from the stack. Read another data item from the stack. Read another data item from the stack.
    - The memory address of the last value stored in the stack is referred to as the stack pointer. This value is stored in r13 of the CPU.



Raspberry Pi Assembler

The Stack

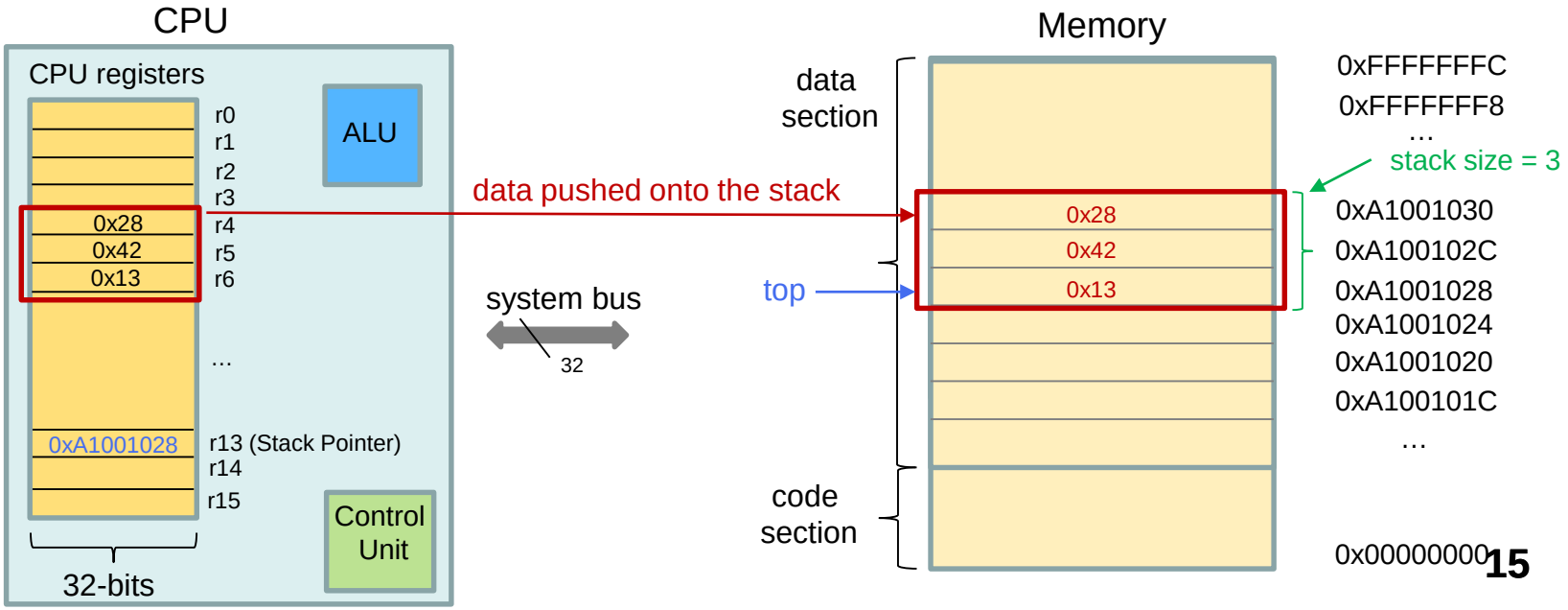
- Why is a stack important?
  - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 – r11, however, before the function exits, r4 – r11 needs to be restored to their initial values at the start of the function. The following process is used to achieve this:



Raspberry Pi Assembler

The Stack

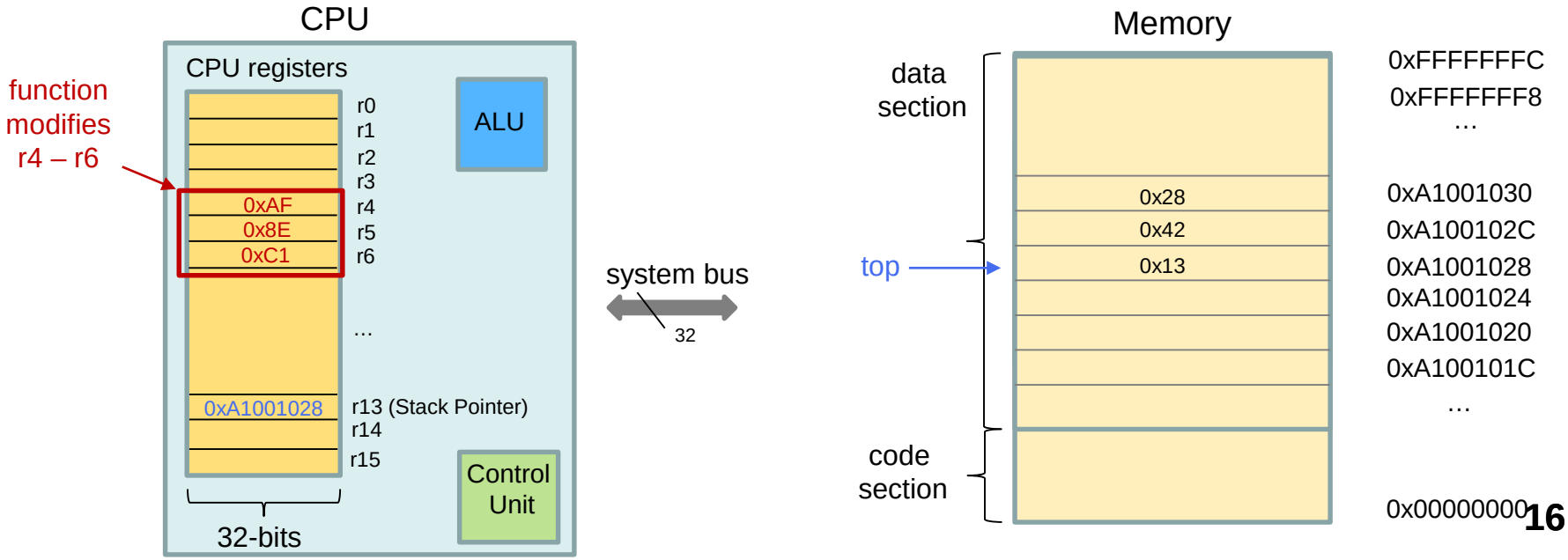
- Why is a stack important?
  - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 – r11, however, before the function exits, r4 – r11 needs to be restored to their initial values at the start of the function. The following process is used to achieve this:
    - At the start of the function, the contents of r4 – r11 are saved on the stack. This is referred to as pushing data onto the stack.



Raspberry Pi Assembler

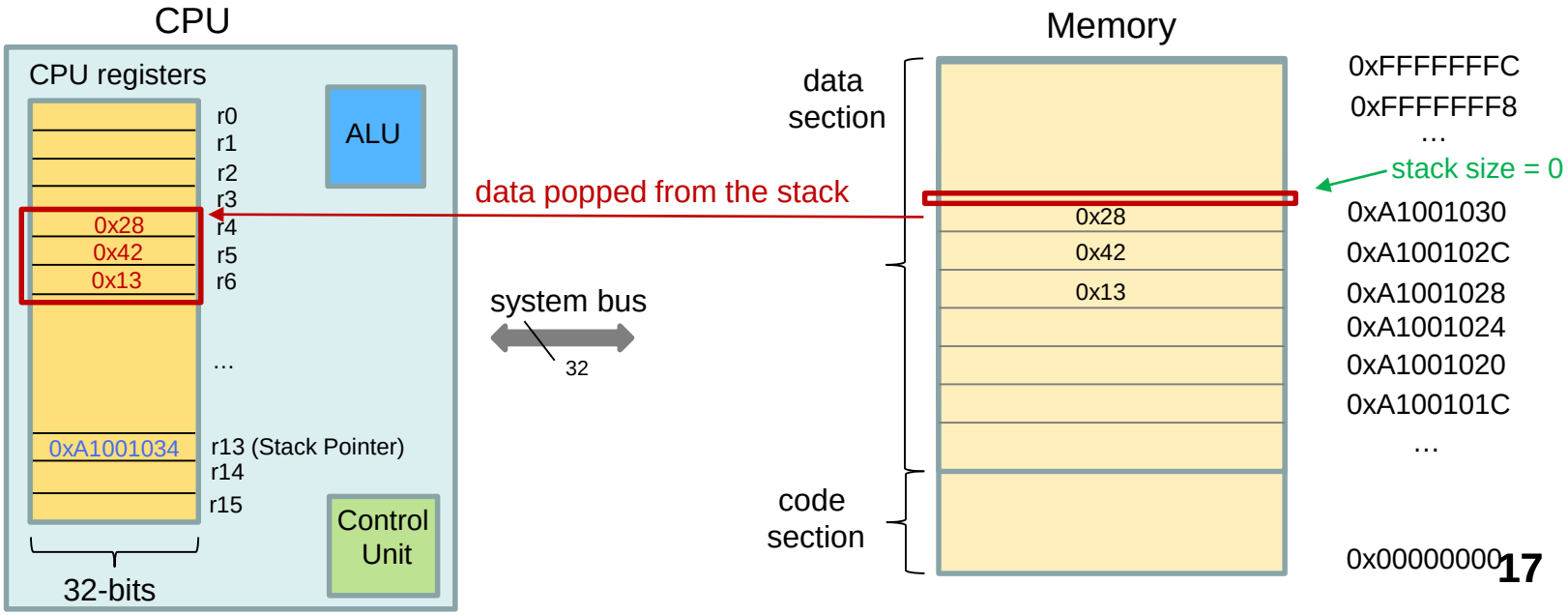
The Stack

- Why is a stack important?
  - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 – r11, however, before the function exits, r4 – r11 needs to be restored to their initial values at the start of the function. The following process is used to achieve this:
    - At the start of the function, the contents of r4 – r11 are saved on the stack. This is referred to as pushing data onto the stack. Thereafter a function modifies r4 – r11.



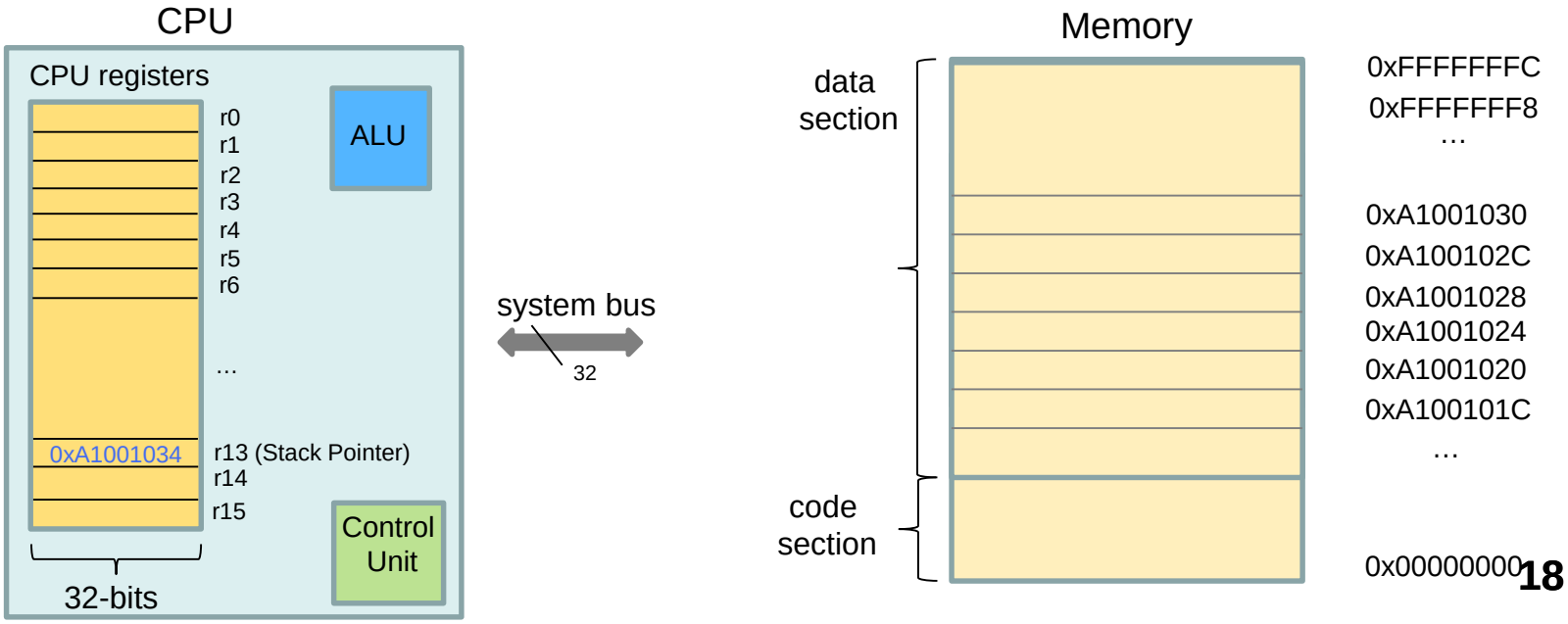
Raspberry Pi Assembler
The Stack

- Why is a stack important?
  - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 – r11, however, before the function exits, r4 – r11 needs to be restored to their initial values at the start of the function. The following process is used to achieve this:
    - At the start of the function, the contents of r4 – r11 are saved on the stack. This is referred to as pushing data onto the stack. Thereafter a function modifies r4 – r11.
    - Before the function exits, the values from the stack are used to restore the CPU registers back to their initial values. This is referred to as popping data from the stack.



Raspberry Pi Assembler
The Stack

- Why is a stack important?
  - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 – r11, however, before the function exits, r4 – r11 needs to be restored to their initial values at the start of the function.
  - Used in recursive functions to save the Link Register (r14) and other CPU registers for each dynamic activation of the recursive function. This is discussed in great depth in this chapter with an example.

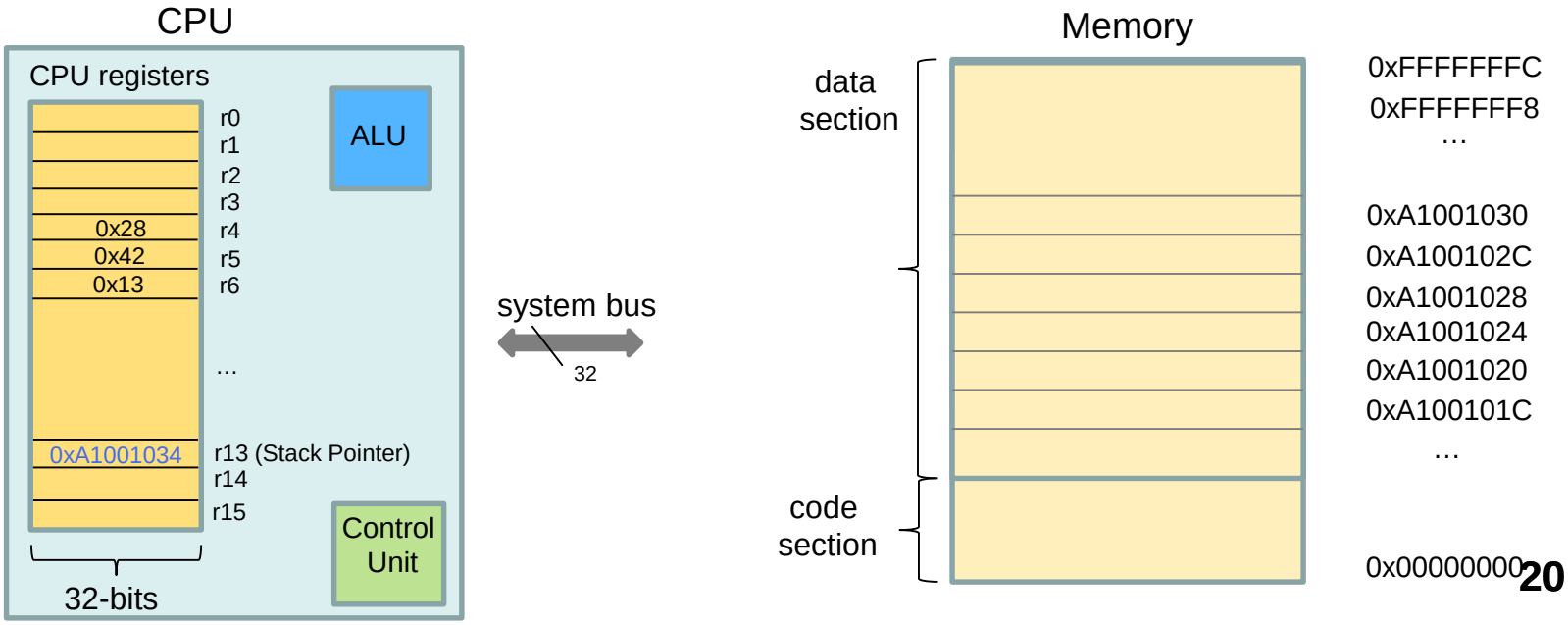


Stack operations

Raspberry Pi Assembler

Stack Operations

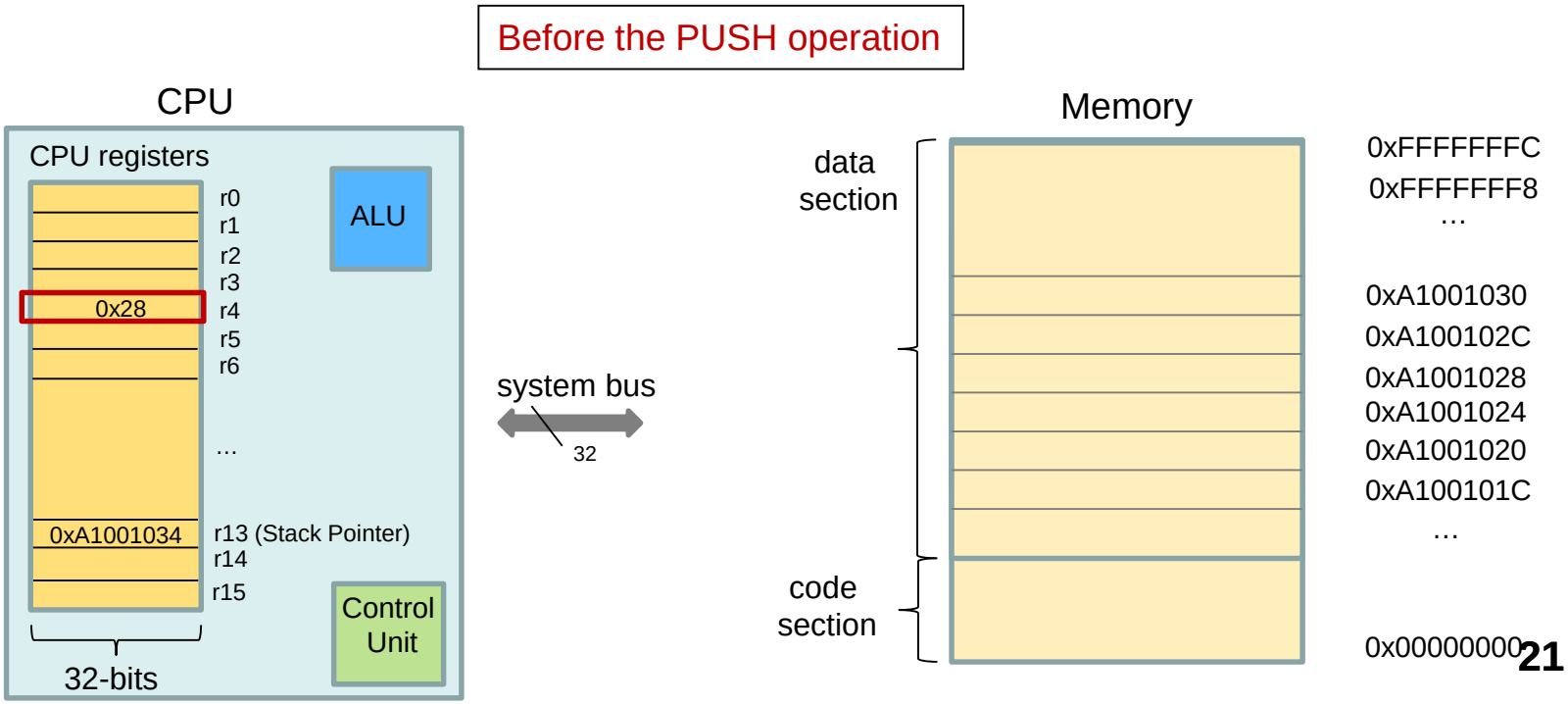
- We have identified two stack operations



Raspberry Pi Assembler

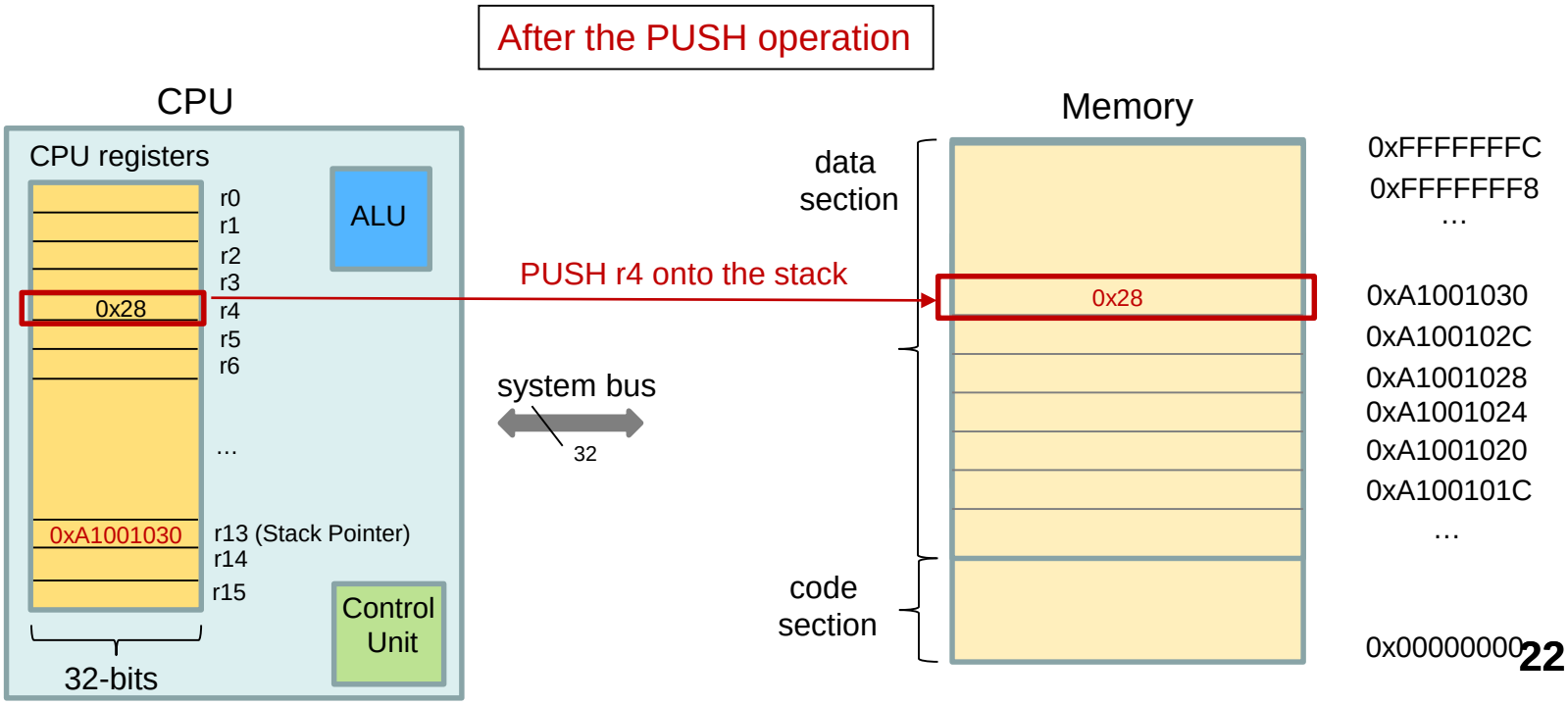
Stack Operations

- We have identified two stack operations
  - Adding data to the stack: also referred to as pushing data onto the stack



Raspberry Pi Assembler
Stack Operations

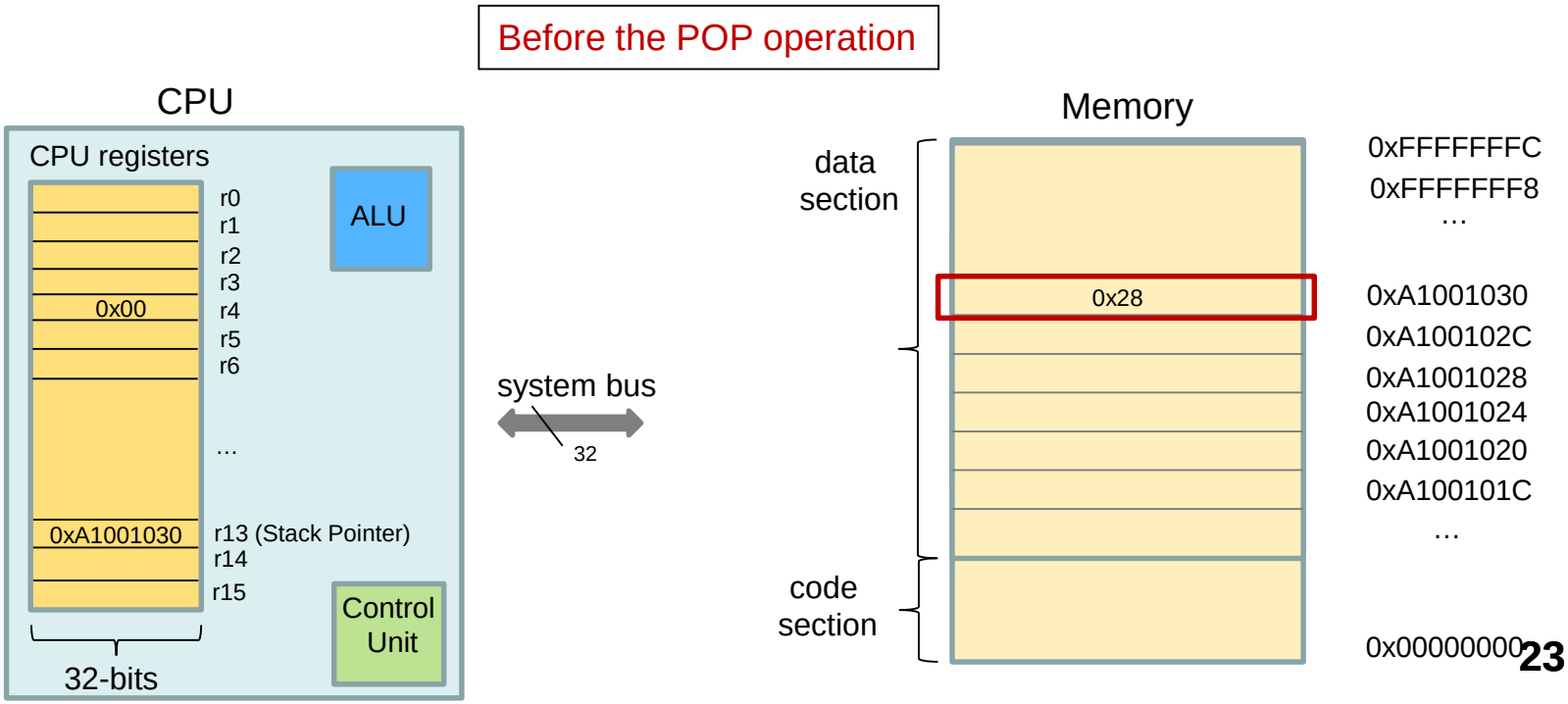
- We have identified two stack operations
  - Adding data to the stack: also referred to as pushing data onto the stack
    - Step 1: Decrement the Stack Pointer (SP) by 4
    - Step 2: Store the CPU register value to the memory address given by the SP



Raspberry Pi Assembler

Stack Operations

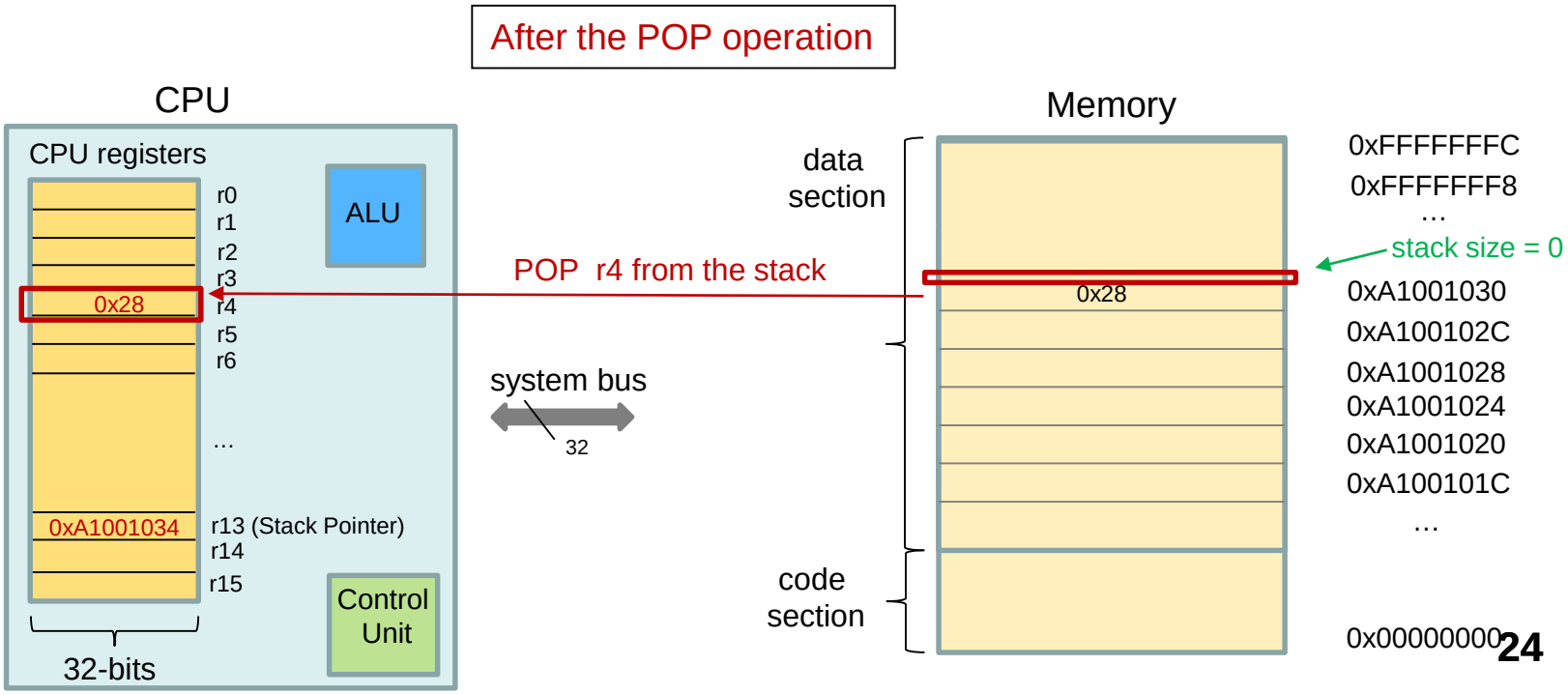
- We have identified two stack operations
  - Adding data to the stack: also referred to as pushing data onto the stack
    - Step 1: Decrement the Stack Pointer (SP) by 4
    - Step 2: Store the CPU register value to the memory address given by the SP
  - Reading data from the stack: also referred to as popping data from the stack



Raspberry Pi Assembler

Stack Operations

- We have identified two stack operations
  - Adding data to the stack: also referred to as pushing data onto the stack
    - Step 1: Decrement the Stack Pointer (SP) by 4
    - Step 2: Store the CPU register value to the memory address given by the SP
  - Reading data from the stack: also referred to as popping data from the stack
    - Step 1: Load the contents at the memory address given by the SP to the CPU register
    - Step 2: Increment the Stack Pointer (SP) by 4



Raspberry Pi Assembler

Stack Operations: Assembly instructions

- Adding data to the stack: pushing data onto the stack
  - Step 1: Decrement the Stack Pointer (SP) by 4
  - Step 2: Store the CPU register value to the memory address given by the SP

Assembly instruction for pushing r4 to the top of the stack

```
str  r4 , [sp,  #-4]!          /*  pre-indexing:   *(sp - 4)  =  r4   */
```

```
push   r4   /*  GNU macro for the push operation */
```

equivalent instructions

Raspberry Pi Assembler

Stack Operations: Assembly instructions

- Adding data to the stack: pushing data onto the stack
  - Step 1: Decrement the Stack Pointer (SP) by 4
  - Step 2: Store the CPU register value to the memory address given by the SP

Assembly instruction for pushing r4 to the top of the stack

```
str  r4 , [sp,  #-4]!          /*  pre-indexing:   *(sp - 4)  =  r4   */
```

```
push   r4   /*  GNU macro for the push operation */
```

equivalent instructions

- Reading data from the stack: popping data from the stack
  - Step 1: Load the contents at the memory address given by the SP to the CPU register
  - Step 2: Increment the Stack Pointer (SP) by 4

Assembly instruction for popping the value from the top of the stack to r4

```
ldr  r4 , [sp],  #4        /*  post-indexing:   r4 = *(sp) ,  sp = sp + 4)    */
```

```
pop   r4        /*  GNU macro for the pop operation */
```

equivalent instructions

Raspberry Pi Assembler

Stack Operations: Assembly instructions

- We can also add and read multiple data items from the stack

Assembly instruction for pushing the contents of r4 and Link Register (LR) to the stack

```
push  { r4,  lr}        /*  saves the higher number CPU registers
                            first and then the lower ones   */
```

```
push   lr       /*  push the contents of r14 (LR) to the stack */
push   r4       /* push the contens of r4 to the stack */
```

equivalent instructions

Assembly instruction for popping the values at the top of the stack to r4 and the LR

```
pop  { r4,  lr}         /*  loads the lower number CPU register first
                            and then the higher ones   */
```

```
pop   r4        /*  pop the top of the stack to register r4  */
pop   lr        /*  pop the top of the stack to register r4 */
```

equivalent instructions

Application of the stack: recursive functions

Raspberry Pi Assembler

Application of the stack: recursive functions

- Calculate the factorial of a number

```
0! = 1
n! = n x (n-1) x (n-2) … x 1
   = n x (n-1)!
```

C-code for a recursive function

```
int  factorial (int n)
  {
       if (n == 0)
          return 1
       else
          return n * factorial(n-1)
       end
   }
```

Raspberry Pi Assembler

Application of the stack: recursive functions

- Calculate the factorial of a number

```
0! = 1
n! = n x (n-1) x (n-2) … x 1
   = n x (n-1)!
```

C-code for a recursive function

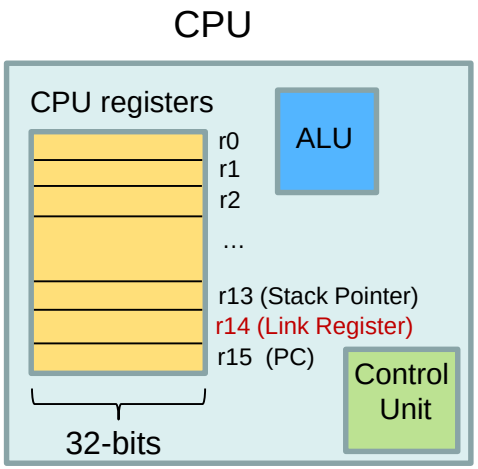
```
int factorial (int n)
  {
      if (n == 0)
         return 1
      else
         return n * factorial(n-1)
      end
   }
```

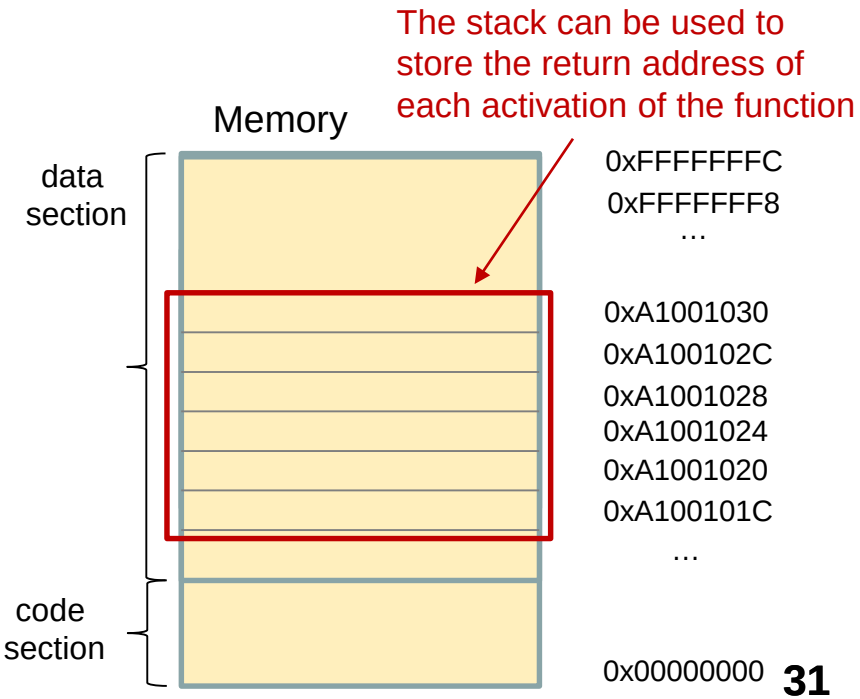
the function factorial calls itself and is an example of a recursive function

Raspberry Pi Assembler

Application of the stack: recursive functions

- Let's look at an assembly program to compute the factorial of a number
- Since the function factorial is called multiple times, a single Link Register (R14) is insufficient to store the return address of each activation of the function.
- The stack offers a solution to save the return address of each activation of the function





Raspberry Pi Assembler
Application of the stack: recursive functions

- The assembly code written by R. Ferrer uses the mul assembly instruction. Let's understand how this instruction works

```
mul  Rdest,  Rsource1,  Rsource2
```

- Performs the following operation

Rdest = Rsource1 x Rsource2

lower 32-bits of the 64-bit result is retained

32-bits

32-bits

- Example:

```
mul  r0,  r1,  r2      /* r0 = r1 x r2 */
```

Assembly programs to compute the factorial of a number

Raspberry Pi Assembler

Application of the stack: factorial assembly program

- Three versions of the code was written. They all perform the same outcome, however, each approach is different.
- Version 1: recursive function implemented. Uses CPU registers r0, r1 as local variables inside the function
- Version 2: recursive function implemented. Uses CPU registers r0, r1, r4 as local variables
- Version 3: non-recursive function or loop used to compute the factorial operation

$$0! = 1$$
$$n! = n \times (n-1) \times (n-2) \ldots \times 1$$
$$= n \times (n-1)!$$

- We can only compute up to 12! Since $13! > (2^{32} - 1)$

Raspberry Pi Assembler

Application of the stack: factorial assembly program

- Three versions of the code was written. They all perform the same outcome, however, each approach is different.
- Version 1: recursive function implemented. Uses CPU registers r0, r1 as local variables inside the function
- Version 2: recursive function implemented. Uses CPU registers r0, r1, r4 as local variables
- Version 3: non-recursive function or loop used to compute the factorial operation

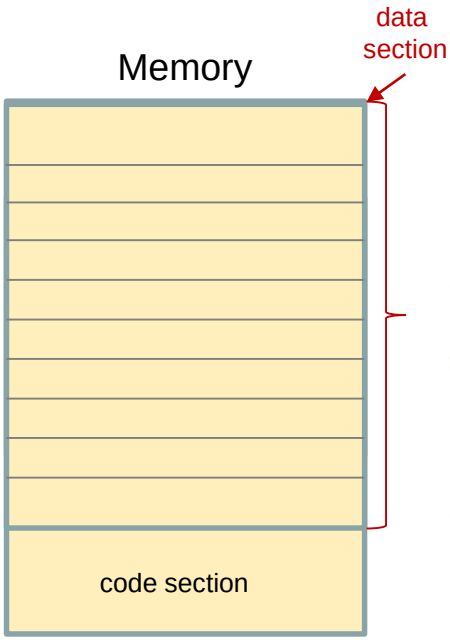
We will only look at version 2

0! = 1
n! = n x (n-1) x (n-2) … x 1
= n x (n-1)!

- We can only compute up to 12! Since 13! > (2^{32} -1)

Version 2: factorial assembly program

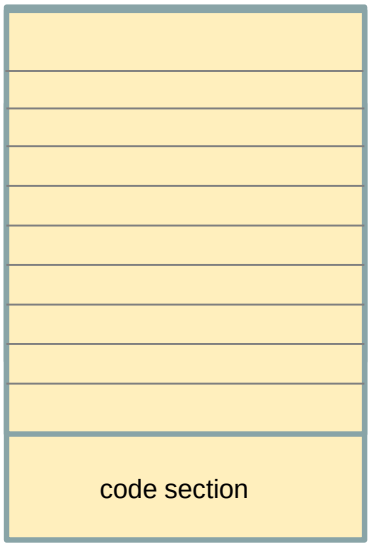
```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
    str lr, [sp,#-4]!   @ Push lr onto the top of the stack

    ldr r0, =message1   @ Set &message1 as the first parameter of printf
    bl  printf          @ Call printf

    ldr r0, =format     @ Set &format as the first parameter of scanf
    sub sp, sp, #4      @ Make room for one 4 byte integer on the stack
                        @ We will keep the number entered by the user there
    mov r1, sp          @ Set the top of the stack as the second parameter
                        @           of scanf
    bl  scanf           @ Call scanf

    ldr r0, [sp]        @ Load the integer read by scanf into r0
                        @ So we set it as the first parameter of factorial
    add sp, sp, #+4     @ Discard the integer read by scanf
    bl  factorial       @ Call factorial

    mov r2, r0          @ Get the result of factorial and move it to r2
                        @ So we set it as the third parameter of printf
    ldr r1, [sp]        @ Load the integer read by scanf into r1
                        @ So we set it as the second parameter of printf
    ldr r0, =message2   @ Set &message2 as the first parameter of printf
    bl  printf          @ Call printf

    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave main
```



Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     str lr, [sp,#-4]!   @ Push lr onto the top of the stack

     ldr r0, =message1   @ Set &message1 as the first parameter of printf
     bl  printf          @ Call printf

     ldr r0, =format     @ Set &format as the first parameter of scanf
     sub sp, sp, #4      @ Make room for one 4 byte integer on the stack
                         @ We will keep the number entered by the user there
     mov r1, sp          @ Set the top of the stack as the second parameter
                         @           of scanf
     bl  scanf           @ Call scanf

     ldr r0, [sp]        @ Load the integer read by scanf into r0
                         @ So we set it as the first parameter of factorial
     add sp, sp, #+4     @ Discard the integer read by scanf
     bl  factorial       @ Call factorial

     mov r2, r0          @ Get the result of factorial and move it to r2
                         @ So we set it as the third parameter of printf
     ldr r1, [sp]        @ Load the integer read by scanf into r1
                         @ So we set it as the second parameter of printf
     ldr r0, =message2   @ Set &message2 as the first parameter of printf
     bl  printf          @ Call printf

     ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
     bx  lr              @ Leave main
```

declare strings to be used later in the program



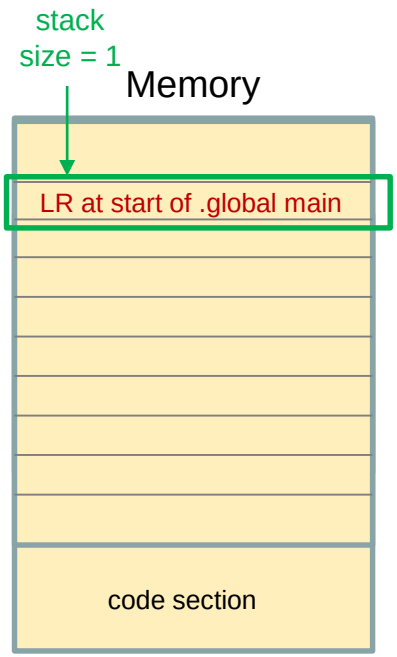
Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
    str lr, [sp,#-4]!   @ Push lr onto the top of the stack

    ldr r0, =message1   @ Set &message1 as the first parameter of printf
    bl  printf          @ Call printf

    ldr r0, =format     @ Set &format as the first parameter of scanf
    sub sp, sp, #4      @ Make room for one 4 byte integer on the stack
                        @ We will keep the number entered by the user there
    mov r1, sp          @ Set the top of the stack as the second parameter
                        @           of scanf
    bl  scanf           @ Call scanf

    ldr r0, [sp]        @ Load the integer read by scanf into r0
                        @ So we set it as the first parameter of factorial
    add sp, sp, #+4     @ Discard the integer read by scanf
    bl  factorial       @ Call factorial

    mov r2, r0          @ Get the result of factorial and move it to r2
                        @ So we set it as the third parameter of printf
    ldr r1, [sp]        @ Load the integer read by scanf into r1
                        @ So we set it as the second parameter of printf
    ldr r0, =message2   @ Set &message2 as the first parameter of printf
    bl  printf          @ Call printf

    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave main
```

At the start of the global main function, save the contents of the Link Register onto the stack

Can be replaced with one instruction

- push lr



Version 2: factorial assembly program

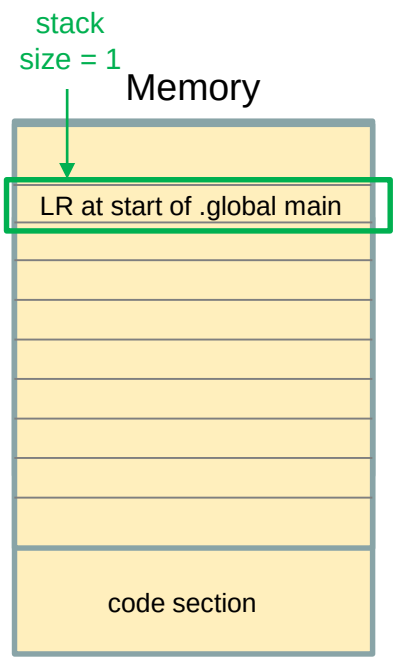
```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     push lr                @ Push lr onto the top of the stack

     ldr r0, =message1      @ Set &message1 as the first parameter of printf
     bl  printf             @ Call printf

     ldr r0, =format        @ Set &format as the first parameter of scanf
     sub sp, sp, #4         @ Make room for one 4 byte integer on the stack
                            @ We will keep the number entered by the user there
     mov r1, sp             @ Set the top of the stack as the second parameter
                            @           of scanf
     bl  scanf              @ Call scanf

     ldr r0, [sp]           @ Load the integer read by scanf into r0
                            @ So we set it as the first parameter of factorial
     add sp, sp, #+4        @ Discard the integer read by scanf
     bl  factorial          @ Call factorial

     mov r2, r0             @ Get the result of factorial and move it to r2
                            @ So we set it as the third parameter of printf
     ldr r1, [sp]           @ Load the integer read by scanf into r1
                            @ So we set it as the second parameter of printf
     ldr r0, =message2      @ Set &message2 as the first parameter of printf
     bl  printf             @ Call printf

     ldr lr, [sp], #+4      @ Pop the top of the stack and put it in lr
     bx  lr                 @ Leave main
```

prepare the input parameter(s) for the printf instruction by loading the address of the string ***message*** to register r0

After instruction has executed:

Type a number:

is displayed to the terminal

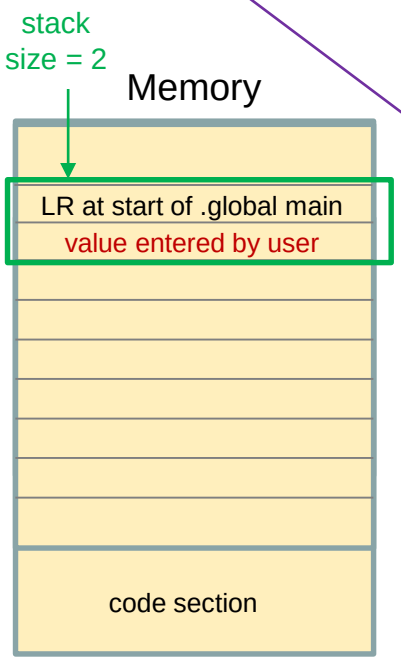


Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     push lr              @ Push lr onto the top of the stack

     ldr r0, =message1    @ Set &message1 as the first parameter of printf
     bl  printf           @ Call printf

     ldr r0, =format      @ Set &format as the first parameter of scanf
     sub sp, sp, #4       @ Make room for one 4 byte integer on the stack
                          @ We will keep the number entered by the user there
     mov r1, sp           @ Set the top of the stack as the second parameter
                          @           of scanf
     bl  scanf            @ Call scanf

     ldr r0, [sp]         @ Load the integer read by scanf into r0
                          @ So we set it as the first parameter of factorial
     add sp, sp, #+4      @ Discard the integer read by scanf
     bl  factorial        @ Call factorial

     mov r2, r0           @ Get the result of factorial and move it to r2
                          @ So we set it as the third parameter of printf
     ldr r1, [sp]         @ Load the integer read by scanf into r1
                          @ So we set it as the second parameter of printf
     ldr r0, =message2    @ Set &message2 as the first parameter of printf
     bl  printf           @ Call printf

     ldr lr, [sp], #+4    @ Pop the top of the stack and put it in lr
     bx  lr               @ Leave main
```

After instruction has executed

the data entered by the user is save to the top of the stack



prepare the input parameter(s) for the scanf instruction

- r0: format of the input data
- r1: memory address to save user entered data

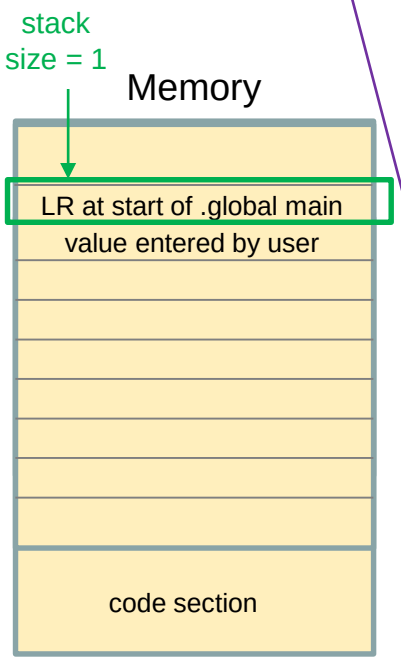
In this case, we want to save the user entered data to the top of the stack

Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     push lr              @ Push lr onto the top of the stack

     ldr r0, =message1    @ Set &message1 as the first parameter of printf
     bl  printf           @ Call printf

     ldr r0, =format      @ Set &format as the first parameter of scanf
     sub sp, sp, #4       @ Make room for one 4 byte integer on the stack
                          @ We will keep the number entered by the user there
     mov r1, sp           @ Set the top of the stack as the second parameter
                          @           of scanf
     bl  scanf            @ Call scanf

     ldr r0, [sp]         @ Load the integer read by scanf into r0
                          @ So we set it as the first parameter of factorial
     add sp, sp, #+4      @ Discard the integer read by scanf
     bl  factorial        @ Call factorial

     mov r2, r0           @ Get the result of factorial and move it to r2
                          @ So we set it as the third parameter of printf
     ldr r1, [sp]         @ Load the integer read by scanf into r1
                          @ So we set it as the second parameter of printf
     ldr r0, =message2    @ Set &message2 as the first parameter of printf
     bl  printf           @ Call printf

     ldr lr, [sp], #+4    @ Pop the top of the stack and put it in lr
     bx  lr               @ Leave main
```

After instruction has executed

r0 will contain the factorial value of the user entered data



POP the data item at the top of the stack into register r0

prepare the input parameter for the factorial function

- r0: data entered by user

Can be replaced with one instruction

- pop r0

Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     push lr              @ Push lr onto the top of the stack

     ldr r0, =message1    @ Set &message1 as the first parameter of printf
     bl  printf           @ Call printf

     ldr r0, =format      @ Set &format as the first parameter of scanf
     sub sp, sp, #4       @ Make room for one 4 byte integer on the stack
                          @ We will keep the number entered by the user there
     mov r1, sp           @ Set the top of the stack as the second parameter
                          @           of scanf
     bl  scanf            @ Call scanf

                          @ Load the integer read by scanf into r0
     pop r0               @ So we set it as the first parameter of factorial
                          @ Discard the integer read by scanf

     bl  factorial        @ Call factorial

     mov r2, r0           @ Get the result of factorial and move it to r2
                          @ So we set it as the third parameter of printf
     ldr r1, [sp]         @ Load the integer read by scanf into r1
                          @ So we set it as the second parameter of printf
     ldr r0, =message2    @ Set &message2 as the first parameter of printf
     bl  printf           @ Call printf

     ldr lr, [sp], #+4    @ Pop the top of the stack and put it in lr
     bx  lr               @ Leave main
```

prepare the input parameter(s) for the printf instruction

- r0 – address of the string message2
- r1 – data to be included in the string
- r2 – data to be included in the string

In this case r1 is the data entered by the user and r2 is the computed factorial value

After instruction has executed

The factorial of 3 is 6

is displayed to the terminal

Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     push lr                @ Push lr onto the top of the stack

     ldr r0, =message1      @ Set &message1 as the first parameter of printf
     bl  printf             @ Call printf

     ldr r0, =format        @ Set &format as the first parameter of scanf
     sub sp, sp, #4         @ Make room for one 4 byte integer on the stack
                            @ We will keep the number entered by the user there
     mov r1, sp             @ Set the top of the stack as the second parameter
                            @           of scanf
     bl  scanf              @ Call scanf

                            @ Load the integer read by scanf into r0
     pop r0                 @ So we set it as the first parameter of factorial
                            @ Discard the integer read by scanf

     bl  factorial          @ Call factorial

     mov r2, r0             @ Get the result of factorial and move it to r2
                            @ So we set it as the third parameter of printf
     ldr r1, [sp]           @ Load the integer read by scanf into r1
                            @ So we set it as the second parameter of printf
     ldr r0, =message2      @ Set &message2 as the first parameter of printf
     bl  printf             @ Call printf

     ldr lr, [sp], #+4      @ Pop the top of the stack and put it in lr
     bx  lr                 @ Leave main
```

restore the Link Register to the value it was at the start of the function

Can be replaced with one instruction

- pop lr

Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     push lr              @ Push lr onto the top of the stack

     ldr r0, =message1    @ Set &message1 as the first parameter of printf
     bl  printf           @ Call printf

     ldr r0, =format      @ Set &format as the first parameter of scanf
     sub sp, sp, #4       @ Make room for one 4 byte integer on the stack
                          @ We will keep the number entered by the user there
     mov r1, sp           @ Set the top of the stack as the second parameter
                          @           of scanf
     bl  scanf            @ Call scanf

                          @ Load the integer read by scanf into r0
                          @ So we set it as the first parameter of factorial
     pop r0
                          @ Discard the integer read by scanf

     bl  factorial        @ Call factorial

     mov r2, r0           @ Get the result of factorial and move it to r2
                          @ So we set it as the third parameter of printf
     ldr r1, [sp]         @ Load the integer read by scanf into r1
                          @ So we set it as the second parameter of printf
     ldr r0, =message2    @ Set &message2 as the first parameter of printf
     bl  printf           @ Call printf

     pop lr               @ Pop the top of the stack and put it in lr
     bx  lr               @ Leave main
```

After the instruction has executed

The main function terminates

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

```
factorial:
    str lr, [sp,#-4]!  @ Push lr onto the top of the stack
    str r4, [sp,#-4]!  @ Push r4 onto the top of the stack

    mov r4, r0         @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0         @ compare r0 and 0
    bne is_nonzero     @ if r0 != 0 then branch
    mov r0, #1         @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:            @ Prepare the call to factorial(n-1)
    sub r0, r0, #1     @ r0 <- r0 - 1
    bl  factorial
                       @ After the call r0 contains factorial(n-1)
                       @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4         @ r1 <- r4
    mul r0, r0, r1     @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4  @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4  @ Pop the top of the stack and put it in lr
    bx  lr             @ Leave factorial
```

r0	r1	r4
2		x

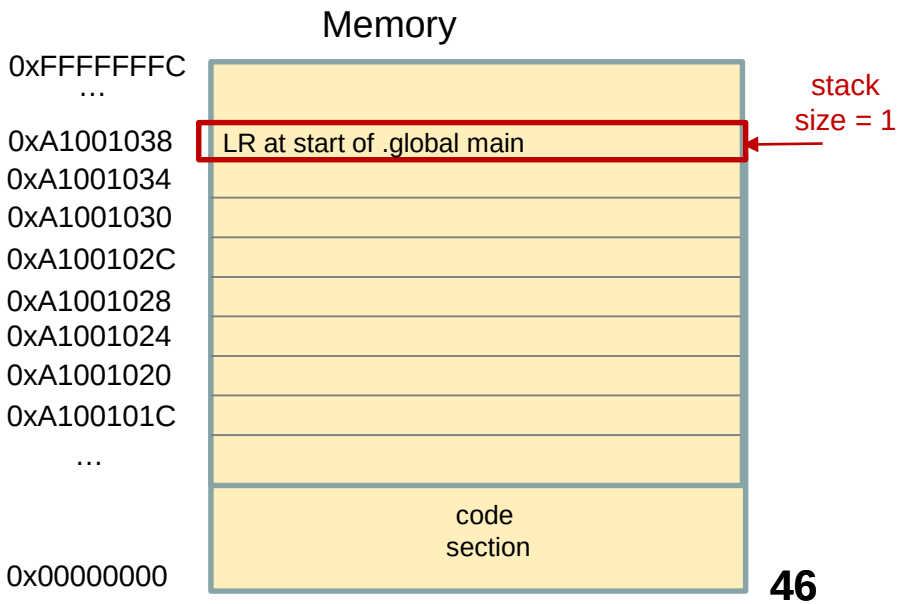
Let contents of r4 before factorial function is called be x

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
2		x

```
factorial:
    str lr, [sp,#-4]!  @ Push lr onto the top of the stack
    str r4, [sp,#-4]!  @ Push r4 onto the top of the stack

    mov r4, r0         @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0         @ compare r0 and 0
    bne is_nonzero     @ if r0 != 0 then branch
    mov r0, #1         @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:            @ Prepare the call to factorial(n-1)
    sub r0, r0, #1     @ r0 <- r0 - 1
    bl  factorial
                       @ After the call r0 contains factorial(n-1)
                       @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4         @ r1 <- r4
    mul r0, r0, r1     @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4  @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4  @ Pop the top of the stack and put it in lr
    bx  lr             @ Leave factorial
```



address of mov r2, r0 from main

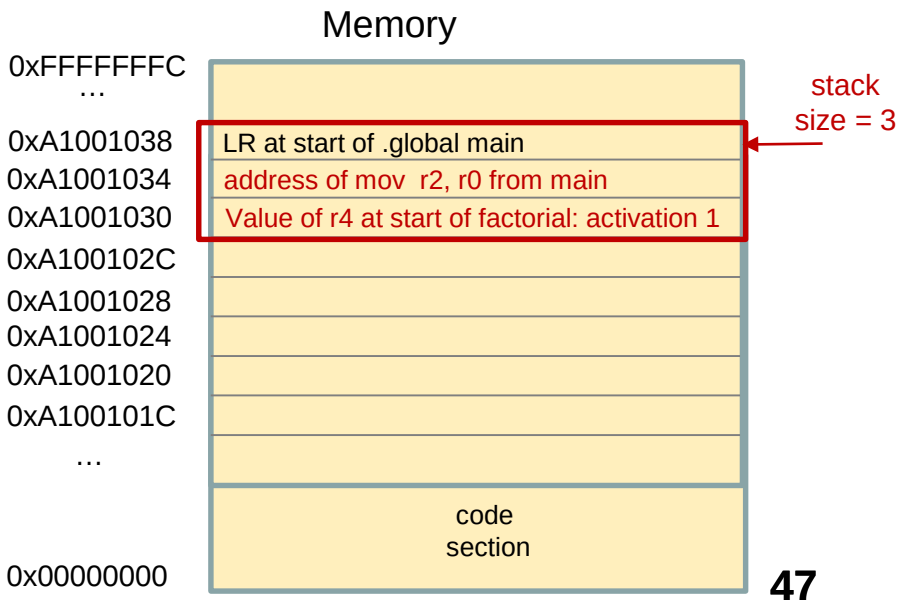
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
2		x

```
factorial:
    str lr, [sp,#-4]!   @ Push lr onto the top of the stack
    str r4, [sp,#-4]!   @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r2, r0 from main

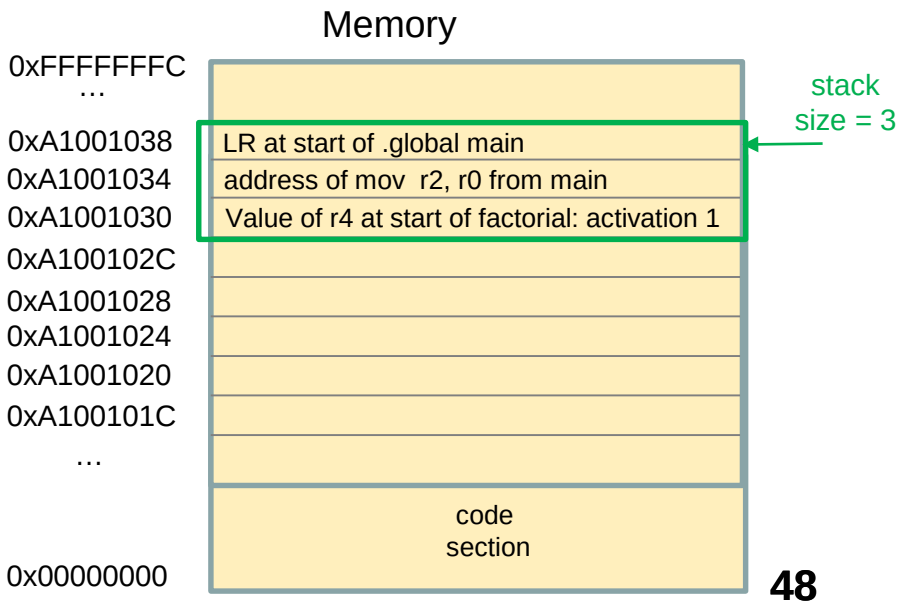
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
2		2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r2, r0 from main

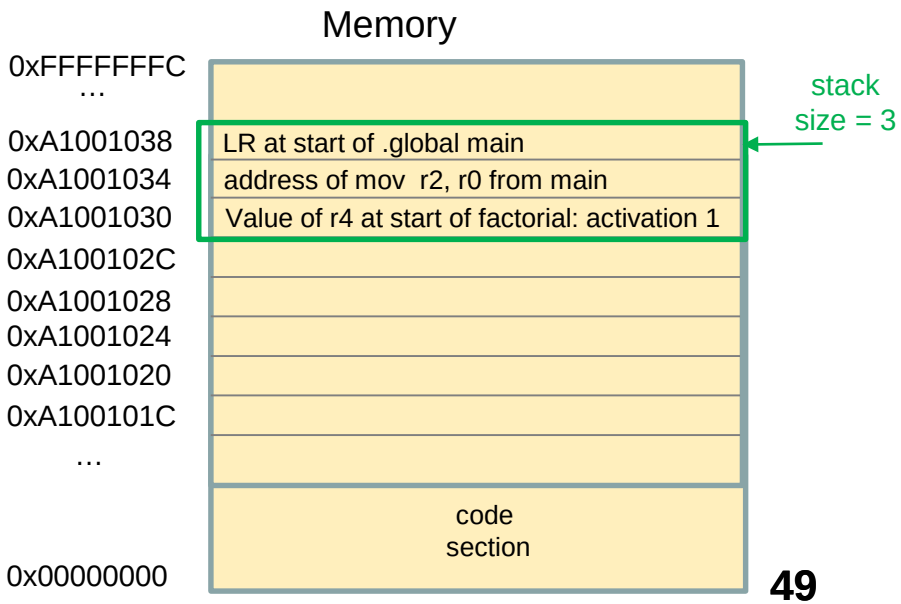
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
2		2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r2, r0 from main

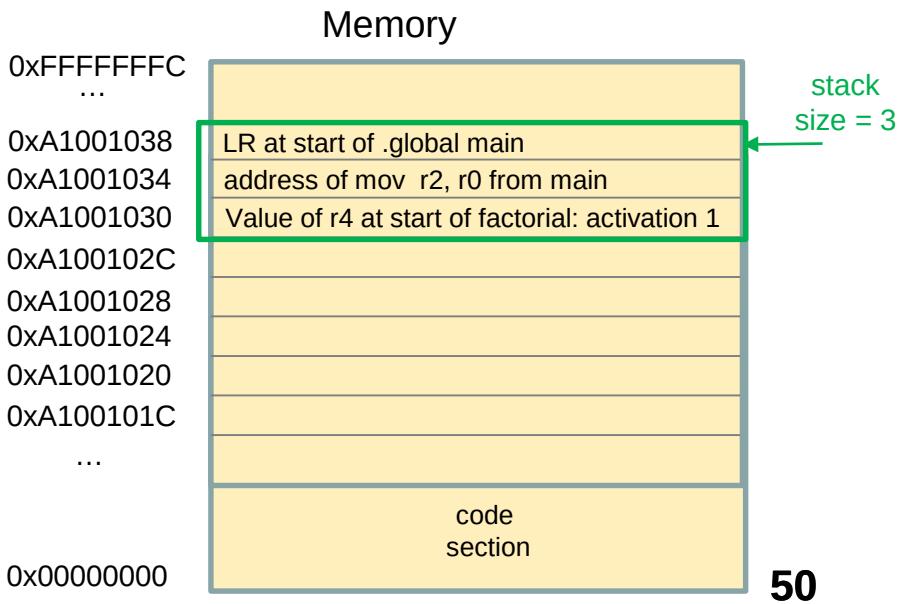
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1		2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r2, r0 from main

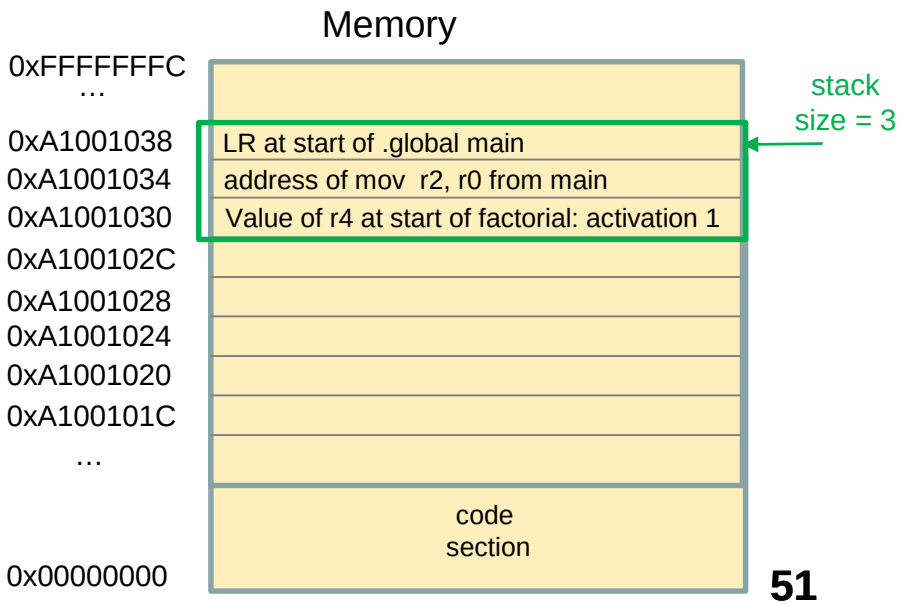
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1		2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 1

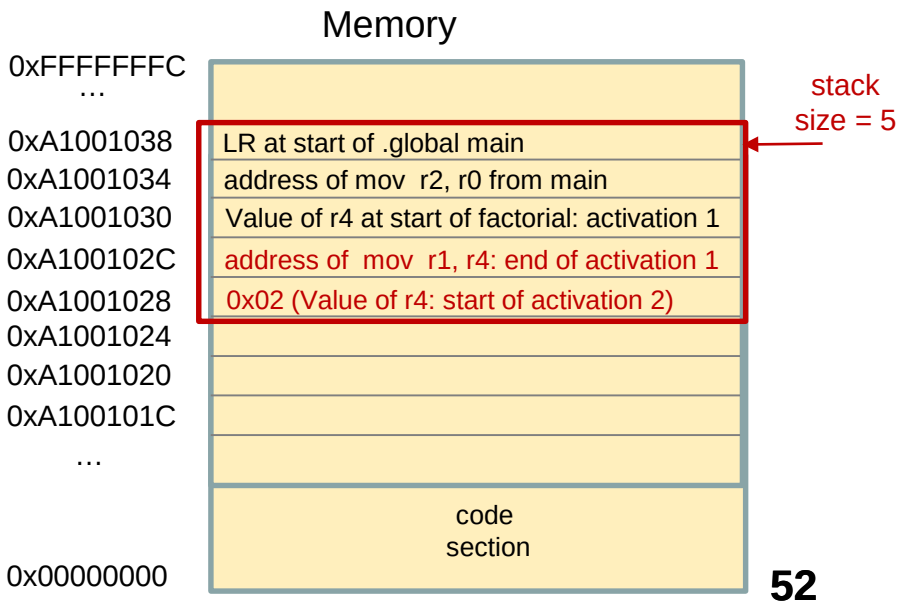
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1		2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 1

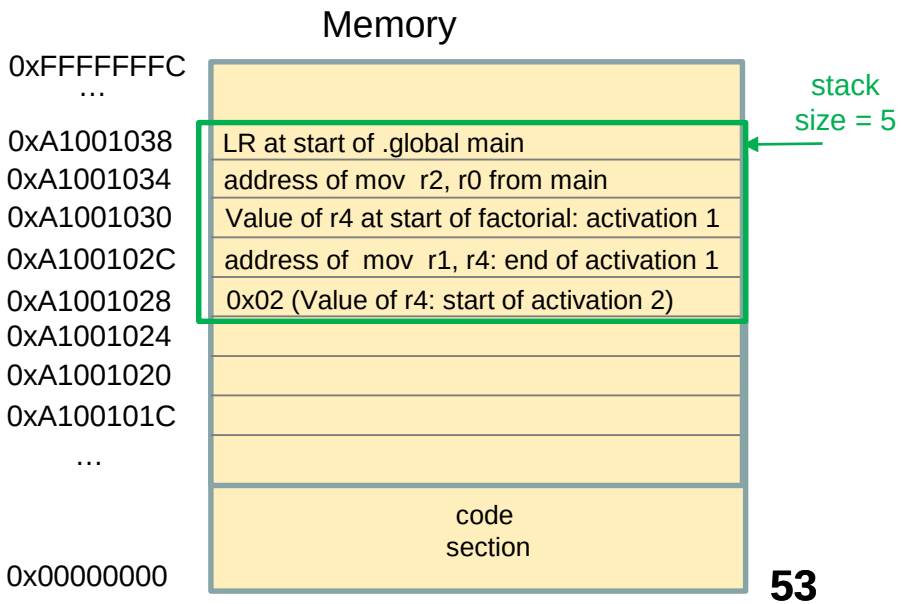
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1		1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

   mov r4, r0           @ Keep a copy of the initial value of r0 in r4

   cmp r0, #0           @ compare r0 and 0
   bne is_nonzero       @ if r0 != 0 then branch
   mov r0, #1           @ r0 <- 1. This is the base case; return
   b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
   sub r0, r0, #1       @ r0 <- r0 - 1
   bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
   mov r1, r4           @ r1 <- r4
   mul r0, r0, r1       @ r0 <- r0 * r1    [See Project]

end:
   ldr r4, [sp], #+4    @ Pop the top of the stack and put it in r4
   ldr lr, [sp], #+4    @ Pop the top of the stack and put it in lr
   bx  lr               @ Leave factorial
```



address of mov r1, r4 : activation 1

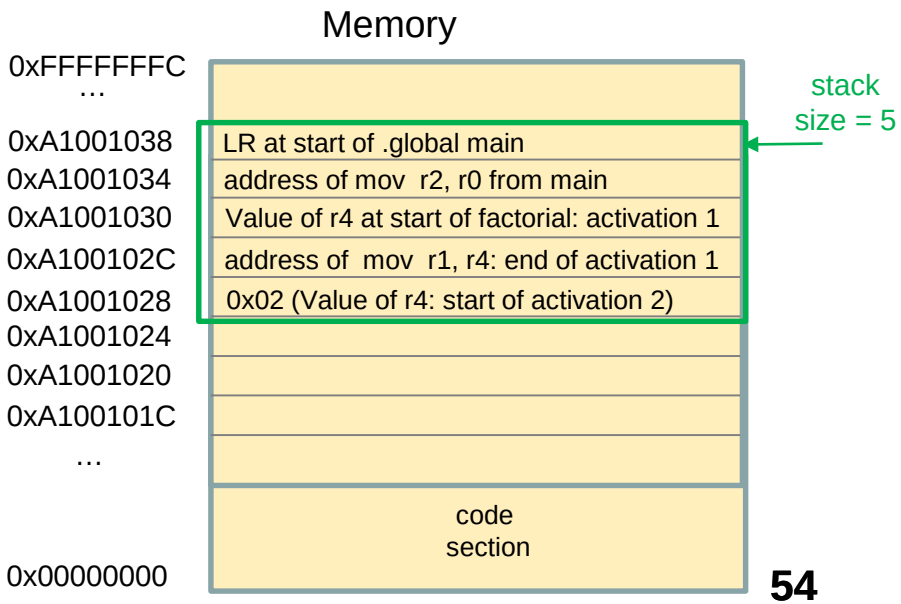
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1		1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 1

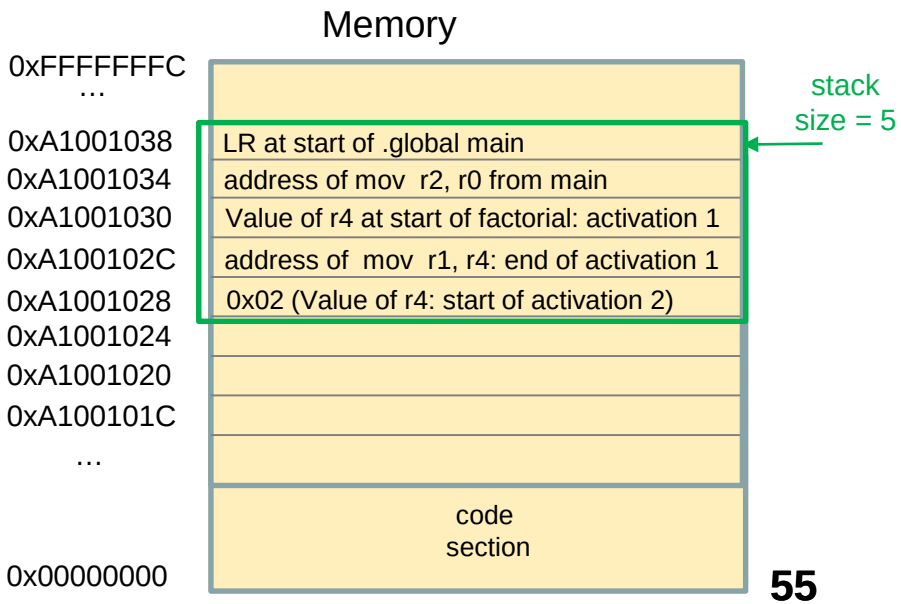
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
0		1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 1

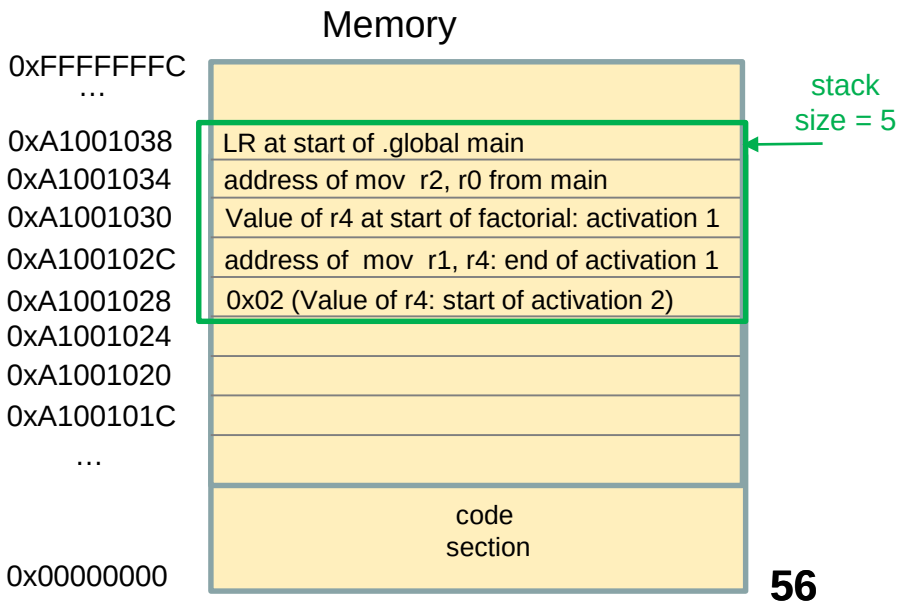
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
0		1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

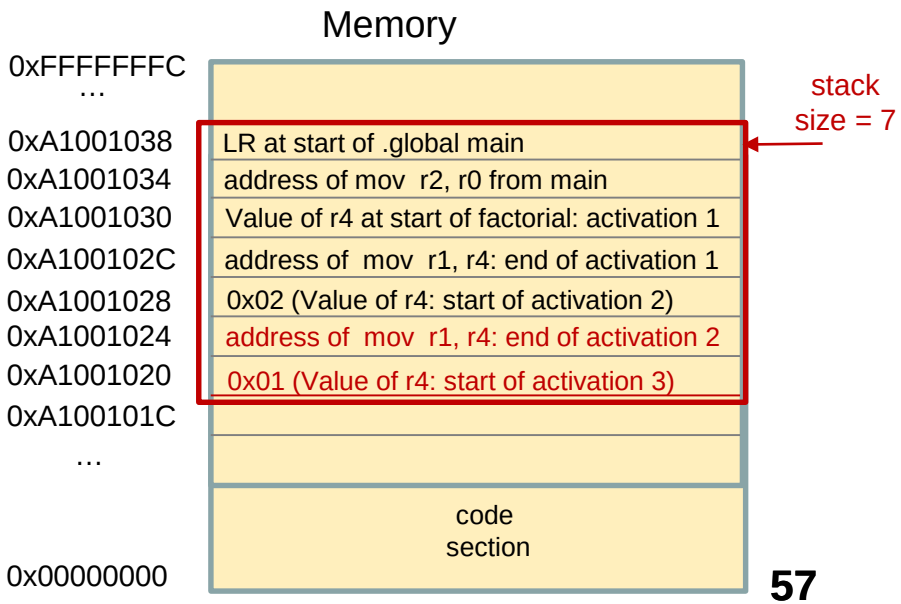
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
0		1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

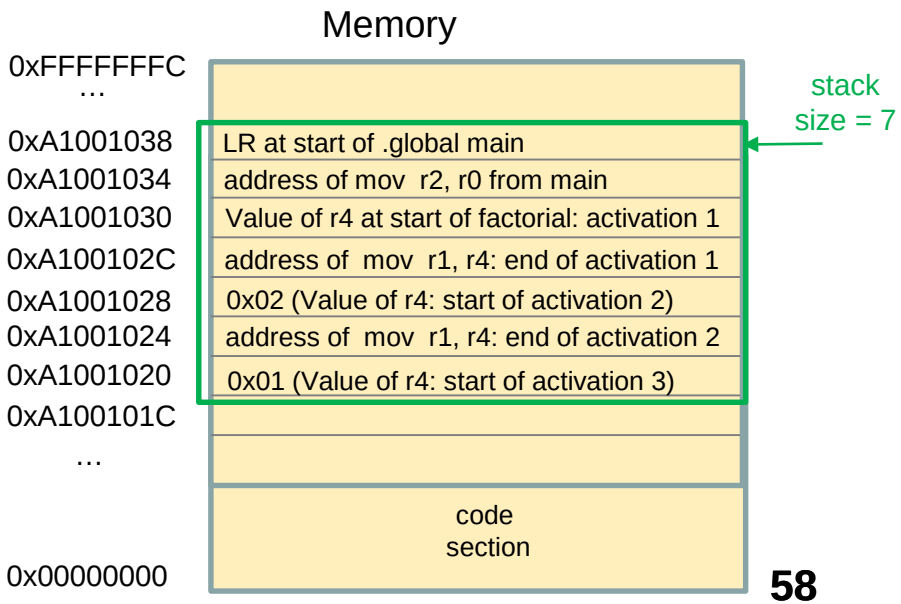
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
0		0

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

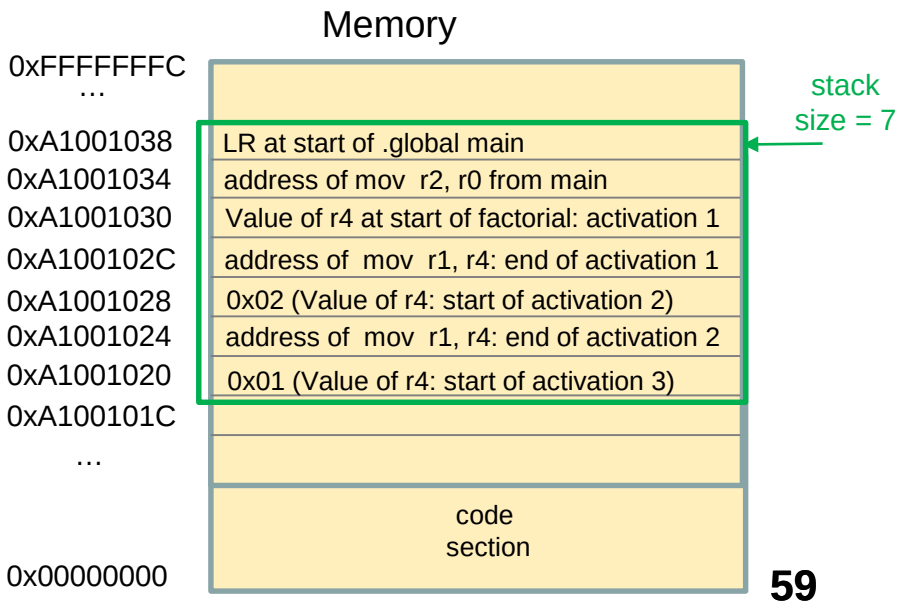
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
0		0

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

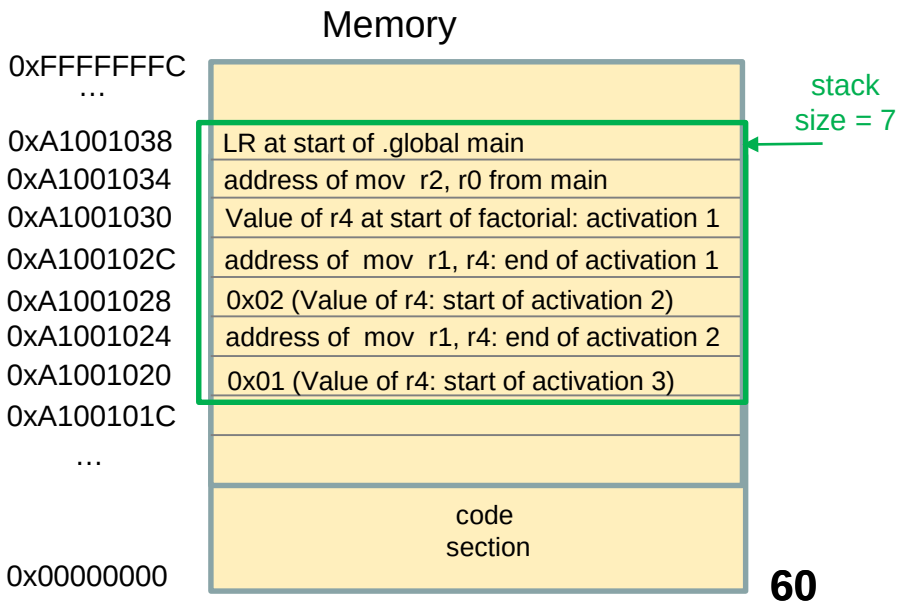
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1		0

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

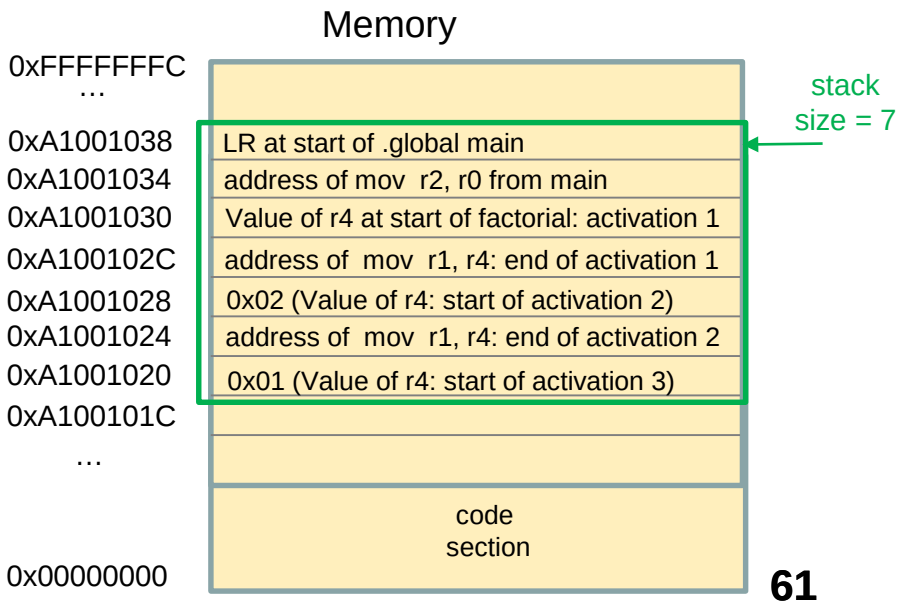
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1		0

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case: return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

r14 (Link Register)

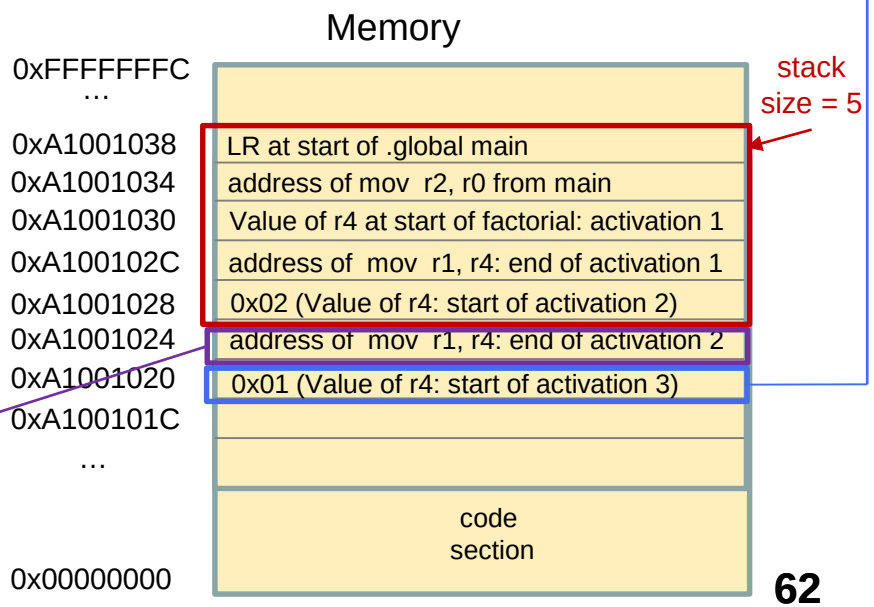
Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
    ldr r4, [sp], #+4   @ Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4   @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```

r0	r1	r4
1		1

0x01 from stack popped into r4



address of mov r1, r4 : activation 2 — r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

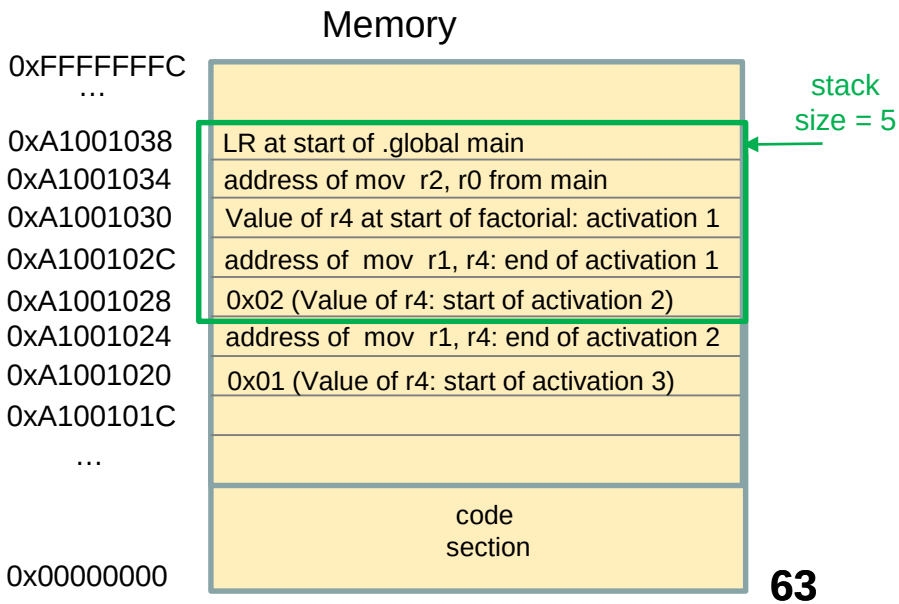
r0	r1	r4
1		1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```

Branch to address specified by LR, which is the address of the instruction mov r1, r4 after activation 2 of the factorial function

address of mov r1, r4 : activation 2

r14 (Link Register)

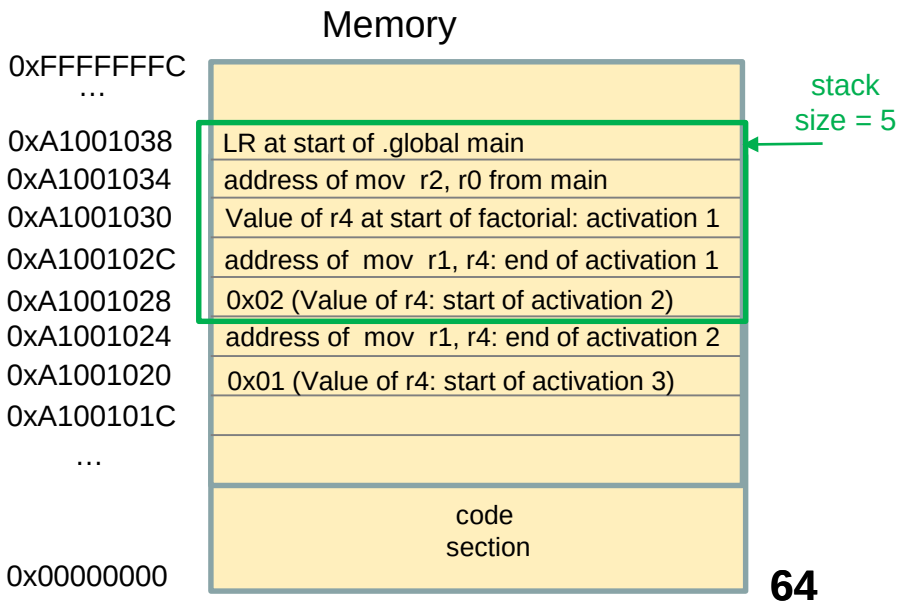


Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1	1	1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1   [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

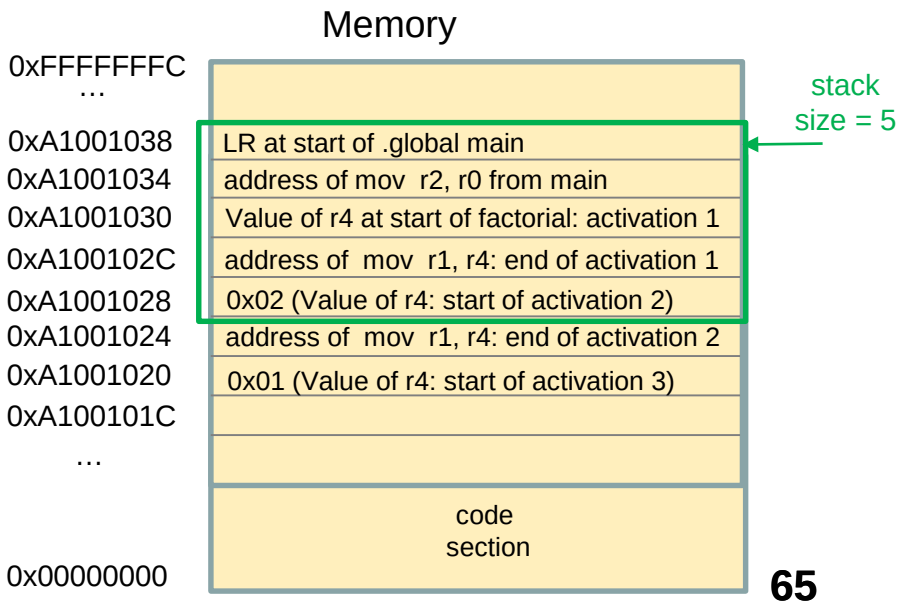
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1	1	1

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 2

r14 (Link Register)

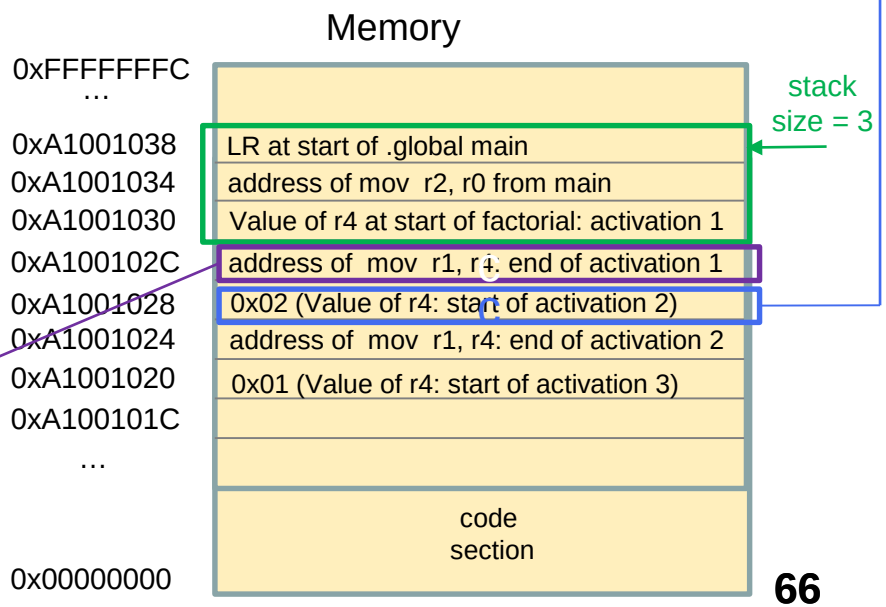
Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```

r0	r1	r4
1	1	2

0x02 from stack popped into r4



address of mov r1, r4 : activation 1

r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

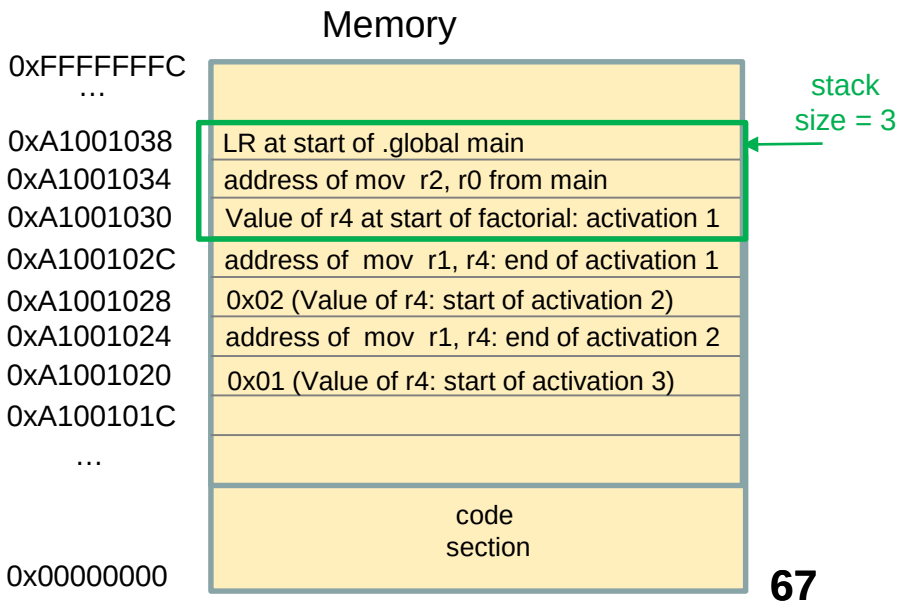
r0	r1	r4
1	1	2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```

Branch to address specified by LR, which is the address of the instruction mov r1, r4 after activation 1 of the factorial function

address of mov r1, r4 : activation 1

r14 (Link Register)

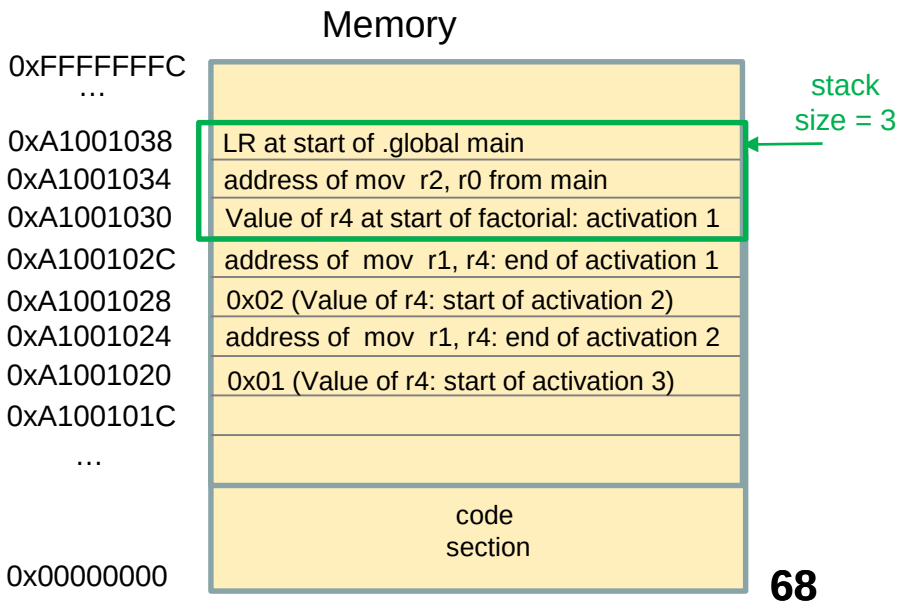


Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
1	2	2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1   [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 1

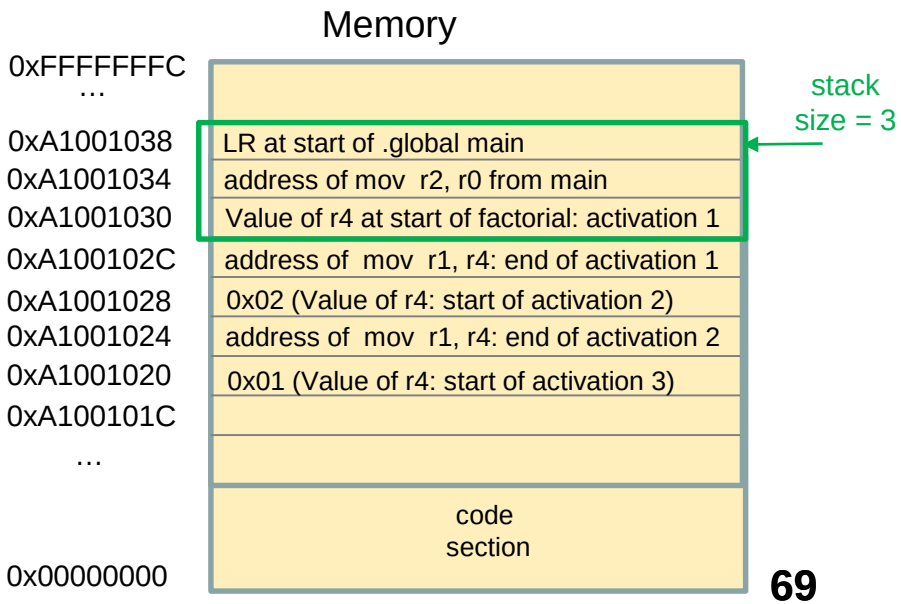
r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

r0	r1	r4
2	2	2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```



address of mov r1, r4 : activation 1

r14 (Link Register)

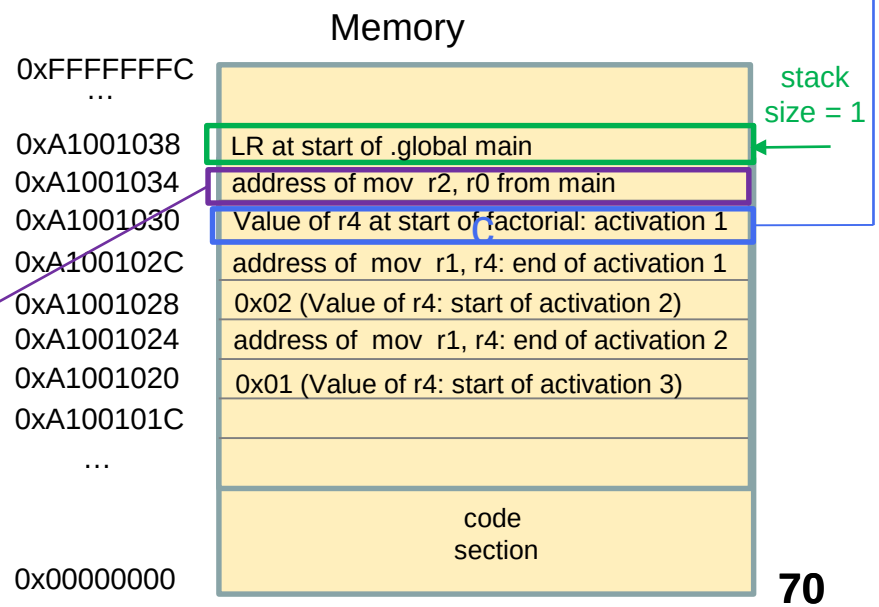
Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```

r0	r1	r4
2	2	x

Value of r4 at the start of the factorial function is stored to value x



address of mov r2, r0 from main

r14 (Link Register)

Version 2: factorial assembly program

- Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the factorial function has completed, we want r0 to have the value 2! or 2

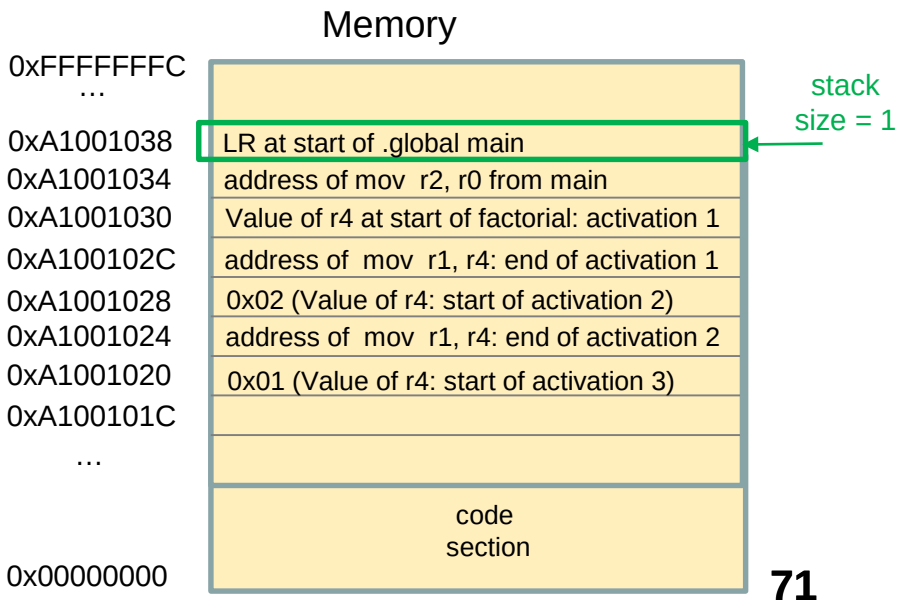
r0	r1	r4
2	2	x

```
factorial:
   push lr              @ Push lr onto the top of the stack
   push r4              @ Push r4 onto the top of the stack

    mov r4, r0          @ Keep a copy of the initial value of r0 in r4

    cmp r0, #0          @ compare r0 and 0
    bne is_nonzero      @ if r0 != 0 then branch
    mov r0, #1          @ r0 <- 1. This is the base case; return
    b   end

is_nonzero:             @ Prepare the call to factorial(n-1)
    sub r0, r0, #1      @ r0 <- r0 - 1
    bl  factorial
                        @ After the call r0 contains factorial(n-1)
                        @ Load initial value of r0 (kept in r4) into r1
    mov r1, r4          @ r1 <- r4
    mul r0, r0, r1      @ r0 <- r0 * r1    [See Project]

end:
     pop r4             @ Pop the top of the stack and put it in r4
     pop lr             @ Pop the top of the stack and put it in lr
    bx  lr              @ Leave factorial
```

Branch to address specified by LR, which is the address of the instruction mov r2, r0 in the global main function

address of mov r2, r0 from main

r14 (Link Register)



Version 2: factorial assembly program

```
/* -- factorial01.s */
.data

message1: .asciz "Type a number: "
format:   .asciz "%d"
message2: .asciz "The factorial of %d is %d\n"

.text
 .globl main
 main:
     push lr              @ Push lr onto the top of the stack

     ldr r0, =message1    @ Set &message1 as the first parameter of printf
     bl  printf           @ Call printf

     ldr r0, =format      @ Set &format as the first parameter of scanf
     sub sp, sp, #4       @ Make room for one 4 byte integer on the stack
                          @ We will keep the number entered by the user there
     mov r1, sp           @ Set the top of the stack as the second parameter
                          @           of scanf
     bl  scanf            @ Call scanf

                          @ Load the integer read by scanf into r0
     pop r0               @ So we set it as the first parameter of factorial
                          @ Discard the integer read by scanf

     bl  factorial        @ Call factorial

     mov r2, r0           @ Get the result of factorial and move it to r2
                          @ So we set it as the third parameter of printf
     ldr r1, [sp]         @ Load the integer read by scanf into r1
                          @ So we set it as the second parameter of printf
     ldr r0, =message2    @ Set &message2 as the first parameter of printf
     bl  printf           @ Call printf

     ldr lr, [sp], #+4    @ Pop the top of the stack and put it in lr
     bx  lr               @ Leave main
```

Branches back to the main function with r0 equal to the value of 2!