RASPBERRY PLASSEMBLER

Roger Ferrer Ibáñez Cambridge, Cambridgeshire, U.K.

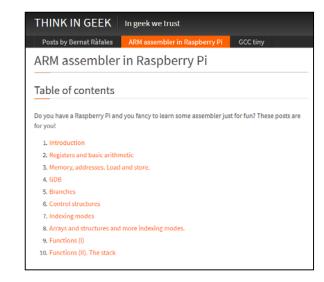
> William J. Pervin Dallas, Texas, U.S.A.

Chapter 11: Raspberry Pi Assembler

"Raspberry Pi Assembler" by R. Ferrer and W. Pervin

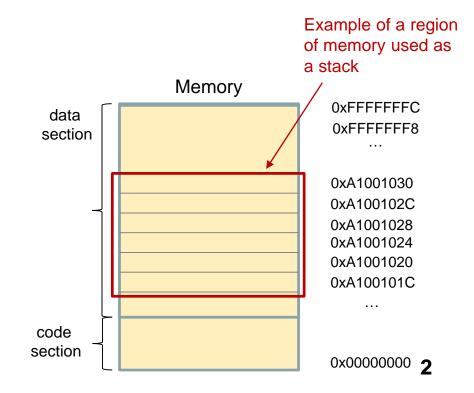
https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/



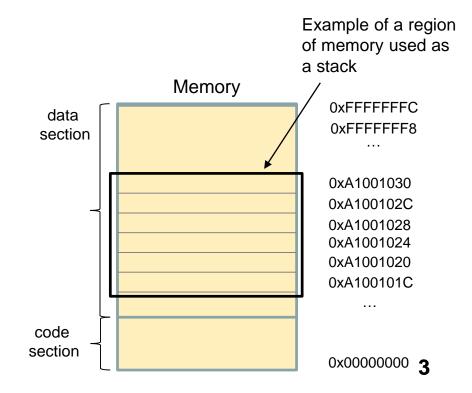


S. Winberg

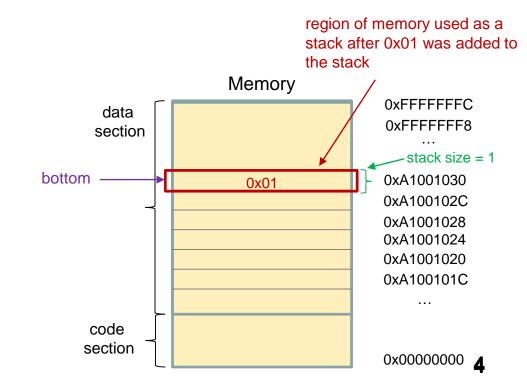
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data



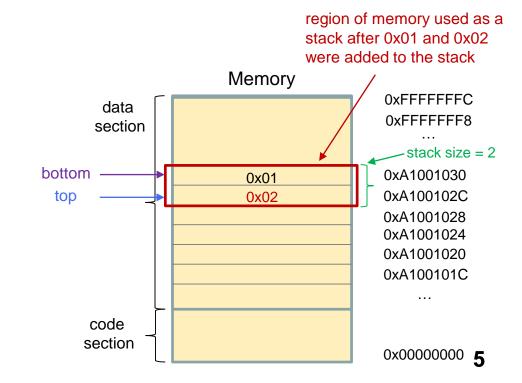
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data
 - Data is stored using a Last-in-first-out (LIFO) buffer



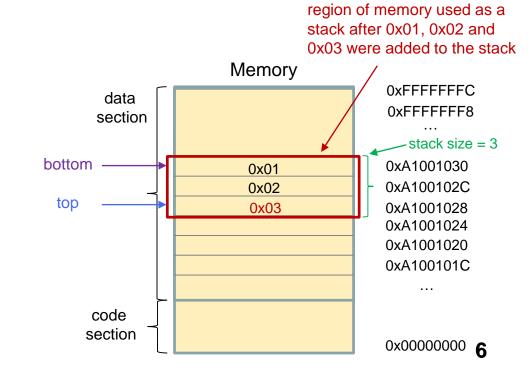
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data
 - Data is stored using a Last-in-first-out (LIFO) buffer
 - Adding data to the stack: for example, add the value 0x01 to the stack.



- 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.



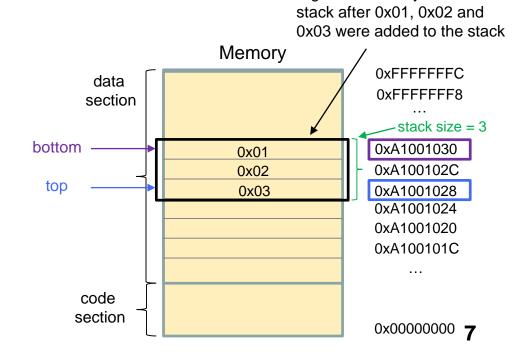
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data
 - Data is stored using a Last-in-first-out (LIFO) buffer
 - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to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



- 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.

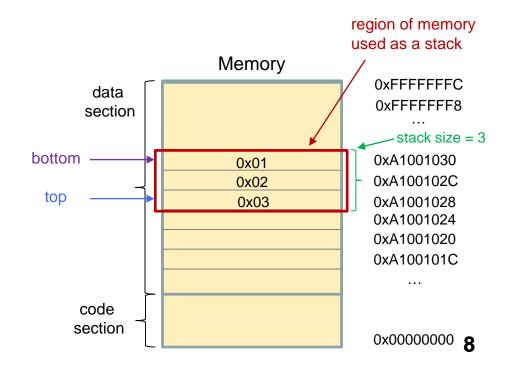
Observations of the stack

- 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

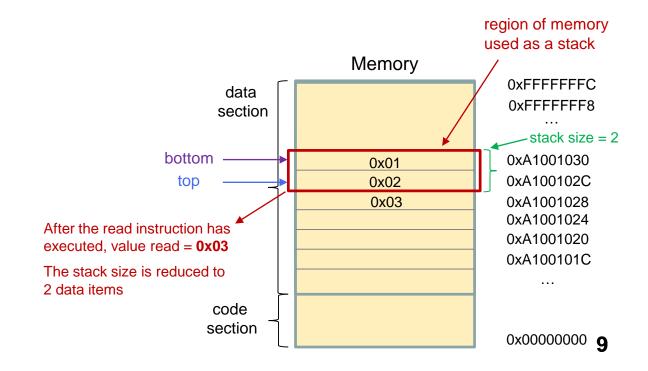


region of memory used as a

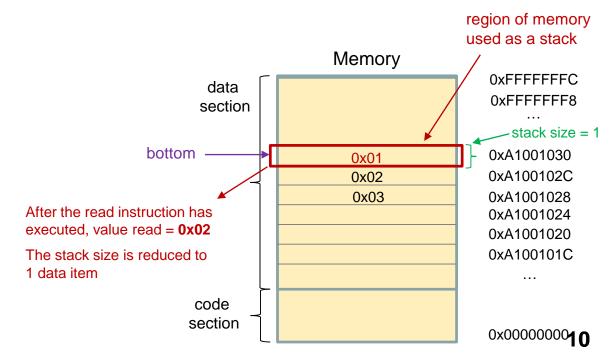
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data
 - Data is stored using a Last-in-first-out (LIFO) buffer
 - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to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



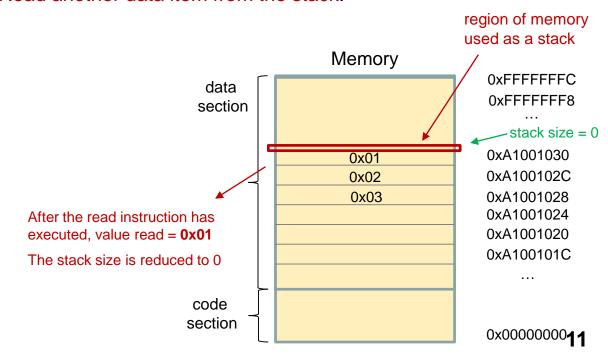
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data
 - Data is stored using a Last-in-first-out (LIFO) buffer
 - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to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



- 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.



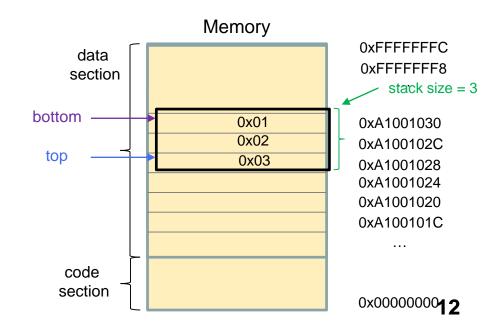
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data
 - Data is stored using a Last-in-first-out (LIFO) buffer
 - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to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



- 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

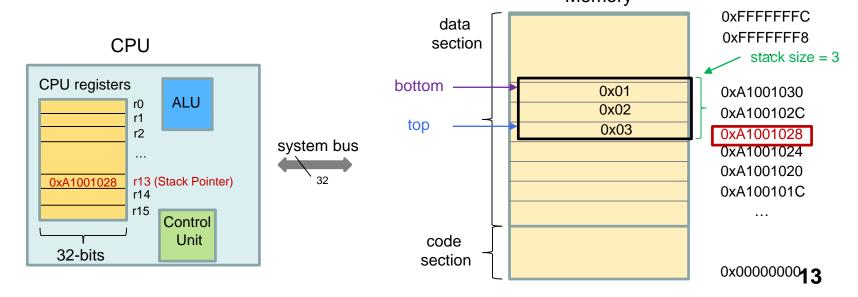
- The data item at the top of the stack is read
- The data item remains in memory, however the size of the stack decreases



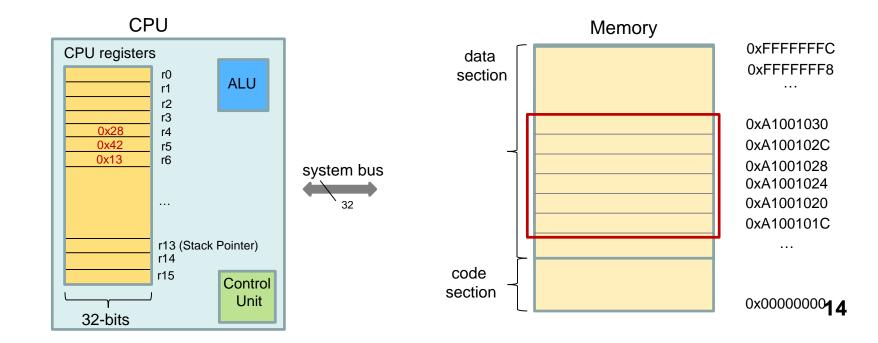
- What is a stack?
 - A stack is a region of memory used by functions to store temporary data
 - Data is stored using a Last-in-first-out (LIFO) buffer
 - Adding data to the stack: for example, add the value 0x01 to the stack. Thereafter, add the value 0x02 to the stack. Lastly, add the value 0x03 to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

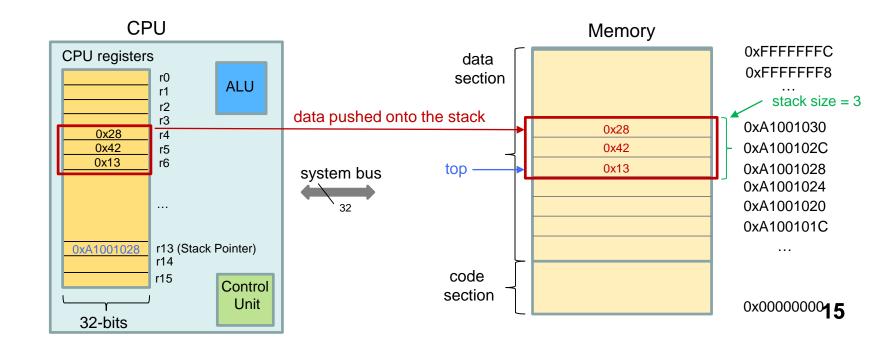
Memory



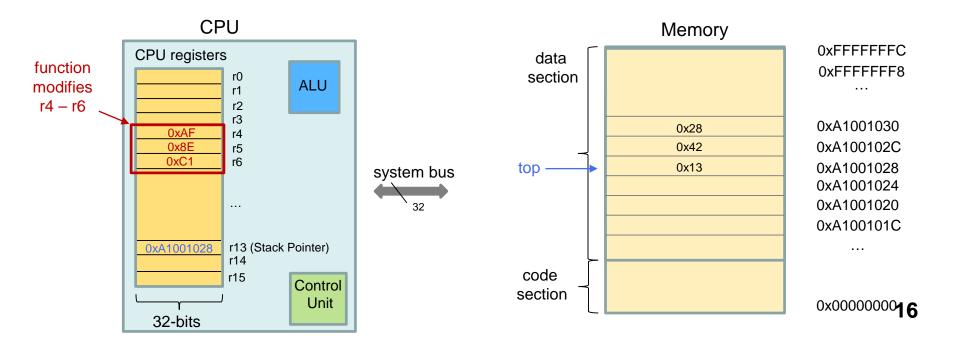
- Why is a stack important?
 - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 r11, however, before the function exits, r4 r11 needs to be restored to their initial values at the start of the function. The following process is used to achieve this:



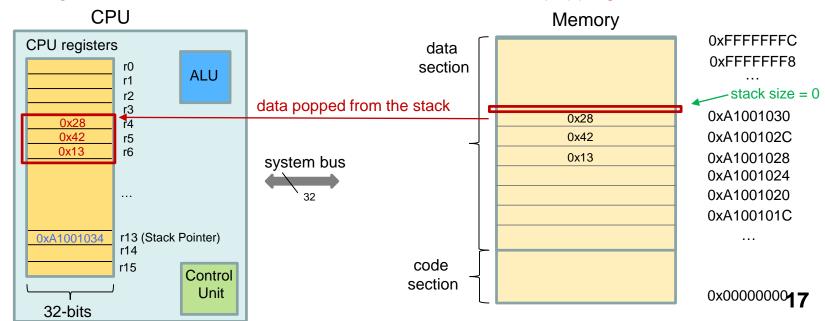
- Why is a stack important?
 - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 r11, however, before the function exits, r4 r11 needs to be restored to their initial values at the start of the function. The following process is used to achieve this:
 - At the start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



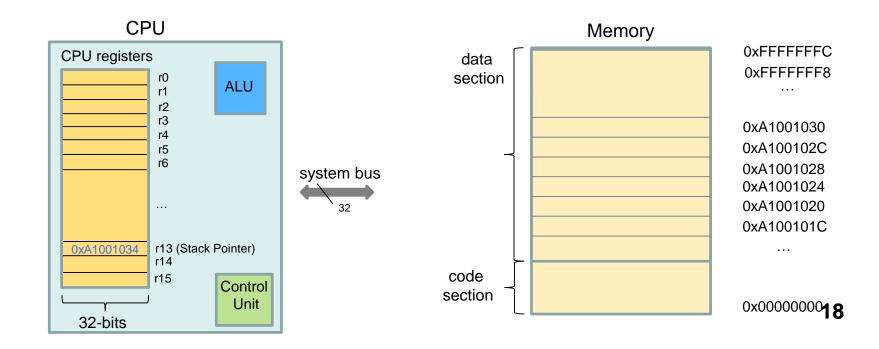
- Why is a stack important?
 - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 r11, however, before the function exits, r4 r11 needs to be restored to their initial values at the start of the function. The following process is used to achieve this:
 - At the start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



- 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.



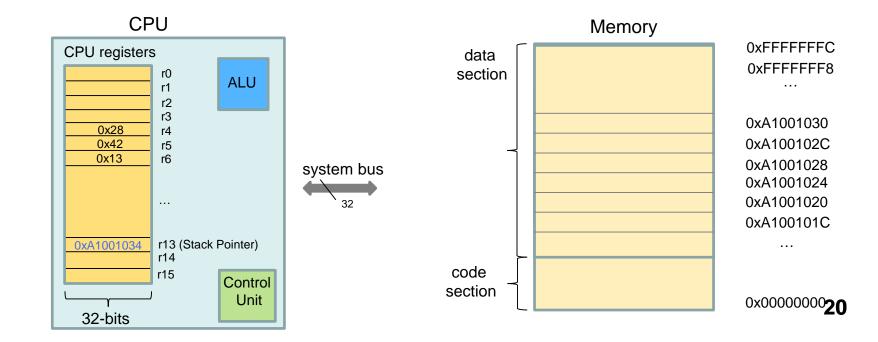
- Why is a stack important?
 - Assembly functions that are AAPCS compliant may modify the contents of CPU registers r4 r11, however, before the function exits, r4 r11 needs to be restored to their initial values at the start of the function.
 - Used in recursive functions to save the Link Register (r14)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



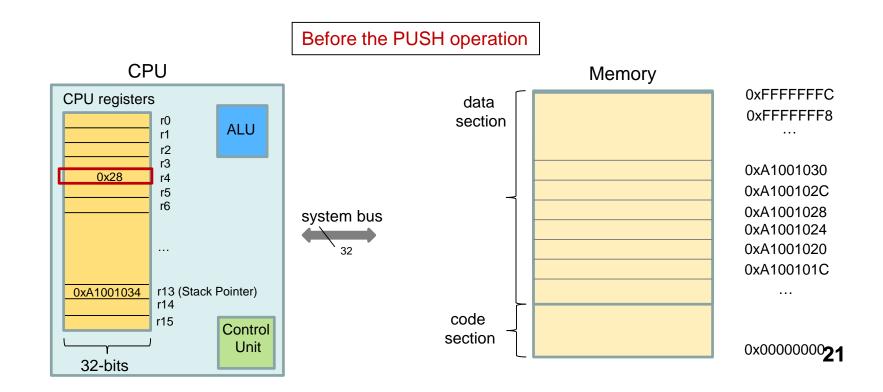
Stack operations



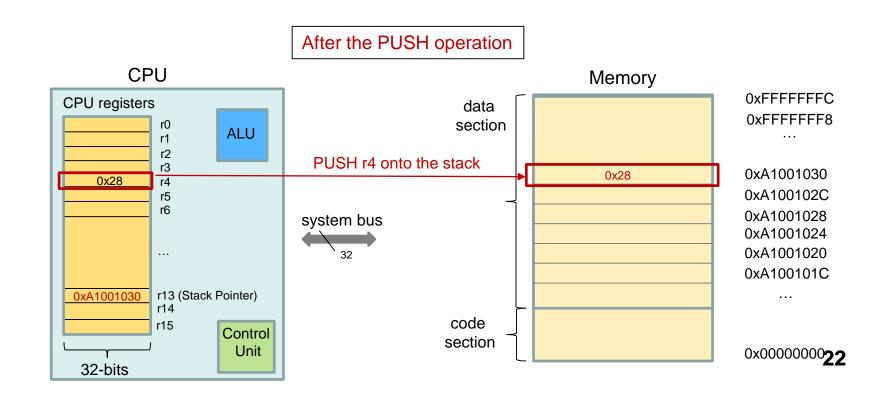
We have identified two stack operations



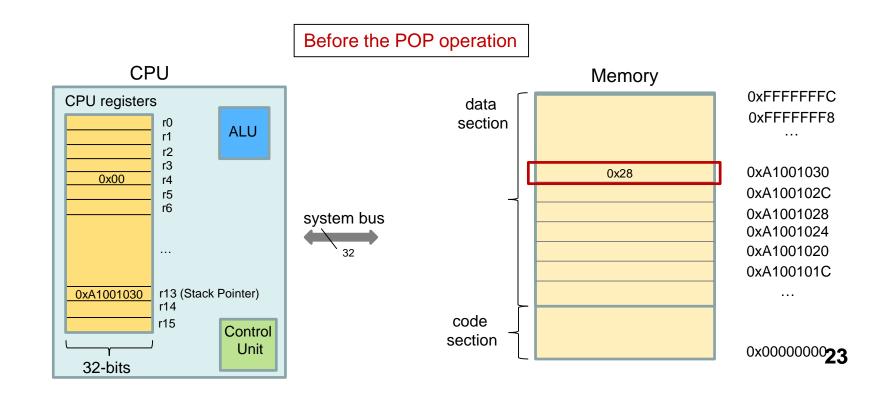
- We have identified two stack operations
 - Adding data to the stack: also referred to as pushing data onto the stack



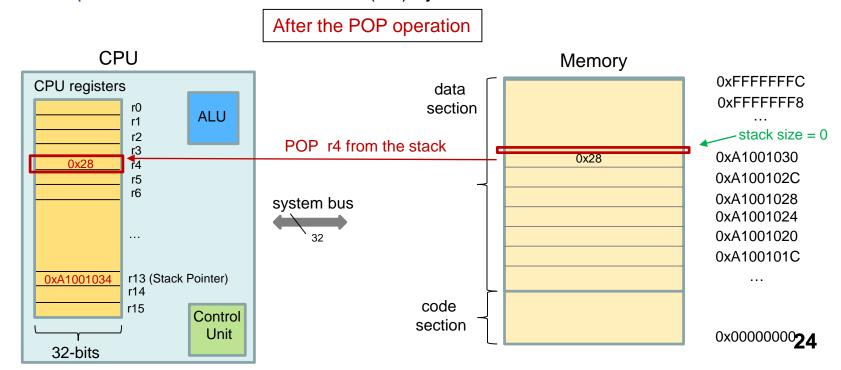
- We have identified two stack operations
 - Adding data to the stack: also referred to as pushing data onto the stack
 - Step 1: Decrement the Stack Pointer (SP) by 4
 - Step 2: Store the CPU register value to the memory address given by the SP



- 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



- 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

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

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

26
```

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, Ir} /* saves the higher number CPU registers first and then the lower ones */

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

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 */
```

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

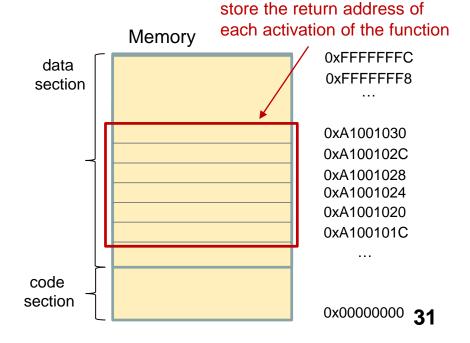
Raspberry Pi Assembler Application of the stack: recursive functions

- 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.

CPU

CPU registers

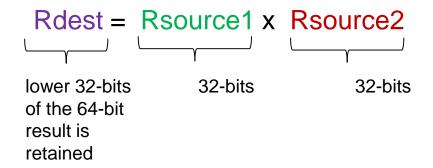
r0
r1
r2
...
r13 (Stack Pointer)
r14 (Link Register)
r15 (PC)
Control
Unit



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

Performs the following operation



Example:

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

• 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
                                            @ Set &format as the first parameter of scanf
                        ldr r0. =format
                        sub sp, sp, #4
                                            @ Make room for one 4 byte integer on the stack
                                            @ We will keep the number entered by the user there
                  section mov r1, sp
                                            @ Set the top of the stack as the second parameter
Memory
                                                        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
                                            @ Discard the integer read by scanf
                        add sp, sp, #+4
                        bl factorial
                                            0 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
                                            @ Call printf
                        bl printf
code section
                        ldr lr, [sp], #+4 @ Pop the top of the stack and put it in lr
                        bx lr
                                            0 Leave main
```

```
/* -- factorial01.s */
.data
                                                             declare strings to be used later in the program
message1: .asciz "Type a number: "
        .asciz "%d"
format:
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
                        @ Set &format as the first parameter of scanf
     ldr r0. =format
    sub sp, sp, #4
                        @ Make room for one 4 byte integer on the stack
                        @ We will keep the number entered by the user there
                        @ Set the top of the stack as the second parameter
    mov r1, sp
                                    of scanf
     bl scanf
                        @ Call scanf
                        @ Load the integer read by scanf into r0
     ldr r0, [sp]
                        @ So we set it as the first parameter of factorial
                        @ Discard the integer read by scanf
     add sp, sp, #+4
                        @ Call factorial
     bl 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
```

0 Leave main

Memory

code section

bx lr

```
/* -- factorial01.s */
.data
                                                                                At the start of the global
message1: .asciz "Type a number: "
                                                                                main function, save the
format: .asciz "%d"
                                                                                contents of the Link
message2: .asciz "The factorial of %d is %d\n"
                                                                                Register onto the stack
.text
.globl main
main:
                                                                                   Can be replaced
    str lr, [sp,#-4]!
                        @ Push lr onto the top of the stack
                                                                                   with one instruction
                                                                                   push lr
    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
                         @ Set the top of the stack as the second parameter
    mov r1, sp
                                     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
                         0 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
                        @ Call printf
     bl printf
    ldr lr, [sp], #+4
                        @ Pop the top of the stack and put it in lr
```

stack

size = 1

Memory

LR at start of .global main

code section

bx lr

0 Leave main

```
/* -- factorial01.s */
                           .data
                          message1: .asciz "Type a number: "
                          format: .asciz "%d"
                          message2: .asciz "The factorial of %d is %d\n"
                                                                                                          prepare the input
                                                                                                          parameter(s) for the printf
                          .text
                                                                                                          instruction by loading the
                           .globl main
                                                                                                          address of the string
                           main:
                                push Ir
                                                   @ Push lr onto the top of the stack
    After instruction
                                                                                                          message to register r0
    has executed:
                               ldr r0, =message1
                                                   @ Set &message1 as the first parameter of printf
    Type a number:
                                                   @ Call printf
                               bl printf
                               ldr r0. =format
                                                   @ Set &format as the first parameter of scanf.
    is displayed to
   the terminal
                               sub sp, sp, #4
                                                   @ Make room for one 4 byte integer on the stack
                                                   @ We will keep the number entered by the user there
 stack
                                                   @ Set the top of the stack as the second parameter
size = 1
                               mov r1, sp
       Memory
                                                               of scanf
                               bl scanf
                                                   @ Call scanf
                               ldr r0, [sp]
                                                   @ Load the integer read by scanf into r0
 LR at start of .global main
                                                   @ So we set it as the first parameter of factorial
                               add sp, sp, #+4
                                                   @ Discard the integer read by scanf
                               bl factorial
                                                   0 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
      code section
                                                   @ Pop the top of the stack and put it in lr
```

0 Leave main

bx lr

```
/* -- factorial01.s */
.data
                                                                             prepare the input
                                                                             parameter(s) for the scanf
message1: .asciz "Type a number: "
                                                                             instruction
format: .asciz "%d"

 r0: format of the input data

message2: .asciz "The factorial of %d is %d\n"

 r1: memory address to

.text
.globl main
main:
                                                                             In this case, we want to save
     push Ir
                        @ Push lr onto the top of the stack
                                                                             the user entered data to the
                                                                             top of the stack
    ldr r0, =message1
                        @ Set &message1 as the first parameter of printf
    bl printf
                        @ Call printf
                         @ Set &format as the first parameter of scanf
     ldr r0. =format
    sub sp, sp, #4
                         @ Make room for one 4 byte integer on the stack
                         @ We will keep the number entered by the user there
                         @ Set the top of the stack as the second parameter
    mov r1, sp
                                    of scanf
     bl scanf
                         @ Call scanf
                         @ Load the integer read by scanf into r0
    ldr r0, [sp]
                        @ So we set it as the first parameter of factorial
                        @ Discard the integer read by scanf
     add sp, sp, #+4
     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
                        @ Load the integer read by scanf into r1
    ldr r1, [sp]
                        @ 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
```

0 Leave main

After instruction

has executed

the data entered

save to the top of

Memory

LR at start of .global main

value entered by user

code section

bx lr

by the user is

the stack

stack

size = 2

40

save user entered data

After instruction

r0 will contain the

factorial value of

the user entered

Memory

LR at start of .global main

value entered by user

code section

bx lr

0 Leave main

data

stack

size = 1

has executed

```
/* -- 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 Ir
                         @ Push lr onto the top of the stack
     ldr r0, =message1
                         @ Set &message1 as the first parameter of printf
                                                                                POP the data item at the top
     bl printf
                         @ Call printf
                                                                                of the stack into register r0
     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
                                                                                prepare the input parameter
                         @ We will keep the number entered by the user there
                                                                                for the factorial function
                         @ Set the top of the stack as the second parameter

    r0: data entered by user

     mov r1, sp
                                     of scanf
     bl scanf
                         @ Call scanf
     ldr r0, [sp]
                         @ Load the integer read by scanf into r0
                                                                                      Can be replaced
                         @ So we set it as the first parameter of factorial
                                                                                      with one instruction
                         @ Discard the integer read by scanf
     add sp, sp, #+4
                                                                                      • pop r0
                         0 Call factorial
     bl factorial
     mov r2, r0
                         @ Get the result of factorial and move it to r2
                         @ So we set it as the third parameter of printf
                         @ Load the integer read by scanf into r1
     ldr r1, [sp]
                         @ 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
                                                                                                    41
     ldr lr, [sp], #+4
                         @ Pop the top of the stack and put it in lr
```

```
/* -- 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 Ir
                         @ Push lr onto the top of the stack
     ldr r0, =message1
                         @ Set &message1 as the first parameter of printf
     bl printf
                         @ Call printf
                         @ Set &format as the first parameter of scanf
     ldr r0. =format
     sub sp, sp, #4
                         @ Make room for one 4 byte integer on the stack
                         @ We will keep the number entered by the user there
                         @ Set the top of the stack as the second parameter
     mov r1, sp
                                     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
                         0 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
                         @ Load the integer read by scanf into r1
     ldr r1, [sp]
                         @ So we set it as the second parameter of printf
                         @ Set &message2 as the first parameter of printf
     ldr r0, =message2
       printf
                         @ Call printf
```

@ Pop the top of the stack and put it in lr

0 Leave main

After instruction has

The factorial of 3 is 6

ldr lr, [sp], #+4

bx lr

is displayed to the

executed

terminal

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

```
/* -- 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 onto the top of the stack
     push Ir
    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
                       @ Set the top of the stack as the second parameter
    mov r1, sp
                                   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
                       0 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
                                                                        restore the Link Register to
                       @ Load the integer read by scanf into r1
    ldr r1, [sp]
                                                                        the value it was at the start
                       @ So we set it as the second parameter of printf
                                                                        of the function
                       @ Set &message2 as the first parameter of printf
    ldr r0, =message2
                                                                                  Can be replaced
    bl printf
                       @ Call printf
                                                                                  with one instruction
    ldr lr, [sp], #+4
                       @ Pop the top of the stack and put it in lr
                                                                                  pop Ir
                       0 Leave main
```

```
/* -- 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 onto the top of the stack
     push Ir
    bl printf
                       @ Call printf
                       @ Set &format as the first parameter of scanf
    ldr r0. =format
    sub sp. sp. #4
                       @ Make room for one 4 byte integer on the stack
                       @ We will keep the number entered by the user there
                       @ Set the top of the stack as the second parameter
    mov r1, sp
                                   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
                       O Discard the integer read by scanf
                       @ Call factorial
    bl 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
                       @ Set &message2 as the first parameter of printf
    ldr r0, =message2
    bl printf
                       @ Call printf
     pop Ir
                       @ Pop the top of the stack and put it in lr
    bx lr
                       @ Leave main
```

After the instruction

The main function

has executed

terminates

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

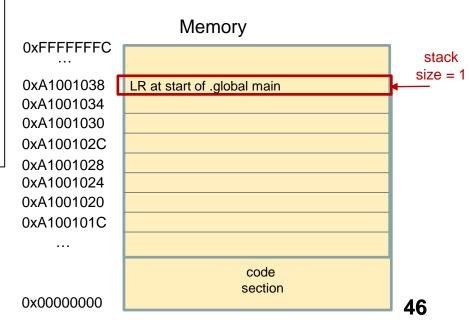
```
factorial:
    str lr, [sp,#-4]! @ Push lr onto the top of the stack
    str r4, [sp,#-4]! @ Push r4 onto the top of the stack
                      @ Keep a copy of the initial value of r0 in r4
   mov r4, r0
    cmp r0, #0
                      @ compare r0 and 0
                      @ if r0 != 0 then branch
    bne is_nonzero
    mov r0, #1
                      @ r0 <- 1. This is the base case; return
      end
                      @ Prepare the call to factorial(n-1)
is nonzero:
                      0 r0 <- r0 - 1
    sub r0, r0, #1
    bl factorial
                      0 After the call r0 contains factorial(n-1)
                      @ Load initial value of r0 (kept in r4) into r1
   mov r1, r4
                      0 r1 <- r4
   mul r0, r0, r1
                      0 r0 <- r0 * r1 [See Project]</pre>
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
                      Q Leave factorial
    bx lr
```



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
                      0 if r0 != 0 then branch
    bne is nonzero
    mov r0, #1
                      @ r0 <- 1. This is the base case; return
    b end
                      @ Prepare the call to factorial(n-1)
is nonzero:
    sub r0, r0, #1
                      0 r0 <- r0 - 1
    bl factorial
                      0 After the call r0 contains factorial(n-1)
                      @ Load initial value of r0 (kept in r4) into r1
   mov r1, r4
                      0 r1 <- r4
   mul r0, r0, r1
                      0 r0 <- r0 * r1 [See Project]</pre>
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 1r
                      Q Leave factorial
```



r0

r1

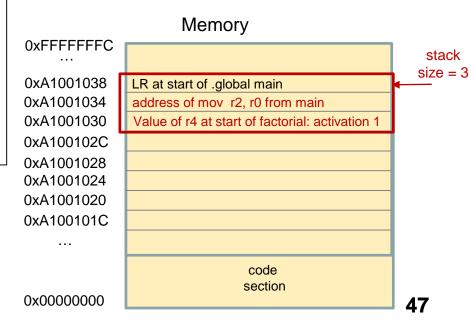
r4

Χ

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

r4
o r1



r0

2

r1

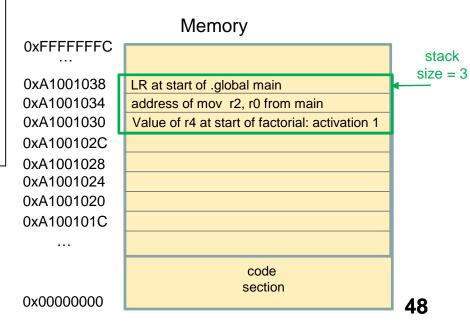
r4

Χ

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 Ir push r4	© Push 1r onto the top of the stack © Push r4 onto the top of the stack
mov r4, r0	0 Keep a copy of the initial value of r0 in r4
cmp r0, #0 bne is_nonzero mov r0, #1 b end	0 if r0 != 0 then branch
is_nonzero: sub r0, r0, #1 bl factorial	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1</pre>
mov r1, r4 mul r0, r0, r1	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4 0 r0 <- r0 * r1 [See Project]</pre>
	<pre>0 Pop the top of the stack and put it in r4 0 Pop the top of the stack and put it in lr 0 Leave factorial</pre>



r0

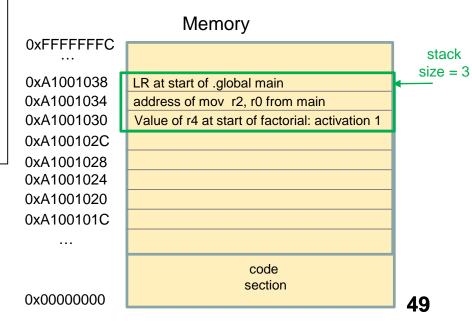
2

r1

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 Ir	© Push lr onto the top of the stack
push r4	Q 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	0 if r0 != 0 then branch
mov r0, #1	0 r0 <- 1. This is the base case; return
b end	
is_nonzero:	@ Prepare the call to factorial(n-1)
sub r0, r0, #1	0 r0 <- r0 - 1
bl factorial	
DI TACCOTTAT	0 16+ +11 -0+
	<pre>0 After the call r0 contains factorial(n-1)</pre>
	© Load initial value of r0 (kept in r4) into r1
mov r1, r4	0 r1 <- r4
mul r0, r0, r1	0 r0 <- r0 * r1 [See Project]
mar 10, 10, 11	2 10 1 10 111 [500 110]500]
4.	
end:	
ldr r4, [sp], #+4	Q 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	0 Leave factorial
DA 11	w neare recovered



r0

2

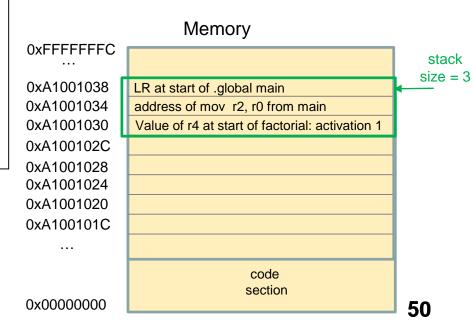
r1

r4

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 Ir push r4	© Push 1r onto the top of the stack © Push r4 onto the top of the stack
mov r4, r0	$\ensuremath{\mathtt{Q}}$ Keep a copy of the initial value of r0 in r4
cmp r0, #0 bne is_nonzero mov r0, #1 b end	
is_nonzero: sub r0, r0, #1	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1</pre>
bl factorial	
mov r1, r4 mul r0, r0, r1	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4 0 r0 <- r0 * r1 [See Project]</pre>
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 1r	<pre>@ Leave factorial</pre>



r0

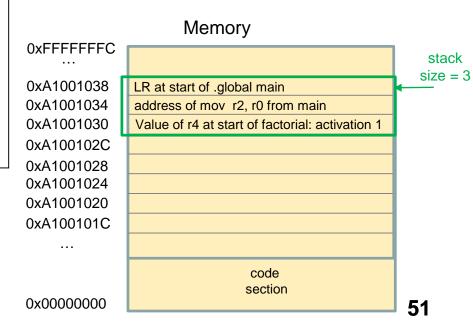
r1

r4

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 Ir	@ 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	0 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	0 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	0 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
_	@ Pop the top of the stack and put it in lr
bx lr	0 Leave factorial
I	



r0

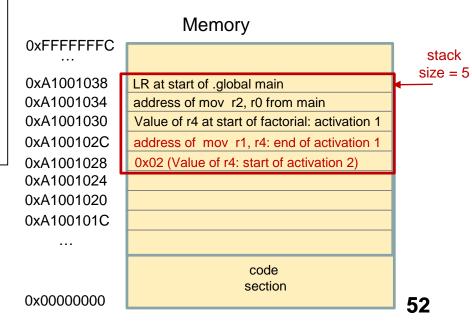
r1

r4

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 1r onto the top of the stack
   push Ir
                       @ Push r4 onto the top of the stack
   push r4
   mov r4, r0
                       @ Keep a copy of the initial value of r0 in r4
    cmp r0, #0
                       @ compare r0 and 0
                       0 if r0 != 0 then branch
    bne is nonzero
    mov r0, #1
                       @ r0 <- 1. This is the base case; return
    b end
                       @ Prepare the call to factorial(n-1)
is nonzero:
    sub r0, r0, #1
                       0 r0 <- r0 - 1
    bl factorial
                       0 After the call r0 contains factorial(n-1)
                      @ Load initial value of r0 (kept in r4) into r1
   mov r1, r4
                       0 r1 <- r4
   mul r0, r0, r1
                       0 r0 <- r0 * r1 [See Project]</pre>
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 1r
                       Q Leave factorial
```



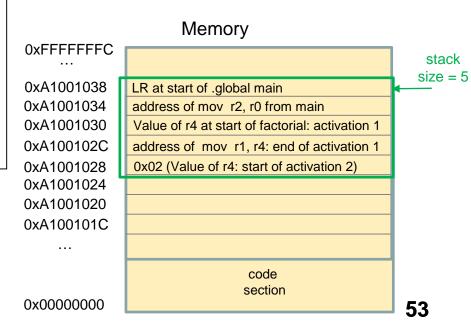
r0

r1

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 Ir push r4	© Push 1r onto the top of the stack © 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 bne is_nonzero mov r0, #1 b end	
is_nonzero: sub r0, r0, #1 bl factorial	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1</pre>
mov r1, r4 mul r0, r0, r1	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4 0 r0 <- r0 * r1 [See Project]</pre>
-	<pre>0 Pop the top of the stack and put it in r4 0 Pop the top of the stack and put it in lr 0 Leave factorial</pre>



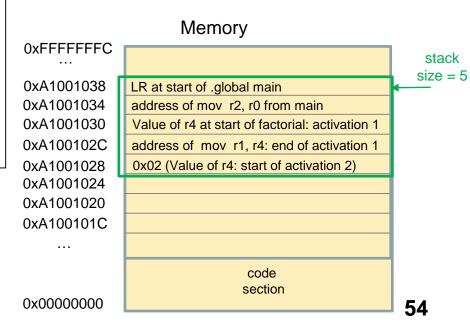
r0

r1

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 Ir push r4	© Push 1r onto the top of the stack © Push r4 onto the top of the stack
mov r4, r0	0 Keep a copy of the initial value of r0 in r4
cmp r0, #0 bne is_nonzero	<pre>0 compare r0 and 0 0 if r0 != 0 then branch</pre>
mov r0, #1 b end	0 r0 <- 1. This is the base case; return
is_nonzero: sub r0, r0, #1 bl factorial	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1</pre>
mov r1, r4	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4</pre>
mul r0, r0, r1	
_	<pre>0 Pop the top of the stack and put it in r4 0 Pop the top of the stack and put it in lr 0 Leave factorial</pre>



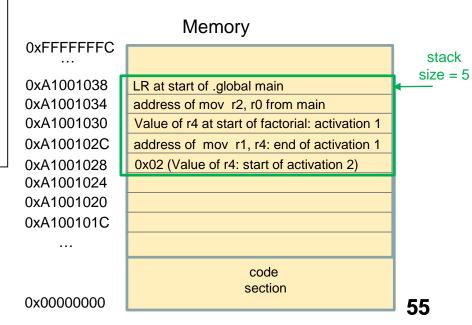
r0

r1

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 Ir	Q Push lr onto the top of the stack
push r4	@ Push r4 onto the top of the stack
parati i i	1
mov r4, r0	@ Keep a copy of the initial value of r0 in r4
	1 17
cmp r0, #0	@ compare r0 and 0
bne is_nonzero	•
	0 r0 <- 1. This is the base case; return
•	w 10 <- 1. This is the base case, lettin
b end	
is_nonzero:	<pre>0 Prepare the call to factorial(n-1)</pre>
sub r0, r0, #1	0 r0 <- r0 - 1
sub r0, r0, #1 bl factorial	0 r0 <- r0 - 1
	<pre>0 r0 <- r0 - 1 0 After the call r0 contains factorial(n-1)</pre>
	@ After the call r0 contains factorial(n-1)
bl factorial	<pre>@ After the call r0 contains factorial(n-1) @ Load initial value of r0 (kept in r4) into r1</pre>
bl factorial mov r1, r4	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4</pre>
bl factorial	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4</pre>
bl factorial mov r1, r4	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4</pre>
bl factorial mov r1, r4	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4</pre>
bl factorial mov r1, r4 mul r0, r0, r1 end:	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4 0 r0 <- r0 * r1 [See Project]</pre>
bl factorial mov r1, r4 mul r0, r0, r1 end: ldr r4, [sp], #+4	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4 0 r0 <- r0 * r1 [See Project]</pre> <pre>0 Pop the top of the stack and put it in r4</pre>
bl factorial mov r1, r4 mul r0, r0, r1 end: ldr r4, [sp], #+4 ldr lr, [sp], #+4	<pre>@ After the call r0 contains factorial(n-1) @ Load initial value of r0 (kept in r4) into r1 @ r1 <- r4 @ r0 <- r0 * r1 [See Project] @ Pop the top of the stack and put it in r4 @ Pop the top of the stack and put it in lr</pre>
bl factorial mov r1, r4 mul r0, r0, r1 end: ldr r4, [sp], #+4	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4 0 r0 <- r0 * r1 [See Project]</pre> <pre>0 Pop the top of the stack and put it in r4</pre>



r0

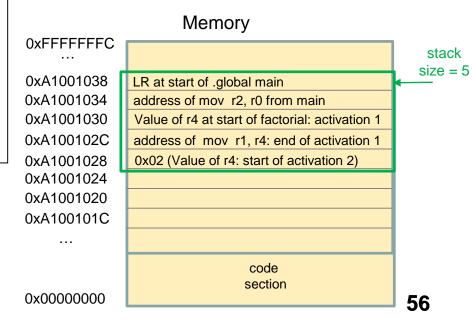
0

r1

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 Ir push r4	<pre>© Push 1r onto the top of the stack</pre> <pre>© Push r4 onto the top of the stack</pre>
mov r4, r0	© Keep a copy of the initial value of r0 in r4
cmp r0, #0 bne is_nonzero mov r0, #1 b end	@ if r0 != 0 then branch
is_nonzero: sub r0, r0, #1	<pre>@ Prepare the call to factorial(n-1) @ r0 <- r0 - 1</pre>
bl factorial	<pre>@ After the call r0 contains factorial(n-1) @ Load initial value of r0 (kept in r4) into r1</pre>
mov r1, r4 mul r0, r0, r1	0 r1 <- r4 0 r0 <- r0 * r1 [See Project]
_	© Pop the top of the stack and put it in r4 © Pop the top of the stack and put it in lr © Leave factorial



r0

0

r1

r4

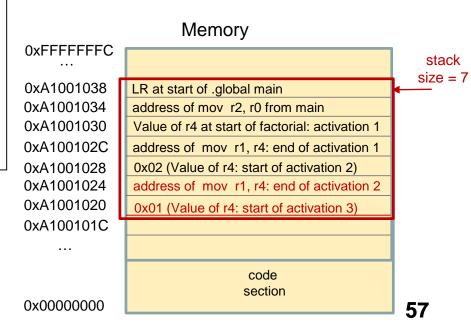
address of mov r1, r4: activation 2

r14 (Link Register)

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 Ir	@ Push lr onto the top of the stack
push r4	© Push r4 onto the top of the stack
	A V of the initial realist of the initial
mov r4, r0	@ Keep a copy of the initial value of r0 in r4
cmp r0, #0	@ compare r0 and 0
bne is_nonzero	Q if rO != O then branch
mov r0, #1	@ r0 <- 1. This is the base case; return
b end	
_	
is_nonzero:	
sub r0, r0, #1	0 r0 <- r0 - 1
bl factorial	
	<pre>0 After the call r0 contains factorial(n-1)</pre>
	Q Load initial value of rO (kept in r4) into r1
mov r1, r4	
mul r0, r0, r1	@ r0 <- r0 * r1 [See Project]
end:	
	A Don the ten of the stock and not it in ma
_	O Pop the top of the stack and put it in r4
	@ Pop the top of the stack and put it in lr
bx 1r	0 Leave factorial



r0

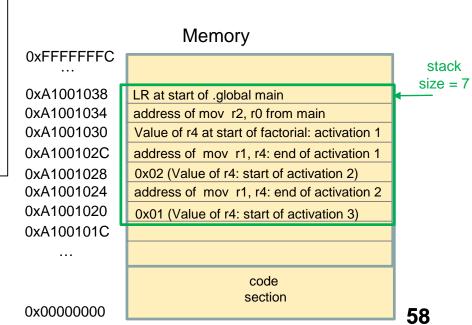
0

r1

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 Ir push r4	© Push 1r onto the top of the stack © 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 bne is_nonzero mov r0, #1 b end	@ if r0 != 0 then branch
is_nonzero: sub r0, r0, #1 bl factorial	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1</pre>
mov r1, r4 mul r0, r0, r1	
_	<pre>0 Pop the top of the stack and put it in r4 0 Pop the top of the stack and put it in lr 0 Leave factorial</pre>



r0

0

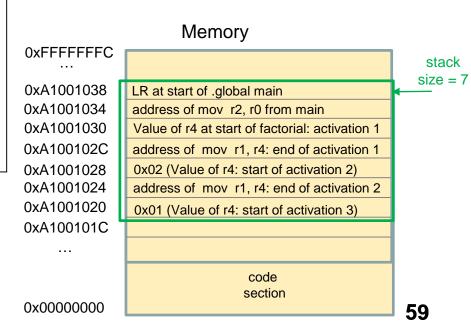
r1

r4

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 Ir push r4	© Push 1r onto the top of the stack © 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 bne is_nonzero	© compare r0 and 0 © if r0 != 0 then branch
mov r0, #1 b end	@ r0 <- 1. This is the base case; return
is_nonzero: sub r0, r0, #1 bl factorial	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1</pre>
	<pre>0 After the call r0 contains factorial(n-1) 0 Load initial value of r0 (kept in r4) into r1</pre>
mov r1, r4 mul r0, r0, r1	0 r1 <- r4 0 r0 <- r0 * r1 [See Project]
end:	
	<pre>@ Pop the top of the stack and put it in r4 @ Pop the top of the stack and put it in lr @ Leave factorial</pre>



r0

0

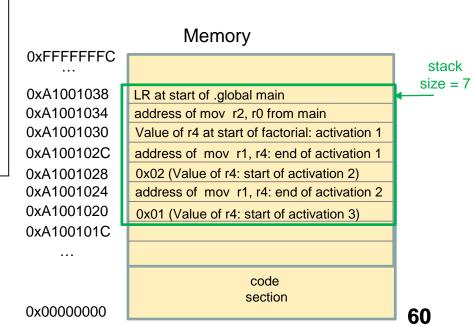
r1

r4

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 1r onto the top of the stack
   push Ir
                       @ Push r4 onto the top of the stack
   push r4
                       @ Keep a copy of the initial value of r0 in r4
   mov r4, r0
    cmp r0, #0
                       @ compare r0 and 0
                       @ if r0 != 0 then branch
    bne is_nonzero
    mov r0, #1
                       @ r0 <- 1. This is the base case; return
       end
                       @ Prepare the call to factorial(n-1)
is nonzero:
    sub r0, r0, #1
                       0 r0 <- r0 - 1
   bl factorial
                       0 After the call r0 contains factorial(n-1)
                       @ Load initial value of r0 (kept in r4) into r1
   mov r1, r4
                       0 r1 <- r4
   mul r0, r0, r1
                       0 r0 <- r0 * r1 [See Project]</pre>
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 1r
                       Q Leave factorial
```



r0

r1

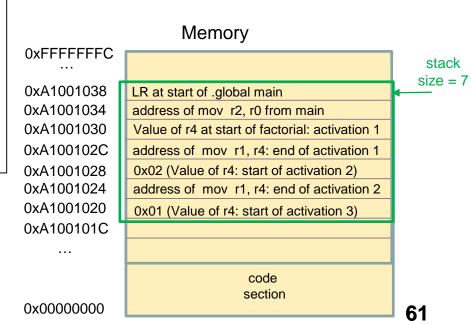
r4

O

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 1r onto the top of the stack
   push Ir
                       @ Push r4 onto the top of the stack
   push r4
                       @ Keep a copy of the initial value of r0 in r4
   mov r4, r0
    cmp r0, #0
                       @ compare r0 and 0
                       0 if r0 != 0 then branch
    bne is_nonzero
    mov r0, #1
                       @ r0 <- 1. This is the base case; return
    b end
                       @ Prepare the call to factorial(n-1)
is nonzero:
    sub r0, r0, #1
                       0 r0 <- r0 - 1
   bl factorial
                       0 After the call r0 contains factorial(n-1)
                       @ Load initial value of r0 (kept in r4) into r1
   mov r1, r4
                       0 r1 <- r4
   mul r0, r0, r1
                       0 r0 <- r0 * r1 [See Project]</pre>
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 1r
                       Q Leave factorial
```



r0

r1

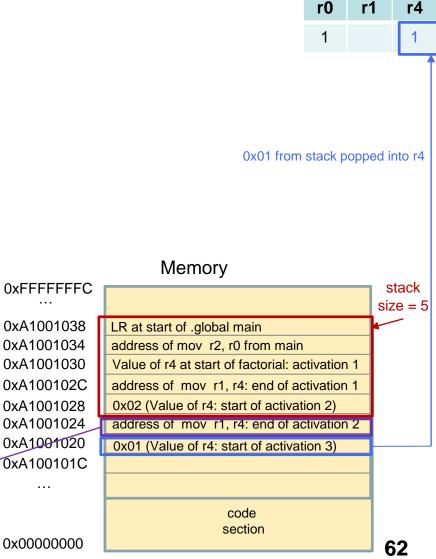
r4

O

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 1r onto the top of the stack
   push Ir
                       @ Push r4 onto the top of the stack
   push r4
                       @ Keep a copy of the initial value of r0 in r4
   mov r4, r0
    cmp r0, #0
                       @ compare r0 and 0
                       0 if r0 != 0 then branch
    bne is_nonzero
    mov r0, #1
                       @ r0 <- 1. This is the base case; return
      end
                       @ Prepare the call to factorial(n-1)
is nonzero:
    sub r0, r0, #1
                       0 r0 <- r0 - 1
   bl factorial
                       0 After the call r0 contains factorial(n-1)
                       @ Load initial value of r0 (kept in r4) into r1
   mov r1, r4
                       0 r1 <- r4
   mul r0, r0, r1
                       0 r0 <- r0 * r1 [See Project]</pre>
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
                       Q Leave factorial
    bx lr
```

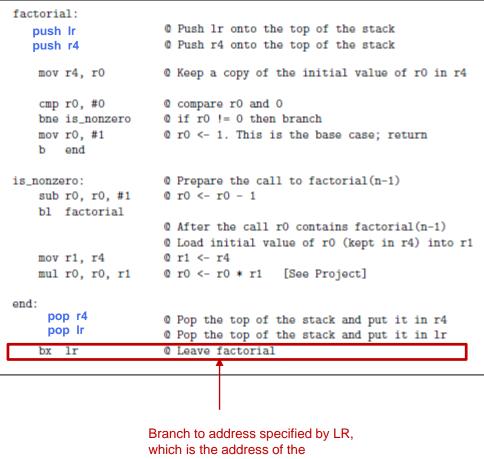


address of mov r1, r4 : activation 2

r14 (Link Register)

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



instruction mov r1, r4 after activation 2 of the factorial function.

address of mov r1, r4: activation 2

Memory 0xFFFFFFC stack size = 50xA1001038 LR at start of .global main 0xA1001034 address of mov r2, r0 from main 0xA1001030 Value of r4 at start of factorial; activation 1 0xA100102C address of mov r1, r4: end of activation 1 0xA1001028 0x02 (Value of r4: start of activation 2) 0xA1001024 address of mov r1, r4: end of activation 2 0xA1001020 0x01 (Value of r4: start of activation 3) 0xA100101C code section 0x00000000 63

r0

r1

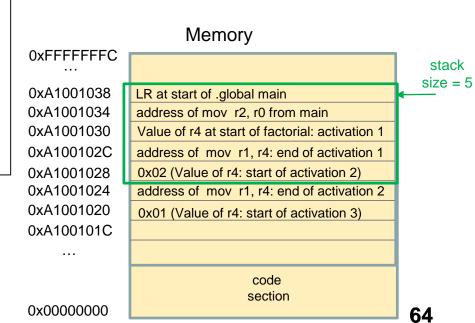
r4

r14 (Link Register)

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 Ir push r4	© Push 1r onto the top of the stack © 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 bne is_nonzero mov r0, #1 b end	0 if r0 != 0 then branch
is_nonzero: sub r0, r0, #1 bl factorial	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1 0 After the call r0 contains factorial(n-1)</pre>
mov r1, r4	<pre>0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4</pre>
mul r0, r0, r1	0 r0 <- r0 * r1 [See Project]
end: pop r4 pop Ir bx 1r	© Pop the top of the stack and put it in r4 © Pop the top of the stack and put it in 1r © Leave factorial



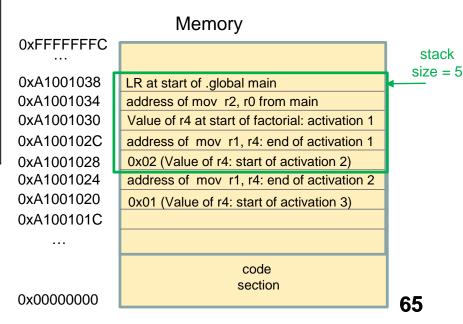
r0

r1

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

١	factorial:	
١	push Ir	@ Push lr onto the top of the stack
١	push r4	@ Push r4 onto the top of the stack
١	•	·
١	mov r4, r0	@ Keep a copy of the initial value of r0 in r4
١		
١	cmp r0, #0	@ compare r0 and 0
١	bne is_nonzero	@ if r0 != 0 then branch
١	mov r0, #1	0 r0 <- 1. This is the base case; return
١	b end	a ro t r. mrs ro one base base, rebarn
١	b ond	
١	is nonzero:	@ Prepare the call to factorial(n-1)
١	sub r0, r0, #1	0 r0 <- r0 - 1
١	bl factorial	w 10 <- 10 - 1
١	DI TACCOFTAI	0 45 +
١		@ After the call r0 contains factorial(n-1)
١		@ Load initial value of r0 (kept in r4) into r1
١	mov r1, r4	0 r1 <- r4
١	mul r0, r0, r1	0 r0 <- r0 * r1 [See Project]
١		
١	end:	
١	pop r4	@ Pop the top of the stack and put it in r4
١	pop Ir	@ Pop the top of the stack and put it in lr
١	bx 1r	Leave factorial
١		

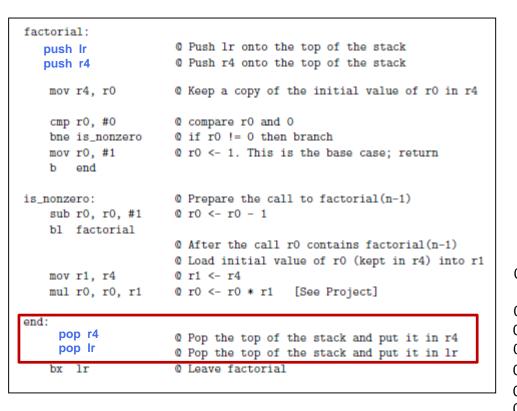


r0

r1

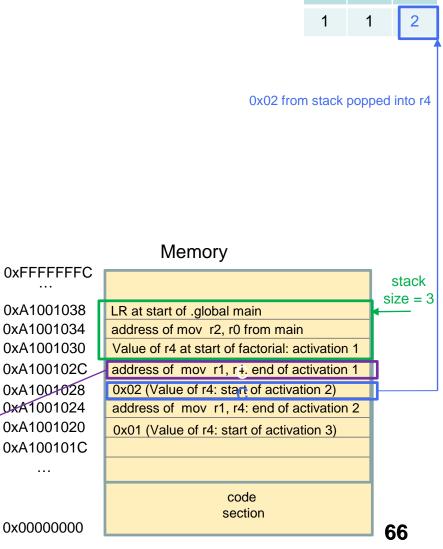
Assume the user entered the value 2. So before the function factorial is called, r0 = 2. After the

factorial function has completed, we want r0 to have the value 2! or 2



r14 (Link Register)

address of mov r1, r4: activation 1

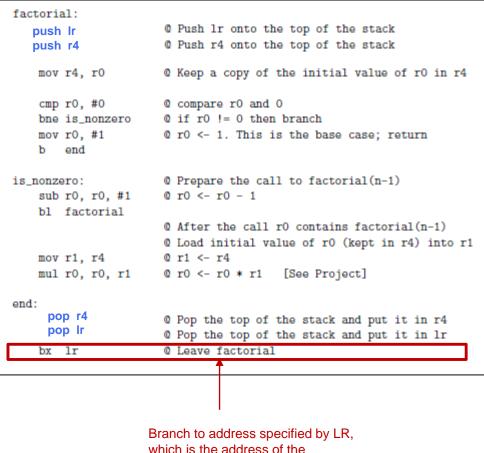


r0

r1

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



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

r14 (Link Register)

address of mov r1, r4; activation 1

Memory 0xFFFFFFC stack size = 30xA1001038 LR at start of .global main 0xA1001034 address of mov r2, r0 from main 0xA1001030 Value of r4 at start of factorial; activation 1 0xA100102C address of mov r1, r4: end of activation 1 0xA1001028 0x02 (Value of r4: start of activation 2) 0xA1001024 address of mov r1, r4: end of activation 2 0xA1001020 0x01 (Value of r4: start of activation 3) 0xA100101C code section 0x00000000 67

r0

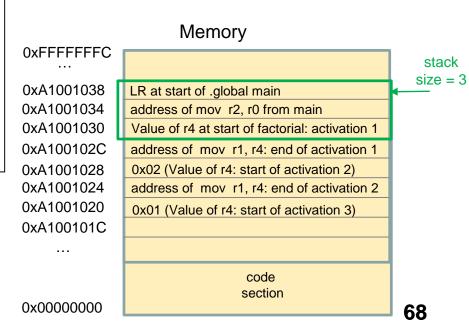
r1

r4

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 Ir push r4	© Push 1r onto the top of the stack © 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 bne is_nonzero mov r0, #1 b end	0 if r0 != 0 then branch
is_nonzero: sub r0, r0, #1 bl factorial	<pre>0 Prepare the call to factorial(n-1) 0 r0 <- r0 - 1 0 After the call r0 contains factorial(n-1)</pre>
mov r1, r4	<pre>0 Load initial value of r0 (kept in r4) into r1 0 r1 <- r4</pre>
mul r0, r0, r1	0 r0 <- r0 * r1 [See Project]
end: pop r4 pop Ir bx 1r	© Pop the top of the stack and put it in r4 © Pop the top of the stack and put it in 1r © Leave factorial



r0

r1

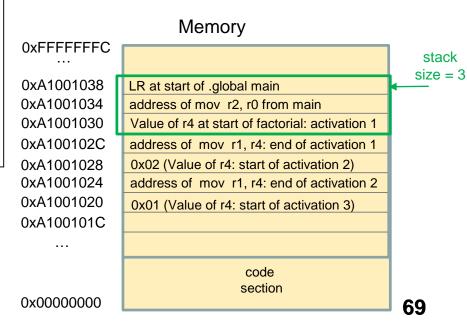
2

r4

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 Ir	Q Push 1r 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	
b ond	
is nonzero:	@ Prepare the call to factorial(n-1)
sub r0, r0, #1	0 r0 <- r0 - 1
bl factorial	W 10 1 10 1
DI TACCOTTAT	@ After the call r0 contains factorial(n-1)
	@ Load initial value of r0 (kept in r4) into r1
mov r1, r4	0 r1 <- r4
mul r0, r0, r1	0 r0 <- r0 * r1 [See Project]
end:	
pop r4	@ Pop the top of the stack and put it in r4
pop Ir	@ Pop the top of the stack and put it in 1r
bx 1r	@ Leave factorial



r0

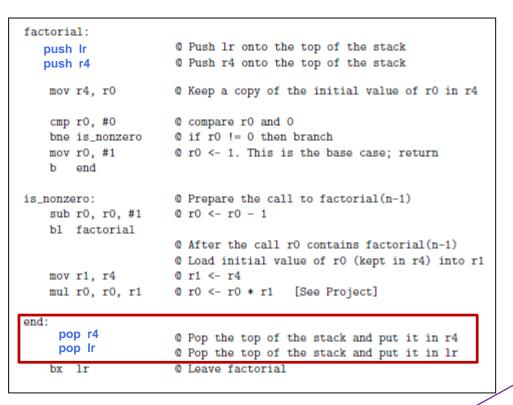
r1

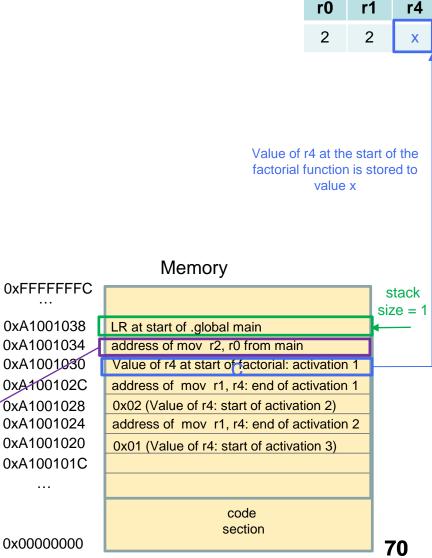
2

r4

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



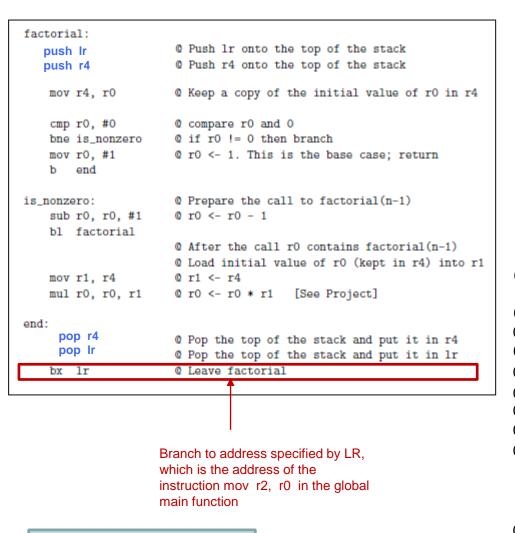


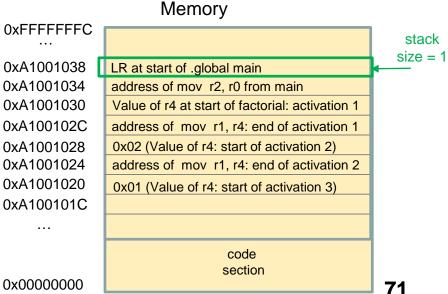
address of mov r2, r0 from main

r14 (Link Register)

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

2

r4

Χ

```
/* -- 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 onto the top of the stack
    push Ir
    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
                       @ Set the top of the stack as the second parameter
    mov r1, sp
                                   of scanf
    bl scanf
                       @ Call scanf
                       @ Load the integer read by scanf into r0
                                                                           Branches back to the main
                       @ So we set it as the first parameter of factorial
     pop r0
                                                                           function with r0 equal to the
                       @ Discard the integer read by scanf
                                                                           value of 2!
    bl factorial
                       0 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
                                                                                              72
                       @ Pop the top of the stack and put it in lr
    ldr lr, [sp], #+4
    bx lr
                       0 Leave main
```