Digital Electronics and Microprocessor Systems (ELEC211)

Dave McIntosh and **Valerio Selis**

dmc@liverpool.ac.uk
V.Selis@liverpool.ac.uk

UNIVERSITY OF LIVERPOOL

Outline

- Subroutines
  - The link register
  - Branch and Link BL
  - Branch and Exchange BX
  - Branch and Link and Exchange
- Nested Subroutines
  - The Stack
  - Push and Pop instructions

Subroutines

Many microprocessor programs include groups of instructions which are repeated many times.

It is wasteful of memory to include these instructions in the program again and again.

Instead, they can be included once in a special structure known as a **subroutine**.

- The main program branches to the start of the subroutine when required.
- At the end of the subroutine, there is another branch back to the main program.

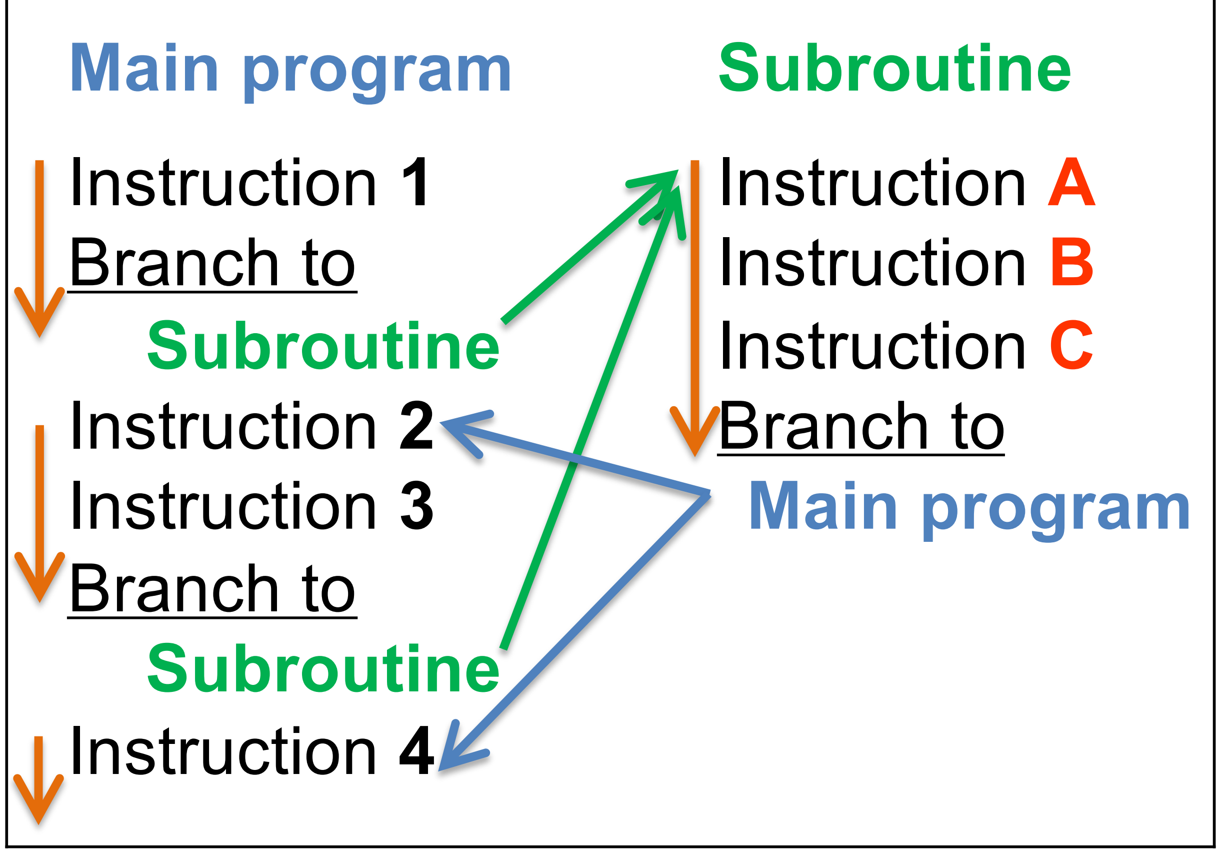
Subroutines

A program requirement is to be able to branch to a subroutine in a way that it is possible to resume the original code sequence

So instead of this:

Instruction **1**
Instruction **A**
Instruction **B**
Instruction **C**
Instruction **2**
Instruction **3**
Instruction **A**
Instruction **B**
Instruction **C**
Instruction **4**

We have this:



Link Register

At the end of the subroutine, how does the processor know which instruction to return to?

This problem is solved by using a ‘link register’ that holds the memory address of the instruction in the main program to which the subroutine returns to.

- The value of the program counter will be saved in the link register
- In the ARM Cortex M0 microprocessor the link register is register ‘r14’
- It can be replaced by ‘lr’ in mnemonics

Branch and link

The mnemonic for ‘branch and link’ is BL

- BL <target_addr>
- Target address can be a label.

The BL instruction uses a 24 bit immediate and it is a 32 bits or 4 bytes machine code.

The range of the destination of the BL instruction is ±16 MiB relative to the current value of the program counter, that is the memory address of the BL instruction.

UNIVERSITY OF LIVERPOOL

Branch and exchange

To return from the subroutine, the value in the link register is moved into the program counter using a 'branch and exchange' instruction (BX lr or   BX r14)

The 'branch and exchange' instruction

BX rz

- Moves the value in register rz into the program counter, r15
- The register rz can be any of the 16 registers
- There is no restriction on the destination of the BX branch (branch to any point in memory)
- **The link register is not updated**

UNIVERSITY OF LIVERPOOL

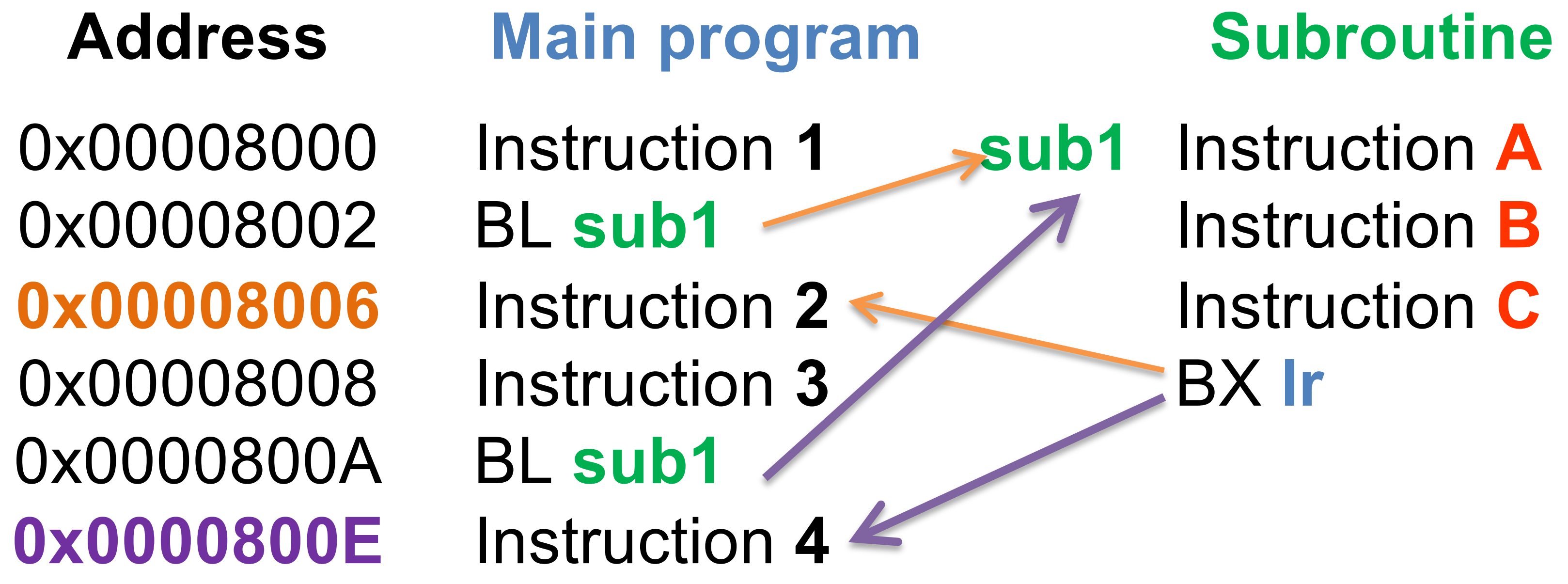
Branch, link and exchange

The ‘branch with link and exchange’ instruction;

BLX rz

- Moves the value in register rz into the program counter, r15
- Updates the link register, r14, with a return address
- The machine code for the BLX instruction is 16 bits or 2 bytes long
- There is no restriction on the destination of the BLX branch (branch to any point in memory)

Subroutines – example



Address	Main program		Subroutine
0x00008000	Instruction **1**	**sub1**	Instruction **A**
0x00008002	BL **sub1**		Instruction **B**
0x00008006	Instruction **2**		Instruction **C**
0x00008008	Instruction **3**		BX **lr**
0x0000800A	BL **sub1**		
0x0000800E	Instruction **4**		

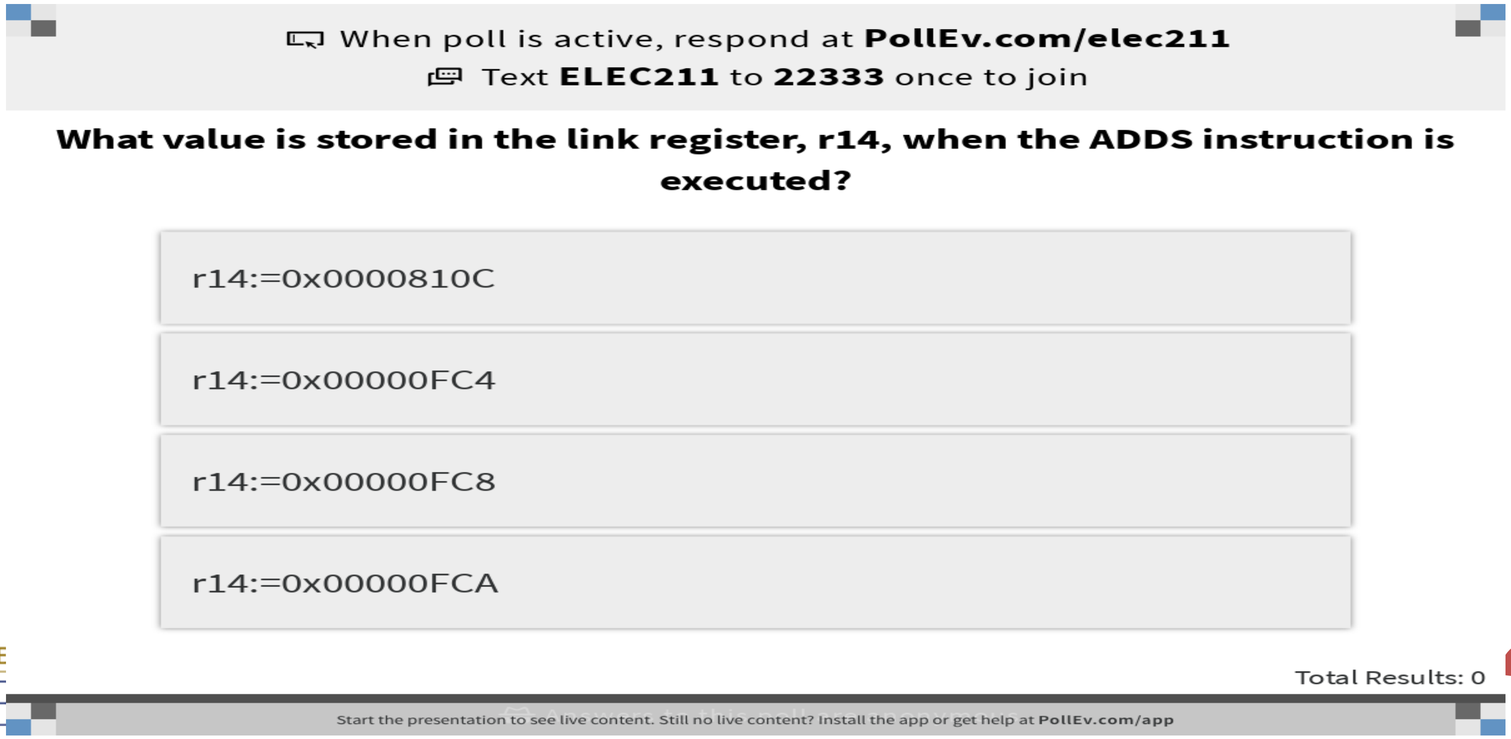
During the **first pass** through the subroutine the link register holds the return address **0x00008006** and during the **second pass** it holds the return address **0x0000800E**.

Question

Address	Instruction
0x00000FC2	MOV r1, #0
0x00000FC4	BL subX
0x00000FC8	MOV r0, #1
0x00000FCA	ADDS r1, r0, #1
0x00000FCC	BL subX
0x00000FD0	SUBS r3, r2, #2

Subroutine

Address	Instruction
subX	BICS r6, r1
0x00008108	ANDS r2, r1
0x0000810A	EORS r5, r6
0x0000810C	BX lr



When poll is active, respond at **PollEv.com/elec211**
Text **ELEC211** to **22333** once to join

What value is stored in the link register, r14, when the ADDS instruction is executed?

r14:=0x0000810C

r14:=0x00000FC4

r14:=0x00000FC8

r14:=0x00000FCA

Total Results: 0

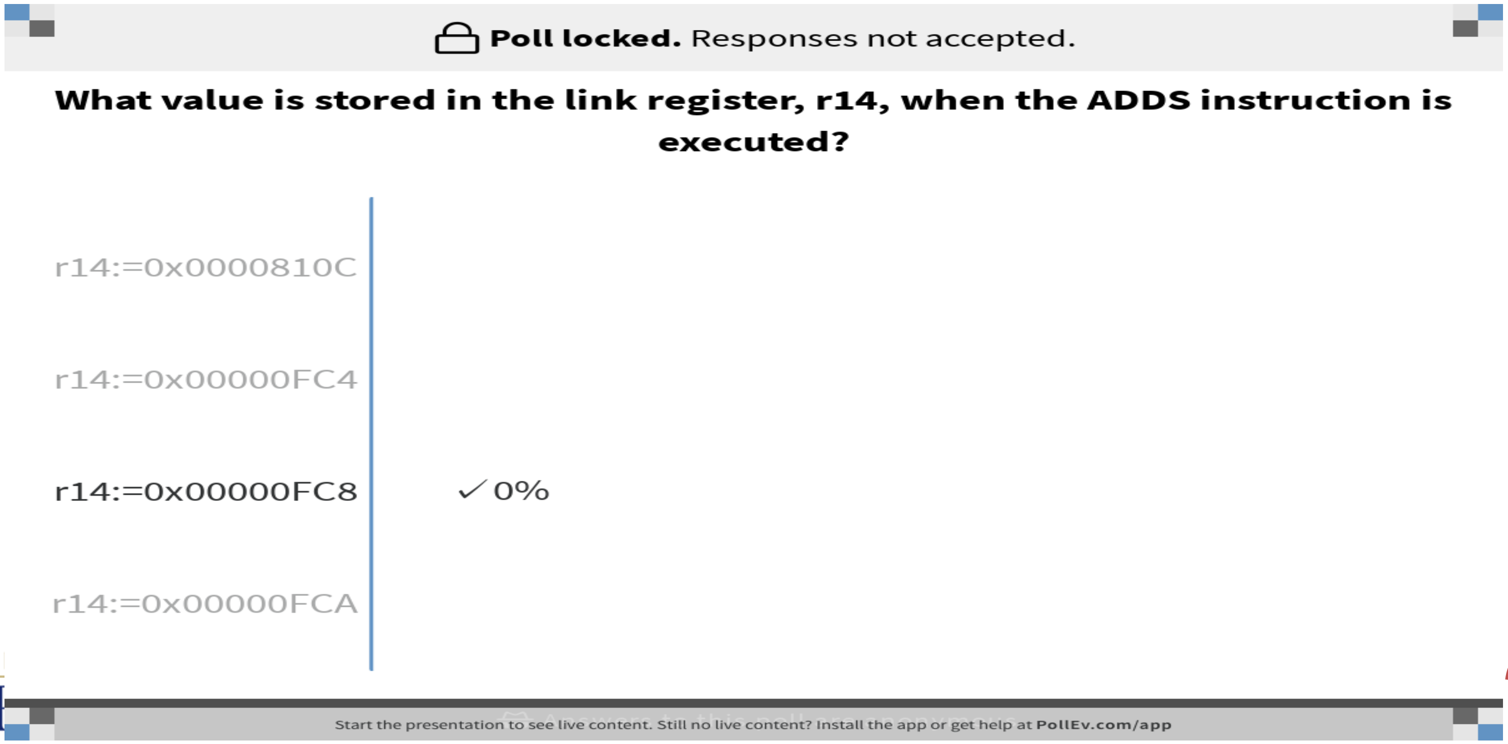
Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Question

Address	Instruction
0x00000FC2	MOV r1, #0
0x00000FC4	BL subX
0x00000FC8	MOV r0, #1
0x00000FCA	ADDS r1, r0, #1
0x00000FCC	BL subX
0x00000FD0	SUBS r3, r2, #2

Subroutine

Address	Instruction
subX	BICS r6, r1
0x00008108	ANDS r2, r1
0x0000810A	EORS r5, r6
0x0000810C	BX lr



Poll locked. Responses not accepted.

What value is stored in the link register, r14, when the ADDS instruction is executed?

r14:=0x0000810C	
r14:=0x00000FC4	
r14:=0x00000FC8	✓ 0%
r14:=0x00000FCA	

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Answer

Address	Instruction
0x00000FC2	MOV r1, #0
0x00000FC4	**BL subX**
0x00000FC8	MOV r0, #1
0x00000FCA	**ADDS** r1, r0, #1
0x00000FCC	**BL subX**
0x00000FD0	SUBS r3, r2, #2

Subroutine

Address	Instruction
subX	BICS r6, r1
0x00008108	ANDS r2, r1
0x0000810A	EORS r5, r6
0x0000810C	BX lr

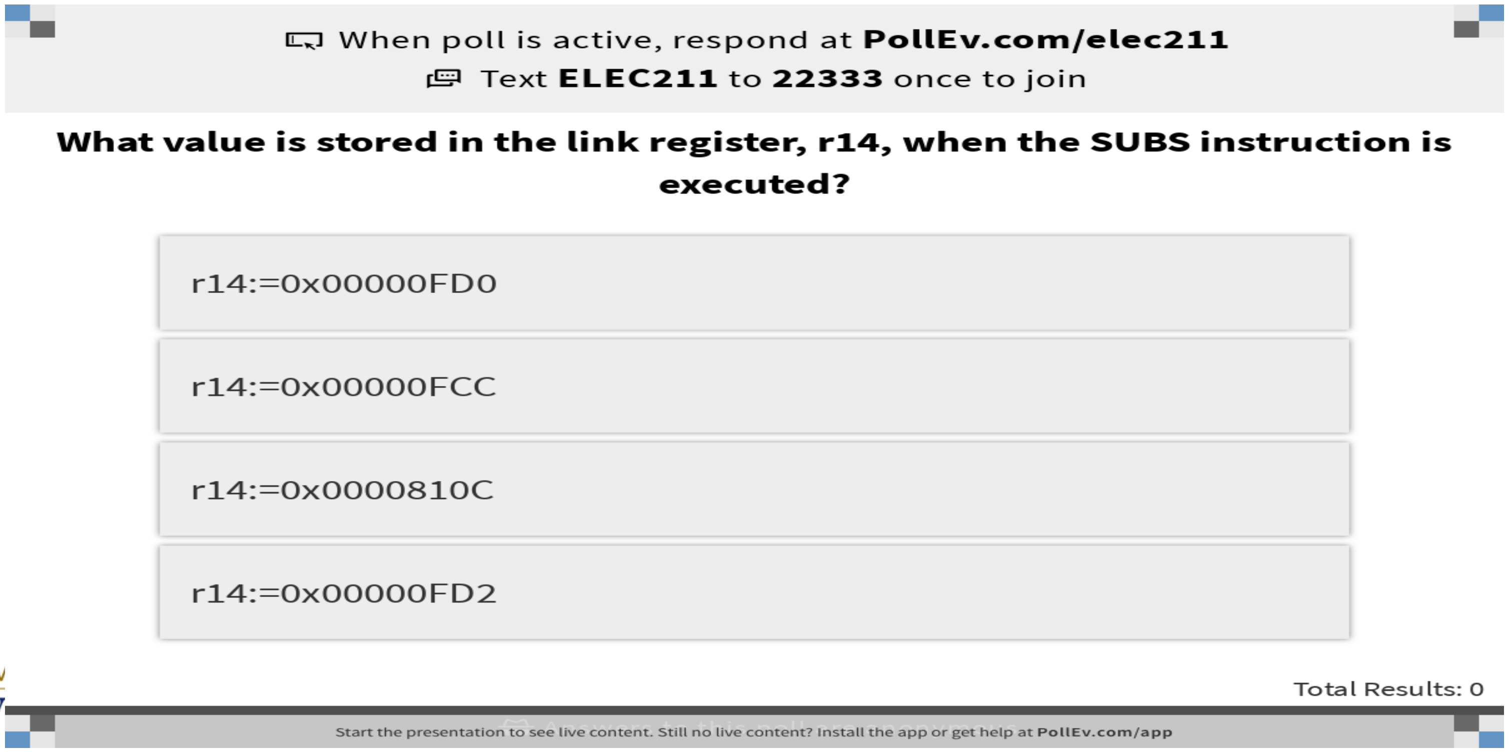
- The 'subX' subroutine is called twice (**BL subX**)
- The **ADDS** instruction is after the first pass through the subroutine but before the second pass.
- So the **link register**, r14, will hold the return address for the first pass, which is **0x00000FC8**.

Question

Address	Instruction
0x00000FC2	MOV r1, #0
0x00000FC4	BL subX
0x00000FC8	MOV r0, #1
0x00000FCA	ADDS r1, r0, #1
0x00000FCC	BL subX
0x00000FD0	SUBS r3, r2, #2

Subroutine

Address	Instruction
subX	BICS r6, r1
0x00008108	ANDS r2, r1
0x0000810A	EORS r5, r6
0x0000810C	BX lr



When poll is active, respond at **PollEv.com/elec211**
Text **ELEC211** to **22333** once to join

What value is stored in the link register, r14, when the SUBS instruction is executed?

- r14:=0x00000FD0
- r14:=0x00000FCC
- r14:=0x0000810C
- r14:=0x00000FD2

Total Results: 0

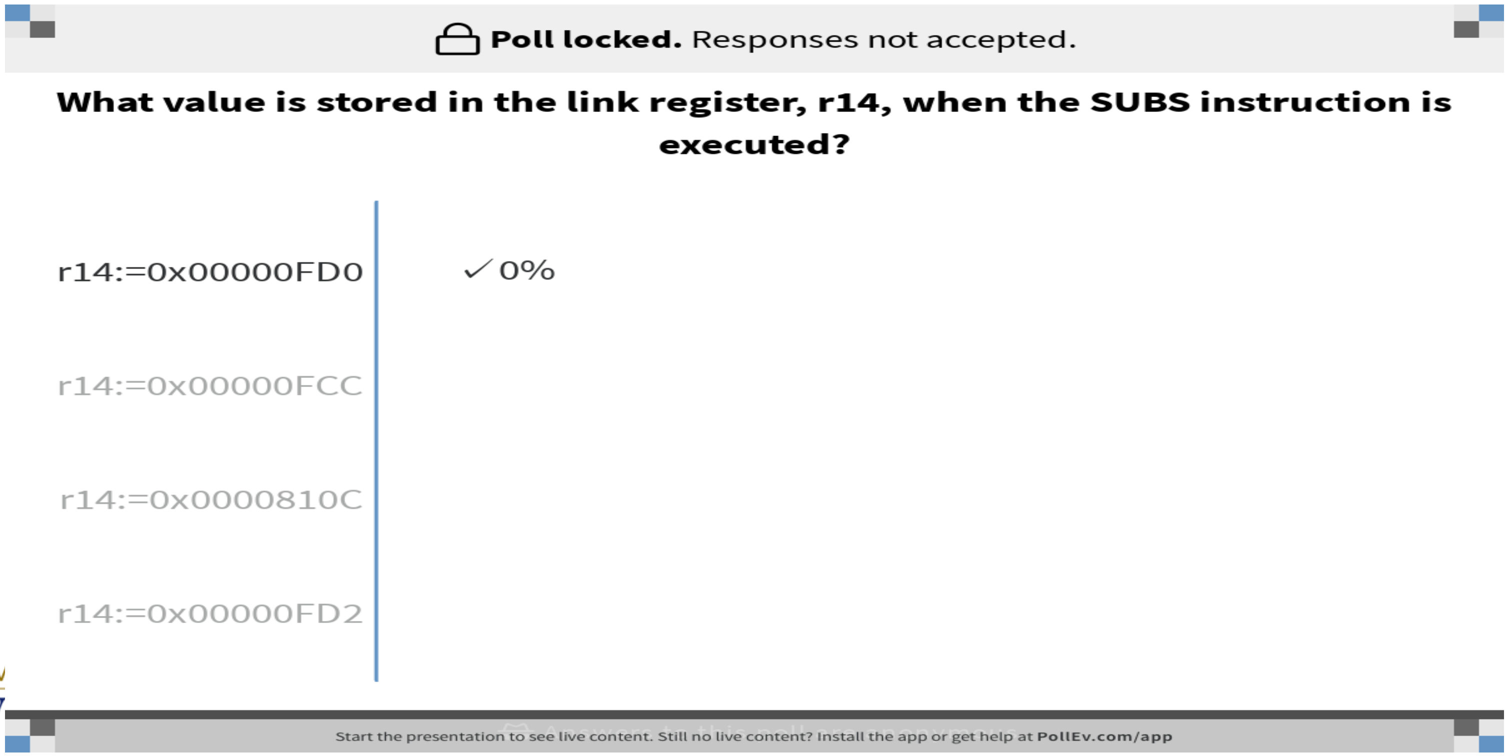
Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Question

Address	Instruction
0x00000FC2	MOV r1, #0
0x00000FC4	BL subX
0x00000FC8	MOV r0, #1
0x00000FCA	ADDS r1, r0, #1
0x00000FCC	BL subX
0x00000FD0	SUBS r3, r2, #2

Subroutine

Address	Instruction
subX	BICS r6, r1
0x00008108	ANDS r2, r1
0x0000810A	EORS r5, r6
0x0000810C	BX lr



Poll locked. Responses not accepted.

What value is stored in the link register, r14, when the SUBS instruction is executed?

r14:=0x00000FD0	✓ 0%
r14:=0x00000FCC	
r14:=0x0000810C	
r14:=0x00000FD2	

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Answer

Address	**Instruction**
0x00000FC2	MOV r1, #0
0x00000FC4	**BL subX**
0x00000FC8	MOV r0, #1
0x00000FCA	ADDS r1, r0, #1
0x00000FCC	**BL subX**
0x00000FD0	**SUBS** r3, r2, #2

Subroutine

Address	**Instruction**
subX	BICS r6, r1
0x00008108	ANDS r2, r1
0x0000810A	EORS r5, r6
0x0000810C	BX lr

- The 'subX' subroutine is called twice (**BL subX**)
- The **SUBS** instruction is after the second pass through the subroutine.
- So the **link register**, r14, will hold the return address for the second pass, which is **0x00000FD0**.

Nested subroutines

What happens if a branch and link (BL or BLX) occurs during execution of a subroutine?

The value held in the link register will be overwritten by a new return address

- Before one subroutine calls another subroutine, **the link register value must be stored elsewhere**

The link register can only hold one return address at a time

- If subroutine A calls subroutine B the link register can not store the return address for both subroutines at the same time

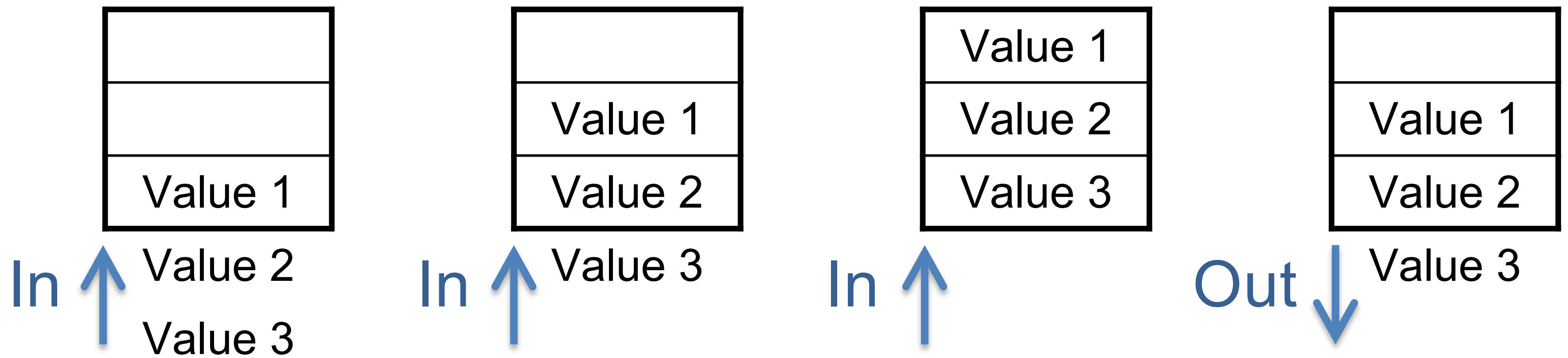
Nested subroutines

Complicated programs will have several subroutines

- Some subroutines will call other subroutines
- This is a standard practice and it is known as nesting

In order to preserve the return addresses of all subroutines, an area of computer memory called the **memory stack** is used

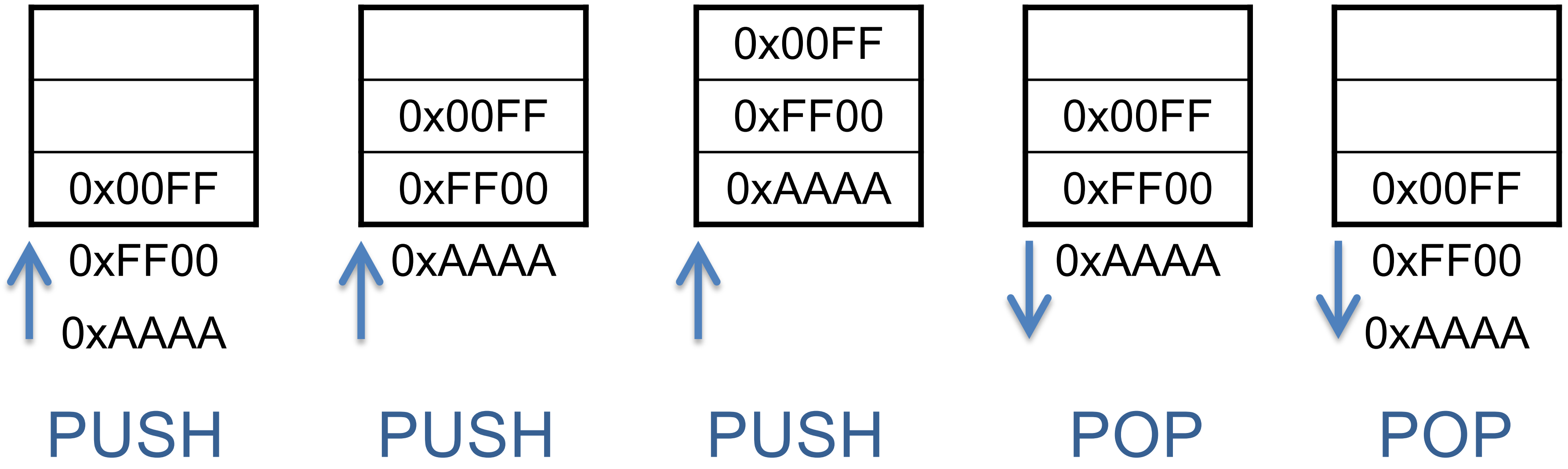
- The stack is a Last In First Out (**LIFO**) queue



The stack

The stack is a LIFO, that means whatever data was added to the stack (‘pushed’) last is taken from the stack (‘popped’) first.

- E.g., if the values pushed onto a stack where 0x00FF, 0xFF00, 0xAAAA in that order then they would be popped from the stack in the reverse order



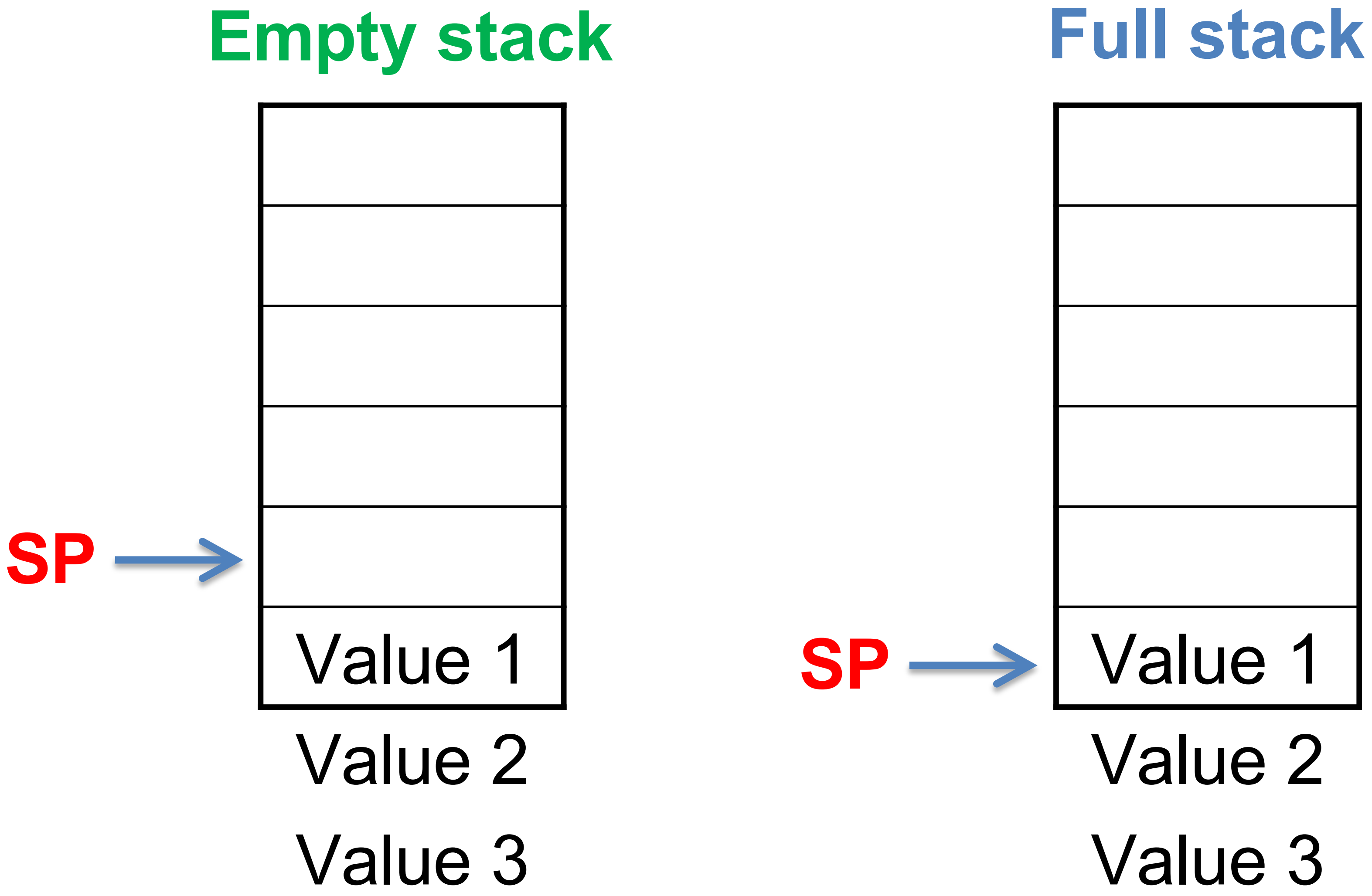
The stack pointer

The **stack pointer** holds an address in memory that identifies the top of the stack.

The address is either the location of **the last data** to be pushed onto the stack or alternatively the location of the **next empty slot** where the next data can be placed.

These two alternatives are known as a ‘**full stack**’ and an ‘**empty stack**’ and the ARM Cortex M0 microprocessor uses a **full stack**.

The stack pointer



Ascending and descending stacks

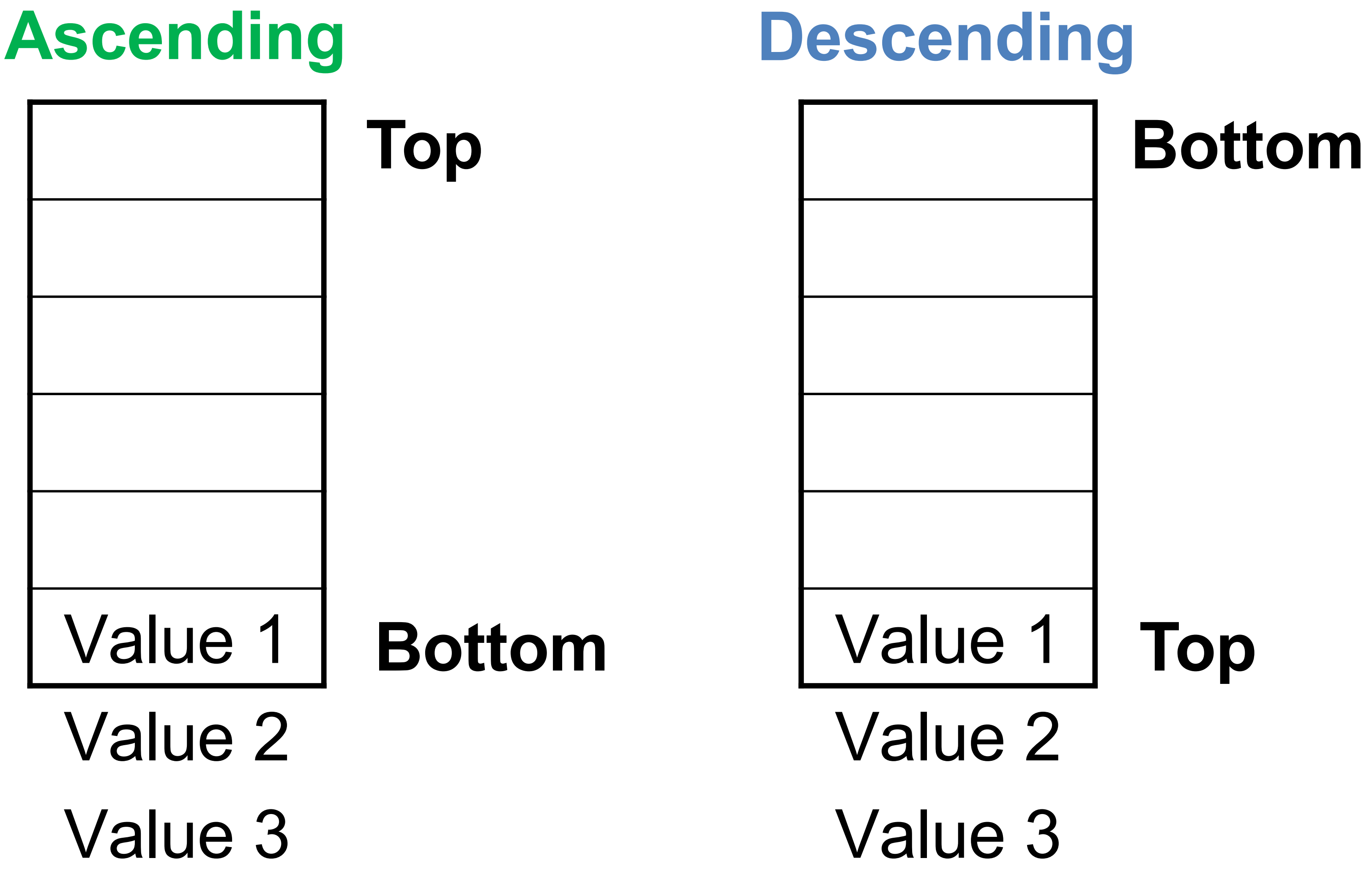
Stacks can be either ascending or descending.

- For a descending stack, the memory address of the top of the stack is less than the memory address for the bottom of the stack.

When data is pushed onto a descending stack, the value held by the stack pointer decreases and when data is popped from a descending stack it increases.

Ascending stacks work in the opposite way; the top is at a higher memory address than the bottom.

Ascending and descending stacks



Stacks and the ARM Cortex M0

The ARM Cortex M0 uses a descending stack.

- Register r13 is used as the stack pointer.

To push the link register value onto a full descending stack the mnemonic is:

```
PUSH {lr}
```

To pop a value from a full descending stack, back into the link register the mnemonic is:

```
POP {lr}
```

Although this instruction is not allowed… (why?)

Stacks and the ARM Cortex M0

The first instruction in a subroutine is generally:

```
PUSH {lr}
```

Then any branch and link in that subroutine can overwrite the link register.

The subroutine ends by popping the return address from the stack into the program counter:

```
POP {pc}
```

POP {pc} is equivalent to POP {lr} followed by moving the value in the link register into the program counter (BX lr). Therefore POP {lr} is not required.

Stacks and the Cortex M0

It is common practice to pop the return address directly into the program counter using:

```
POP {pc}
```

Pushing and popping the stack can be achieved with several registers at the same time.

The low registers, r0 to r7, can be used in addition to the link register, lr or r14, with PUSH.

The low registers, r0 to r7, can be used in addition to the program counter, pc or r15, with POP.

Stacking other registers

Again if a subroutine overwrites any register, not just the link register, then the value held in that register:

1. Can be pushed onto the stack at the start of the subroutine
2. Can be popped from the stack at the end of the subroutine

To push registers r1, r5, r6 and r7 with the link register use:

```
PUSH {r1, r5-r7, lr}
```

To pop the same registers use:

```
POP {r1, r5-r7, pc}
```

Stacking other registers

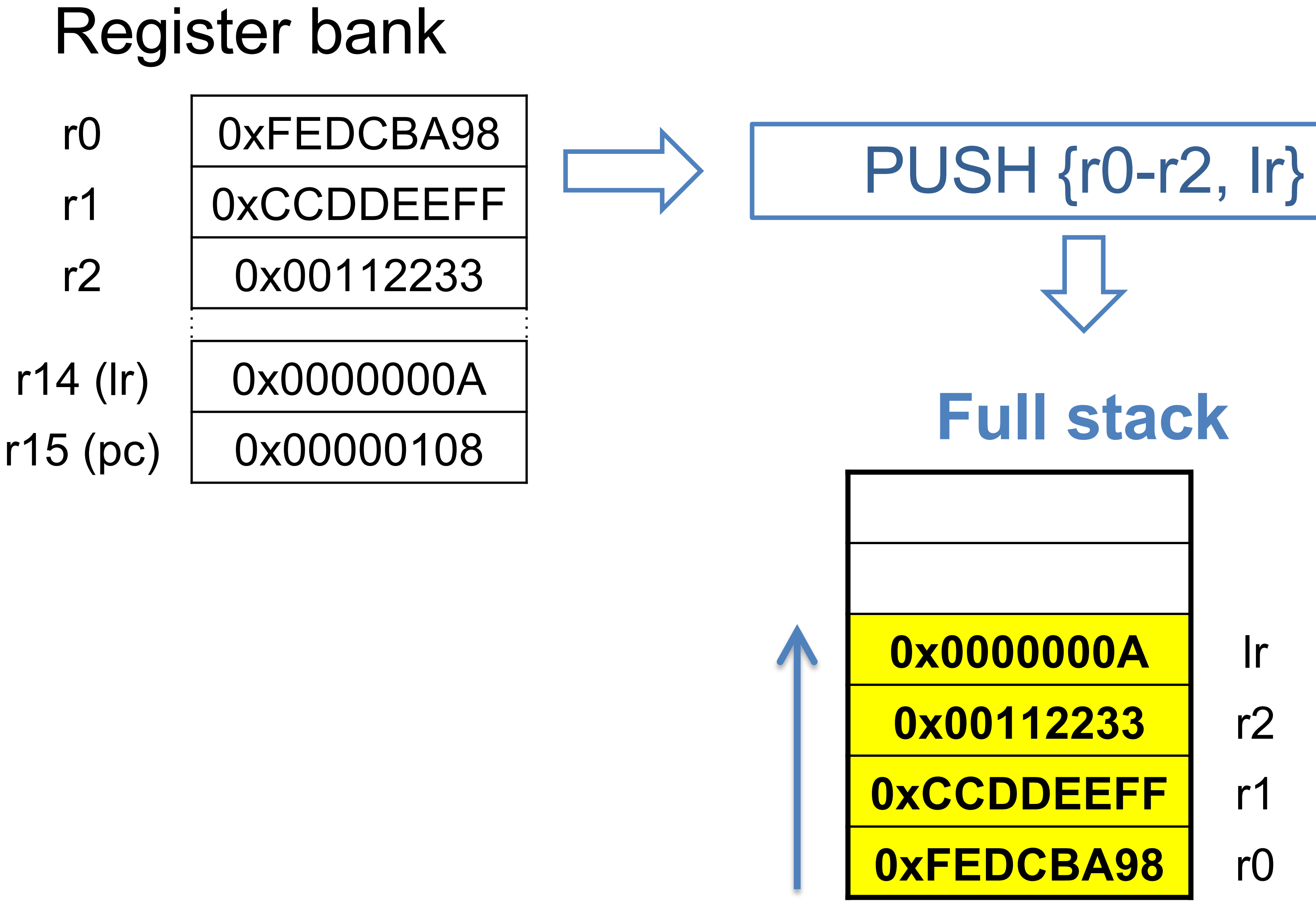
Great care must be taken when using push and pop to ensure that the number of registers in the POP is the same as the number in the PUSH.

For example, for the following

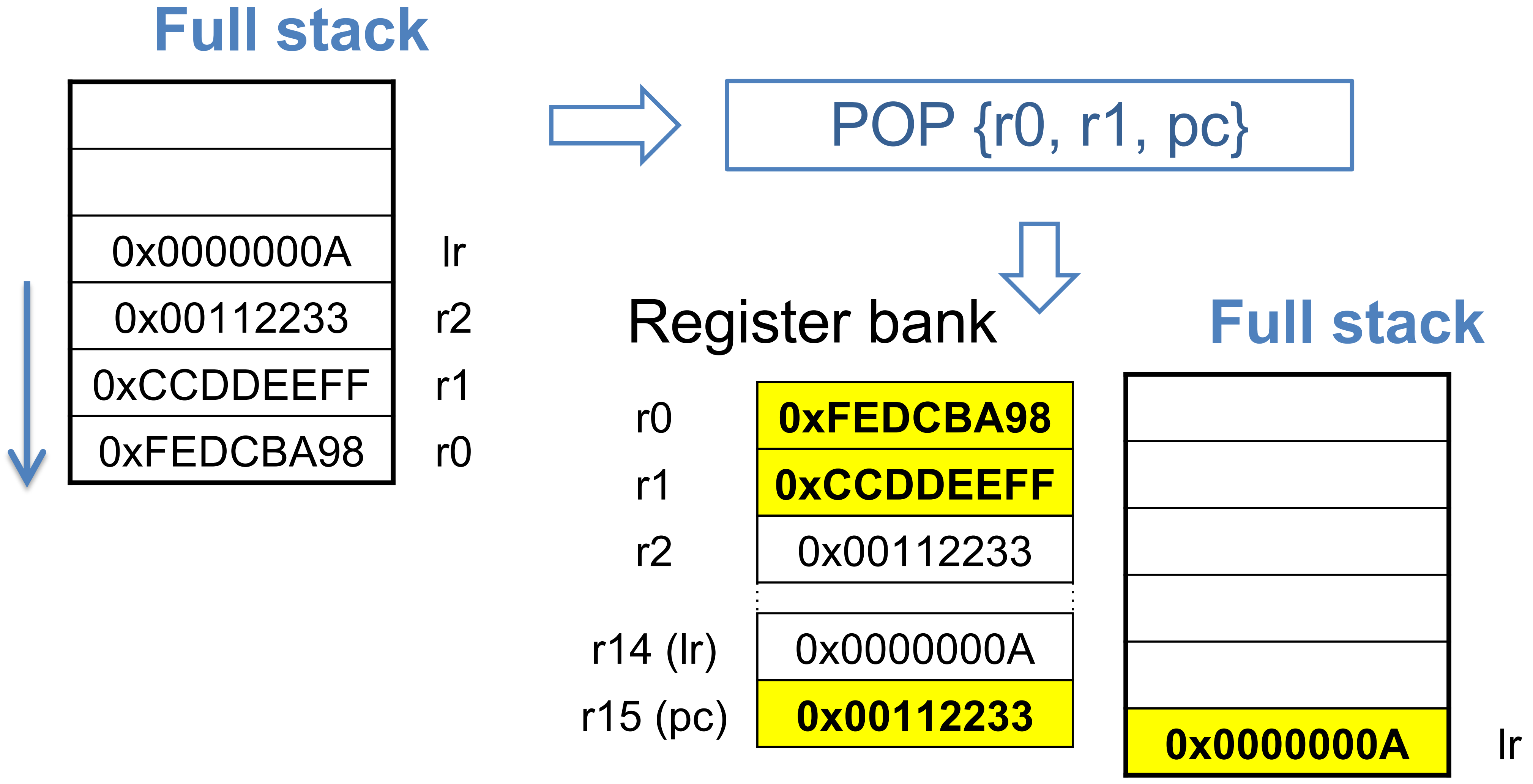
- PUSH {r0-r2, lr}
- some instructions....
- POP {r0, r1, pc}

The program counter is loaded with the value previously held in register r2, not the return address previously held in the link register.

Stacking other registers – take care!



Stacking other registers – take care!



Error, the pc is loaded with the value previously held by r2, not the return address previously held in the lr

Recommended reading

ARM system-on-chip architecture

- Section 5.5: Branch, Branch with Link and Exchange (B, BL, BLX)

Summary

- Subroutines
- Link register and branches
- Nested subroutines
- Stack (PUSH and POP instructions)

Next class?

Tomorrow at 2 p.m. in the Building 502,
Lecture Theatre 2
(502-LT2)