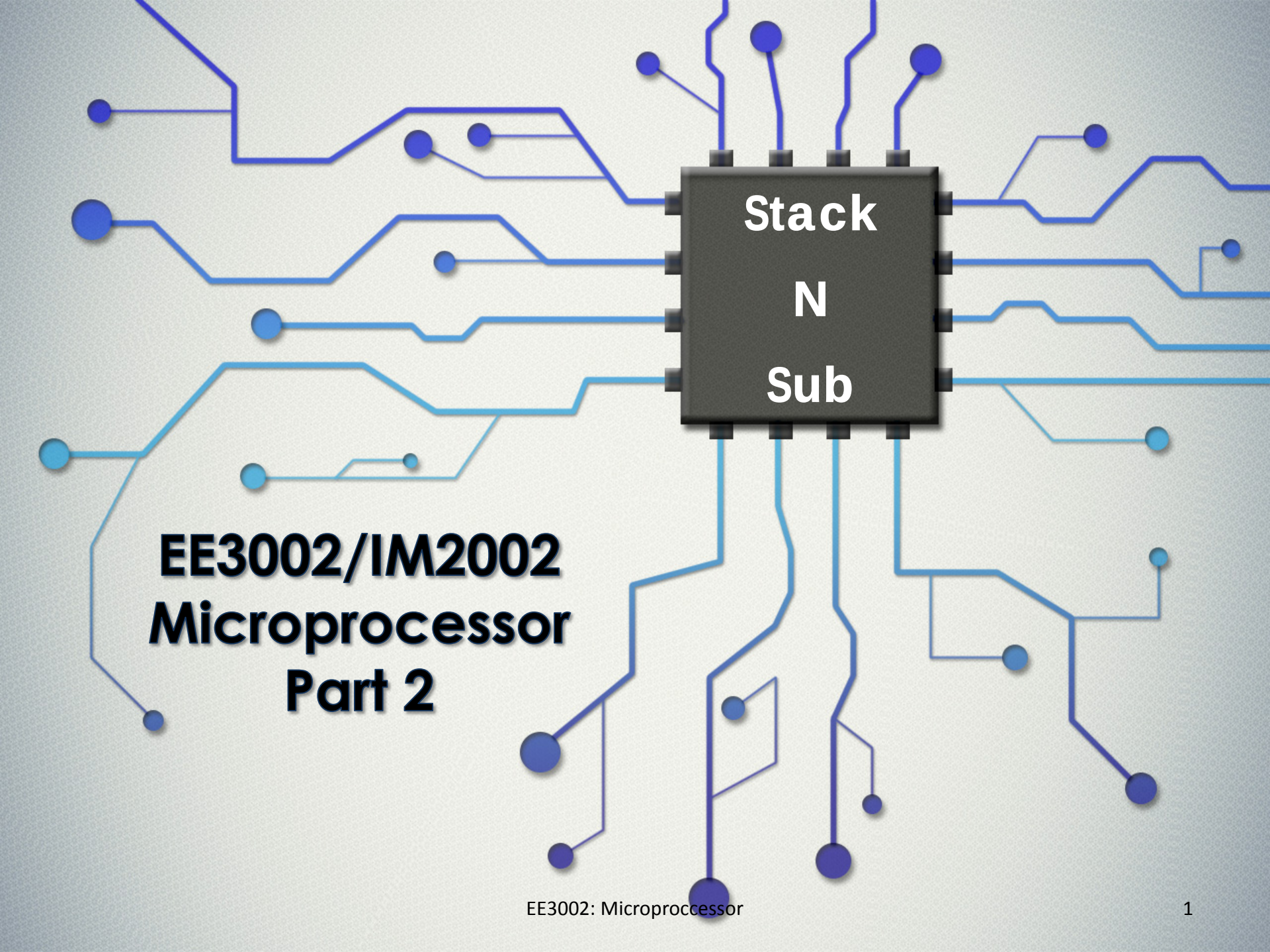
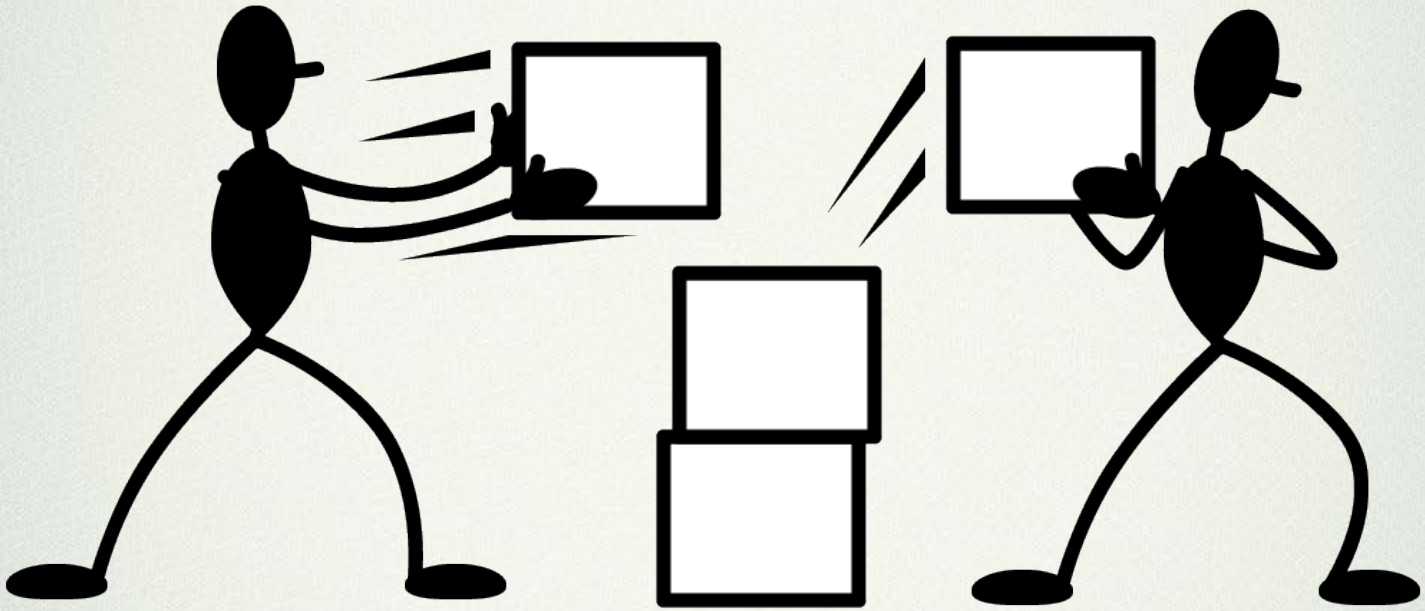


EE3002/IM2002 Microprocessor Part 2

Stack



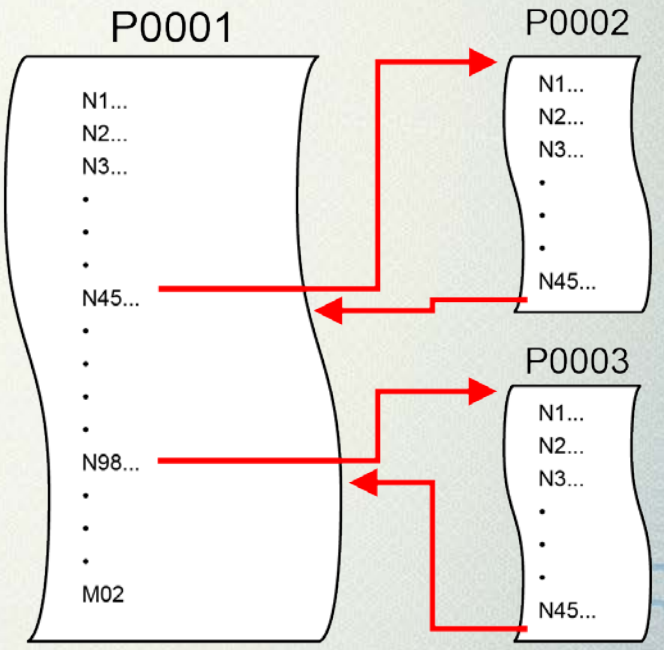
Purpose Of Using Stacks

- Save return addresses in subroutines calls
- Save affected registers to be preserved across subroutine calls
- Memory used to pass parameters to subroutines
- Memory used for allocating space for local variables

Purpose Of Using Subroutines

- Use divide and conquer strategy to break a large program into smaller subroutines
- Example: to program a microcontroller, we break it into several subroutines

```
main BL  GetPosition
     BL  CalcOffset
     BL  DisplayData
      .
      .
      .
```



Requirements Of Writing Subroutines

- Use stack to save and restore affected registers and return address (environment)
- Passing information to and from subroutines
- To build stack in ARM
  - 2 additional ARM instructions are provided
  - STM and LDM instructions

STM/LDM Instructions

- Store and Load Multiple instructions
- Transfer one or more words to/from a list of registers and a pointer (base register)
- Use often in stack and exception handling
- To store content of affected registers before branching to a subroutine or handle an exception → **STR**
- These stored values must be restored to the respective registers upon return from a subroutine or an exception → **LDM**
- Advantages:
  - A single STM or LDM instruction can load or store up to 15 registers instead of using15 instructions
  - Execution time is also shorter

Syntax Of STM Instruction

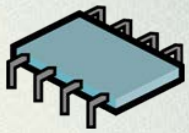
STM {<cond>}<address-mode> <Rn>{!},<reg-list>{^}

- {<cond>} is an optional condition
- <address-mode> specifies addressing mode
- <Rn> is the base register
- <reg-list> is a comma delimited list of registers

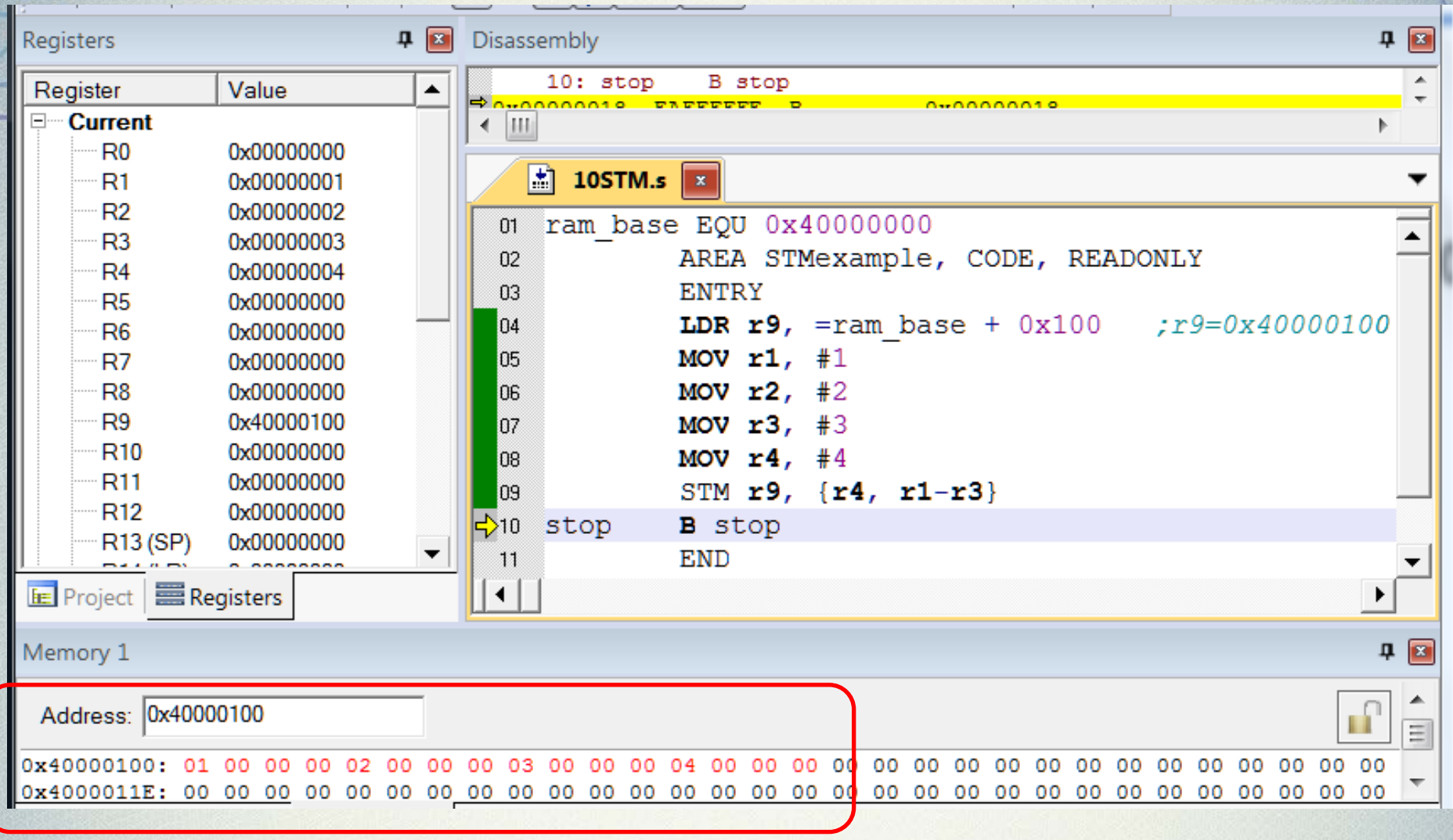
STM Instruction: Example

- **STM r9, {r4, r1-r3}**
  - Store to memory the content of a list of registers r1-r4
  - Register r9 holds the base address, eg., 0x40000100
- **Same effect as having 4 separate STM instructions operating at the same time**
  - STR r1, [r9, #0] ;ea = 0x40000100
  - STR r2, [r9, #4] ;ea = 0x40000104
  - STR r3, [r9, #8] ;ea = 0x40000108
  - STR r4, [r9, #12] ;ea = 0x4000010C
  - ea = effective address
  - r9 remains unchanged after execution of the instruction

STM Instruction: Demo



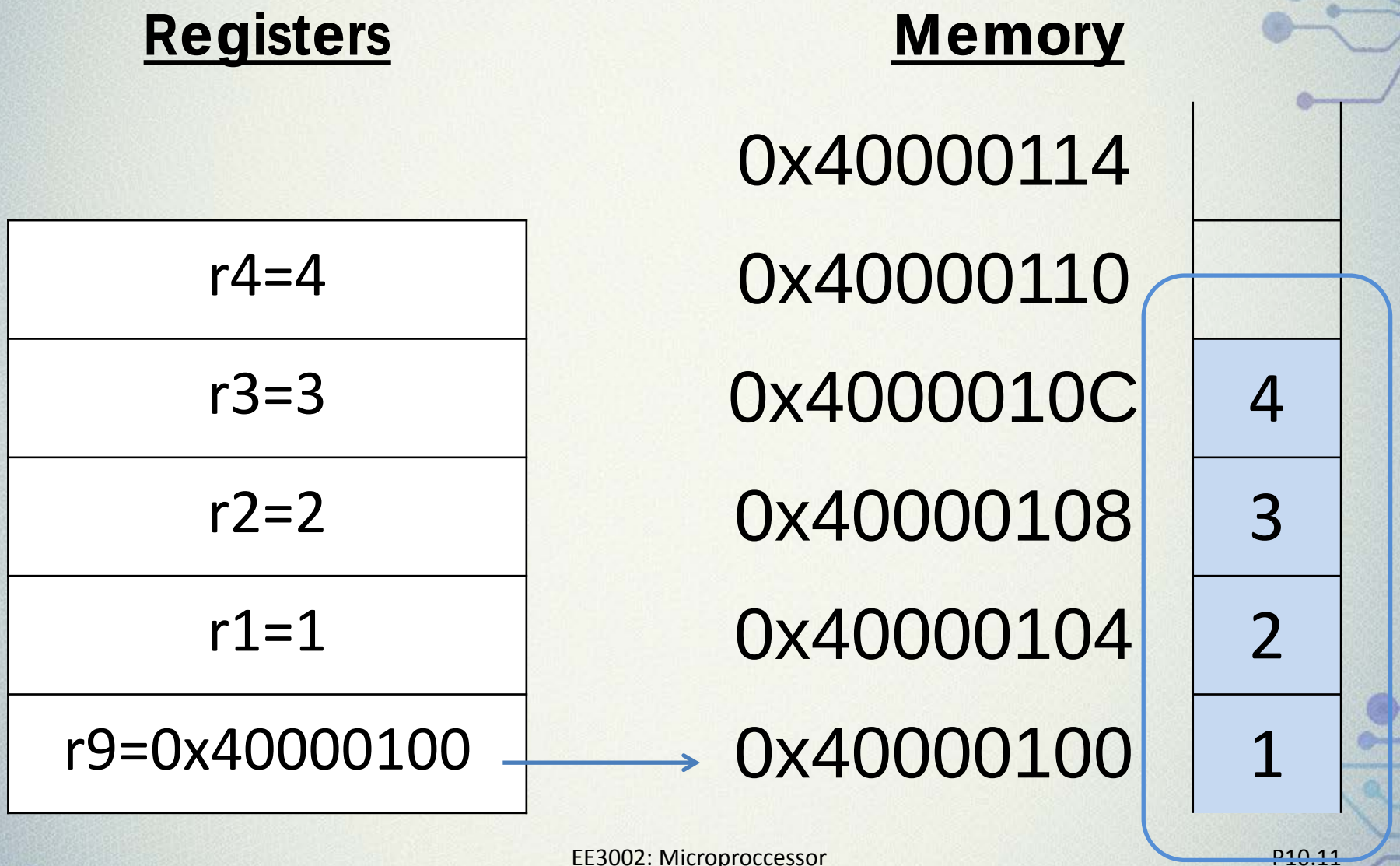
```
ram_base EQU 0x40000000
    AREA STMexample, CODE, READONLY
    ENTRY
    LDR  r9, =ram_base + 0x100   ;r9=0x40000100
    MOV r1, #1
    MOV r2, #2
    MOV r3, #3
    MOV r4, #4
    STM r9, {r4, r1-r3}
stop    B stop
    END
```



Note:

- Content of low register transfers to low memory address.
- Order in register-list is not important.

After Executing STM R9, {R4, R1-r3}



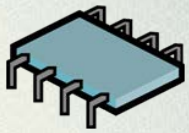
Syntax Of LDM Instruction

- LDM {<cond>}<address-mode> <Rn>{!},<reg-list>{^}
- {<cond>} is an optional condition
- <address-mode> specifies addressing mode
- <Rn> is the base register
- <reg-list> is a comma delimited list of registers

LDM Instruction: Example

- **LDM r9, {r4, r1-r3}**
  - Load from memory to a list of registers r1-r4
  - Register r9 holds the base address, eg., 0x40000100
- **Same effect as having 4 separate LDR instructions operating at the same time**
  - LDR r1, [r9, #0] ;ea = 0x40000100
  - LDR r2, [r9, #4] ;ea = 0x40000104
  - LDR r3, [r9, #8] ;ea = 0x40000108
  - LDR r4, [r9, #12] ;ea = 0x4000010C
  - ea = effective address
  - r9 remains unchanged after execution of the instruction

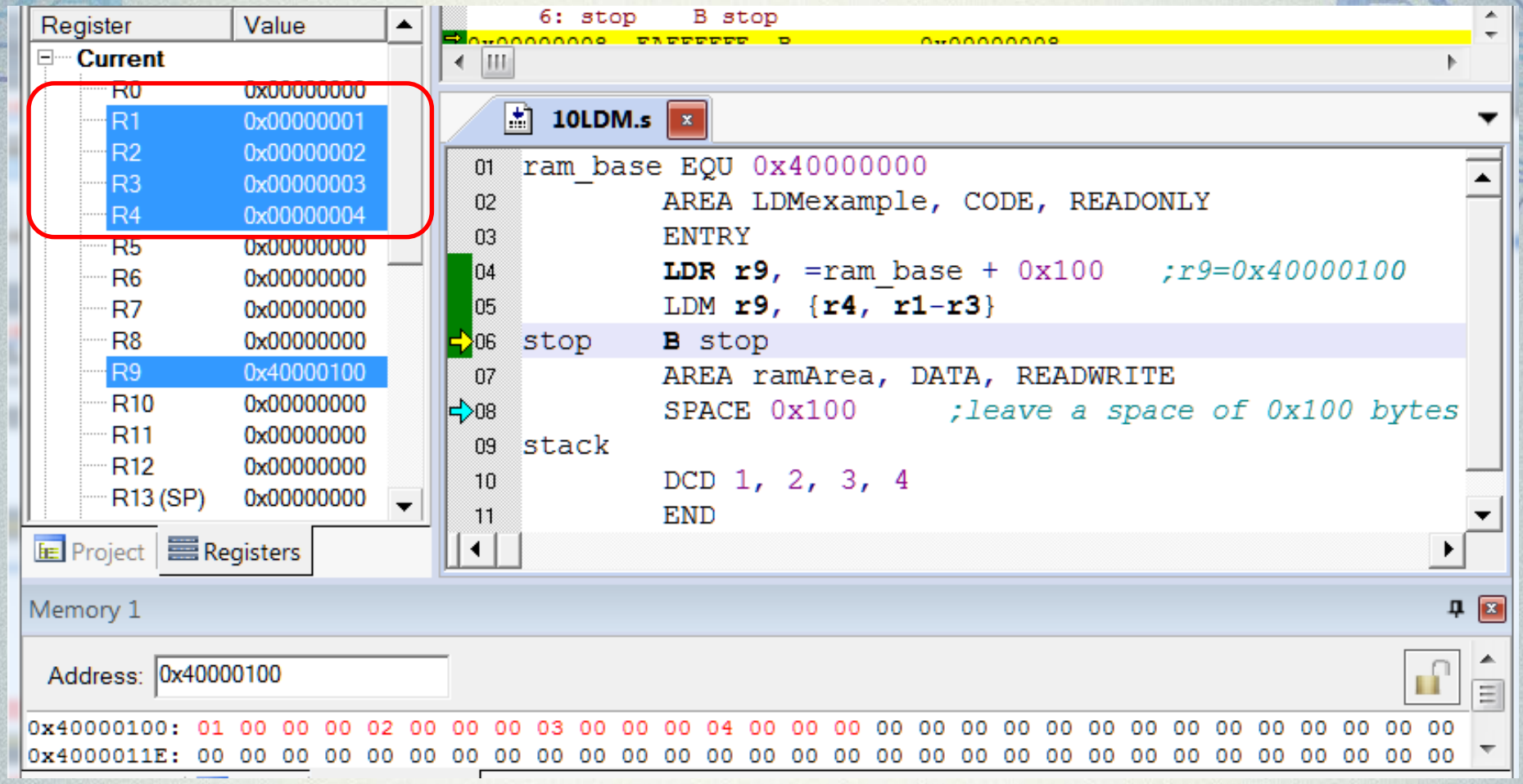
LDM Instruction: Demo



```
ram_base EQU 0x40000000
    AREA LDMexample, CODE, READONLY
    ENTRY
    LDR r9, =ram_base + 0x100    ;r9=0x40000100
    LDM r9, {r4, r1-r3}
stop    B stop
    AREA ramArea, DATA, READWRITE
    SPACE 0x100 ;leave a space of 0x100 bytes
stack
    DCD 1, 2, 3, 4
    END
```

Review On DCD (Define Constant Data)

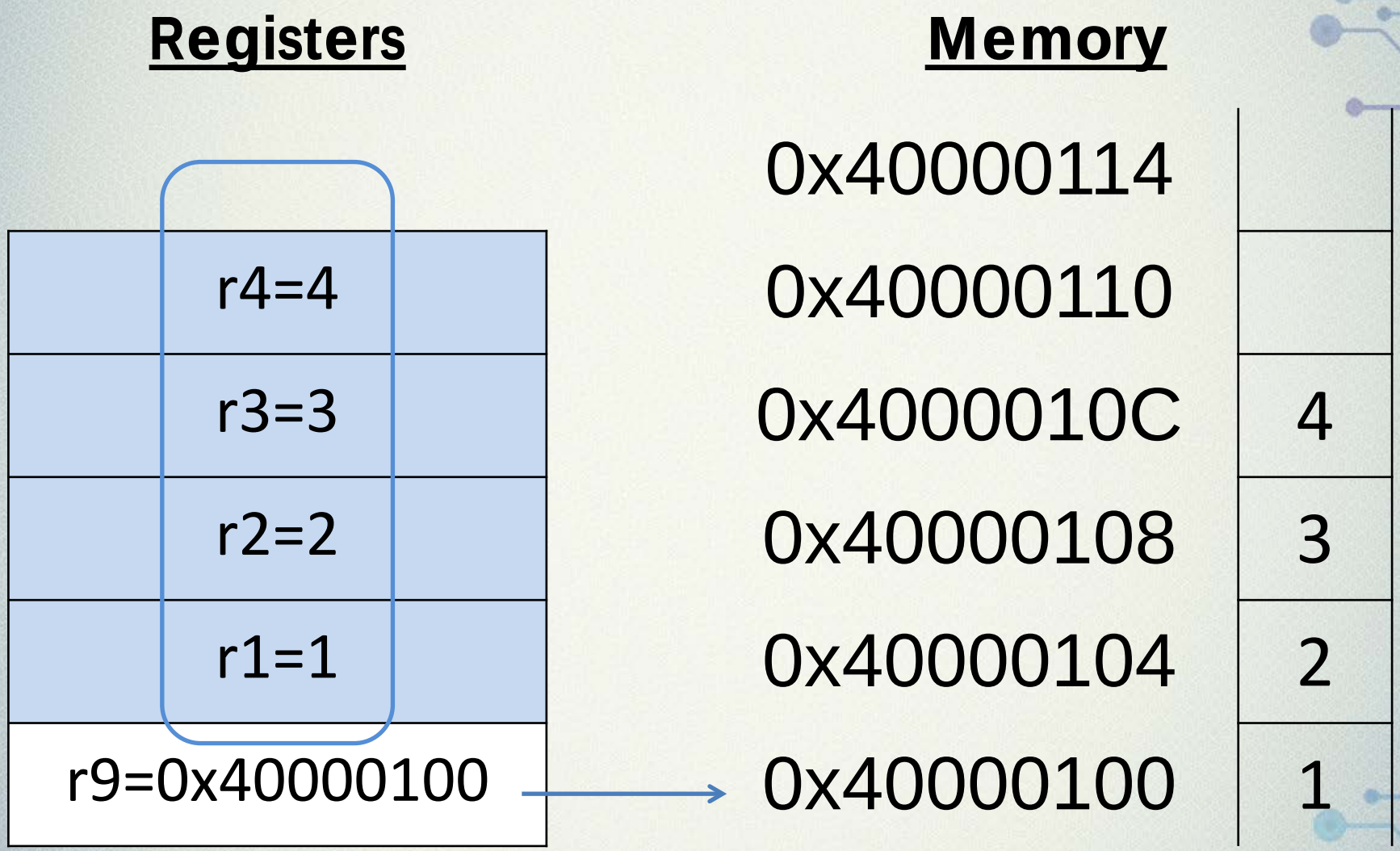
- DCD (Define Constant Data) → Reserve a word (32 bit value)
- DCB (Byte) → Reserve a 8-bit value
- DCW (word) → Reserve a 16-bit value
- DCS (string) → Reserve a string of 255 bytes
- The memory data has to be initialized **externally** but not by these directives
- That means, DCD tells the rest of the program such data exist but not initializing them
- Assembly examples:
- valuex1 DCD 0, 1, 5, 4, 0xFFFFEEEE
- reserved five 32-bit memory locations with 5 initialized values 0, 1, 5, 4, 0xFFFFEEEE.
- In this example, the symbolic name of the first location is valuex1



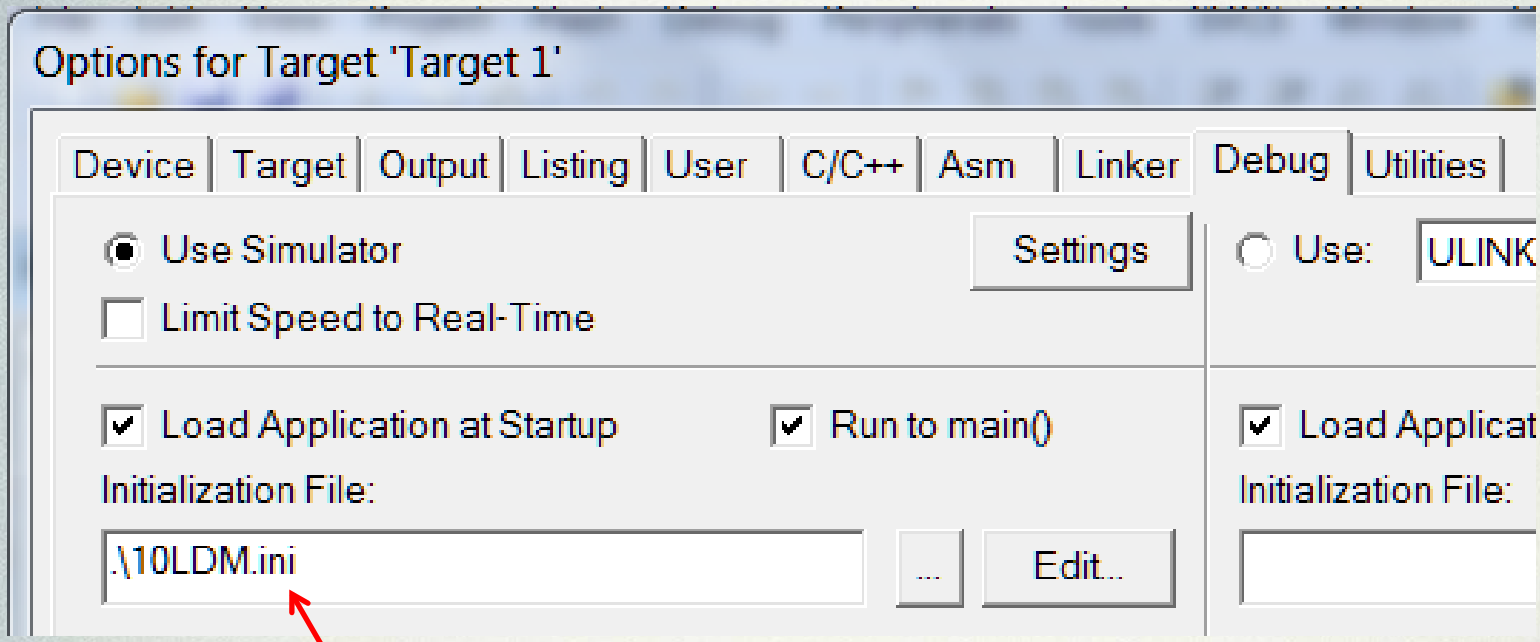
Note:

- Content of low memory address to low register.
- Order in register-list is not important.

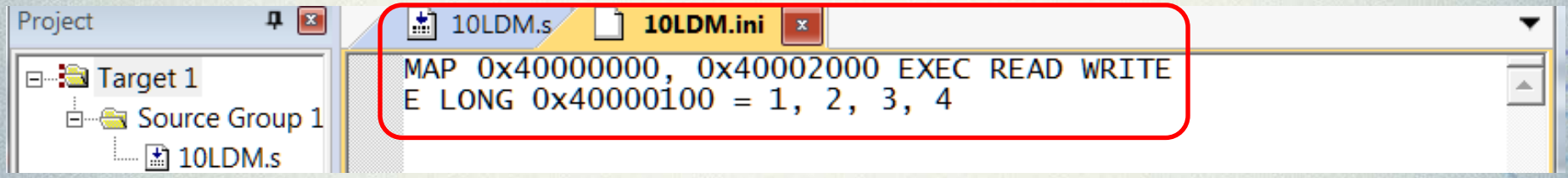
After Executing LDM R9, {R4, R1-r3}



LDM Instruction: Initial File



Attach an initialization file



Addressing Mode On LDM/STM Instructions

- Addressing mode
  - IA – Increment After
  - IB – Increment Before
  - DA – Decrement After
  - DB – Decrement Before
  - Base register remains unchanged after instruction completes, unless being force to update by the ! option

STMDB Instruction: Example

STMDB r9, {r4, r1, r2-r3}

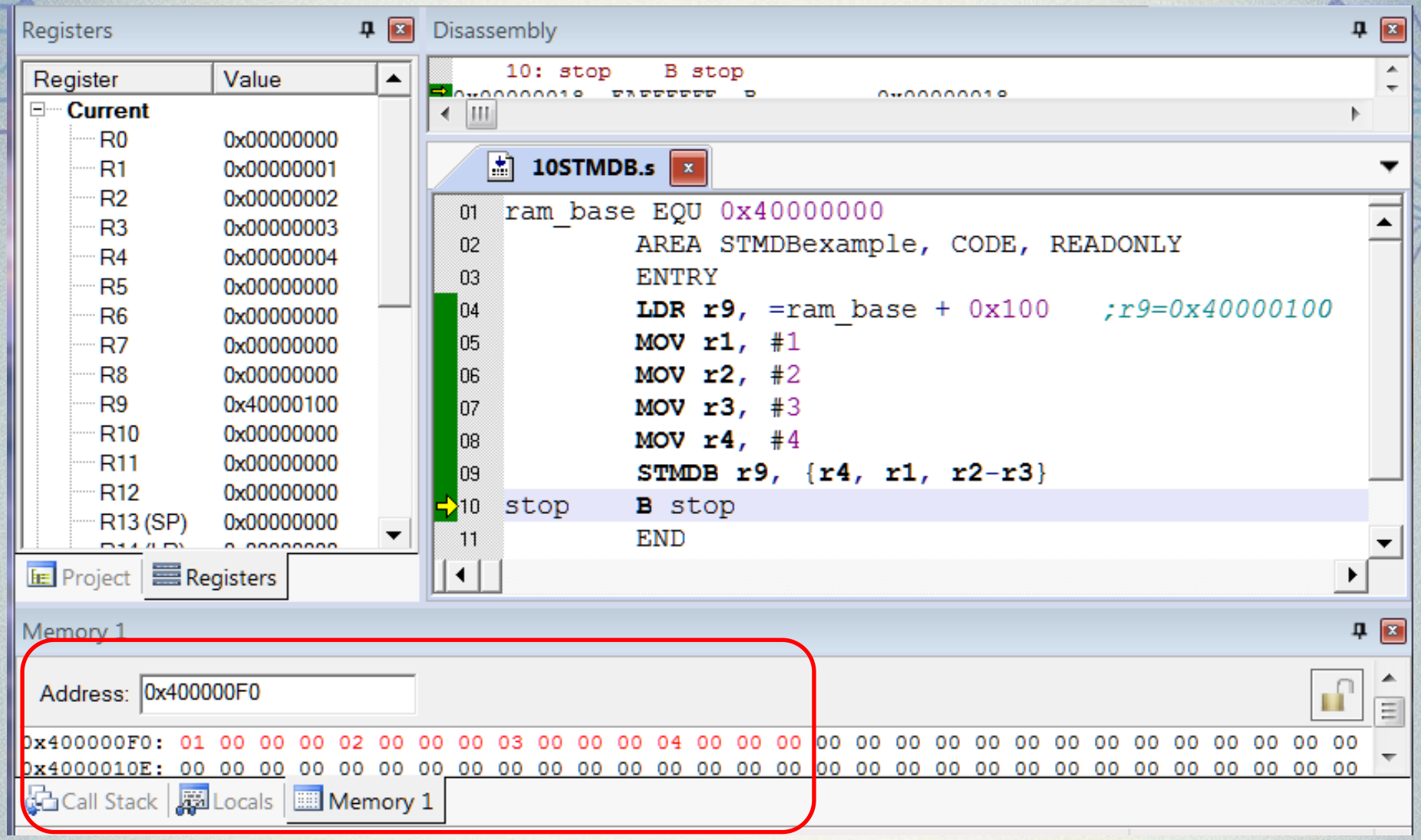
- Store to memory with content of registers r1-r4
- Base register r9 holds the address 0x40000100

Same effect as having 4 separate STR instructions operating at the same time

- STR r1, [r9, #-16] ;ea = 0x400000F0
- STR r2, [r9, #-12] ;ea = 0x400000F4
- STR r3, [r9, #-8] ;ea = 0x400000F8
- STR r4, [r9, #-4] ;ea = 0x400000FC
- The offset is decremented and done before the transfer
- Low register to low memory address
- Order in register list is not important

STMDB Instruction: Demo

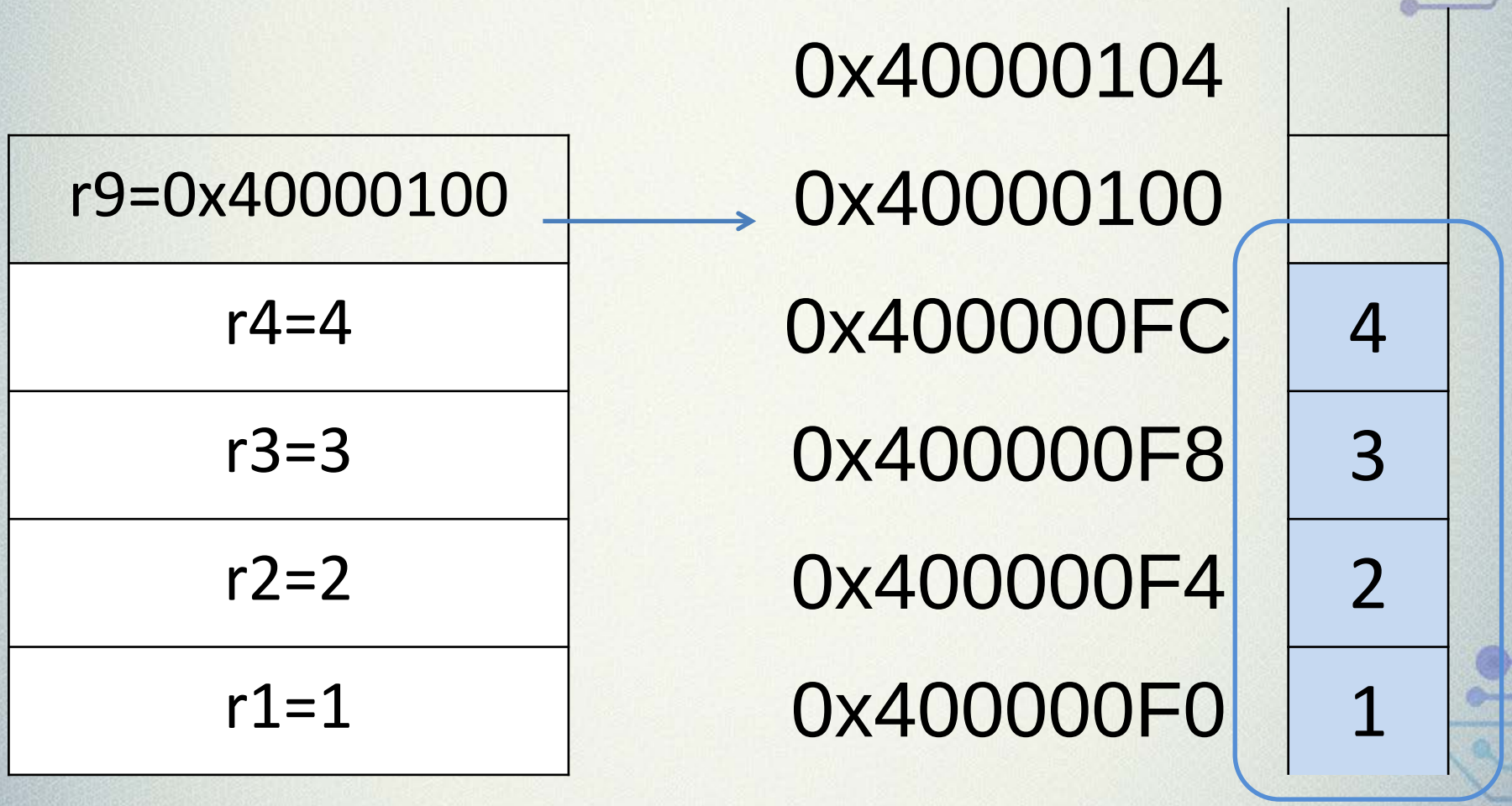
```
ram_base EQU 0x40000000
    AREA STMDBexample, CODE, READONLY
    ENTRY
    LDR r9, =ram_base + 0x100    ;r9=0x40000100
    MOV r1, #1
    MOV r2, #2
    MOV r3, #3
    MOV r4, #4
    STMDB r9, {r4, r1, r2-r3}
stop    B stop
    END
```



Note:

- Content of low register transfers to low memory address.
- Order in register-list is not important.

After Executing STMDB R9, {R4, R1, R2-r3}



LDMDA Instruction: Example

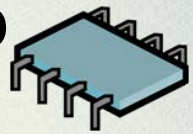
LDMDA r9, {r4, r1, r2, r3}

- Load from memory to a list of registers r1-r4
- Register r9 holds the base address, eg., 0x40000100

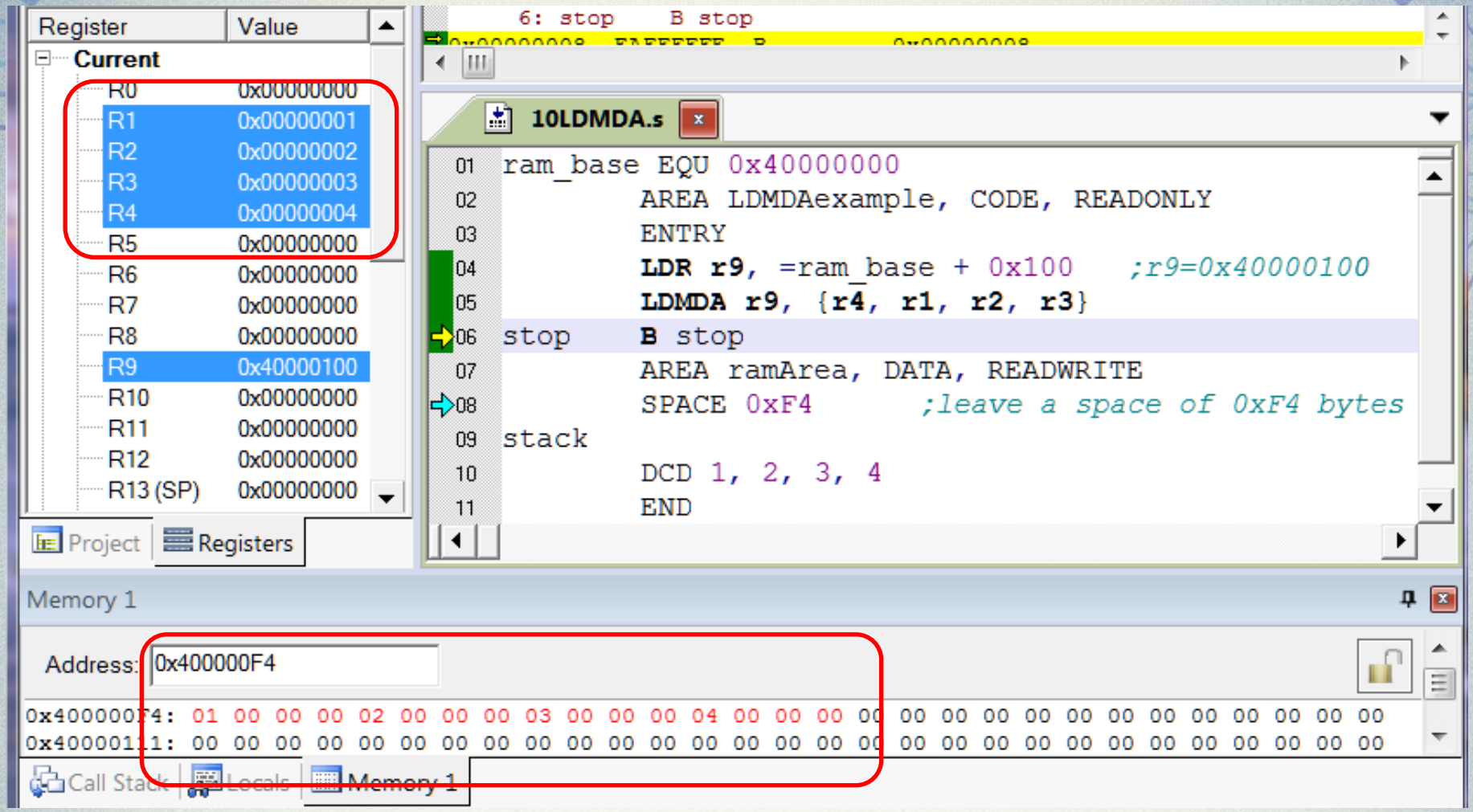
Same effect as having 4 separate LDR instructions operating at the same time

- LDR r1, [r9, #-12] ;ea = 0x400000F4
- LDR r2, [r9, #-8] ;ea = 0x400000F8
- LDR r3, [r9, #-4] ;ea = 0x400000FC
- LDR r4, [r9, #0] ;ea = 0x40000100
- ea = effective address
- r9 remains unchanged after execution of the instruction

LDMDA Instruction: Demo



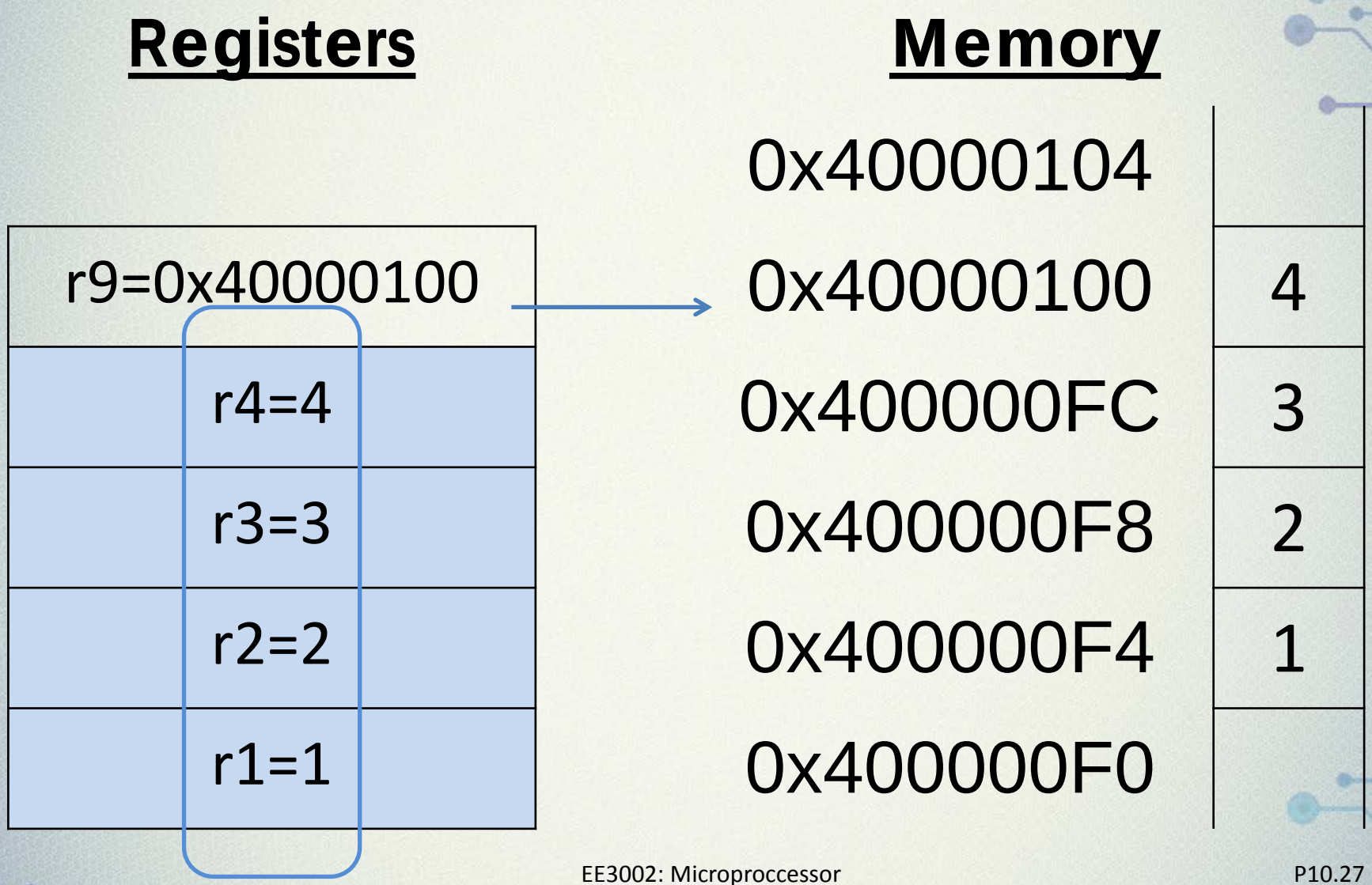
```
ram_base EQU 0x40000000
    AREA LDMDAexample, CODE, READONLY
    ENTRY
    LDR r9, =ram_base + 0x100    ;r9=0x40000100
    LDMDA r9, {r4, r1, r2, r3}
stop    B stop
    AREA ramArea, DATA, READWRITE
    SPACE 0xF4   ;leave a space of 0xF4 bytes
stack
    DCD 1, 2, 3, 4
    END
```



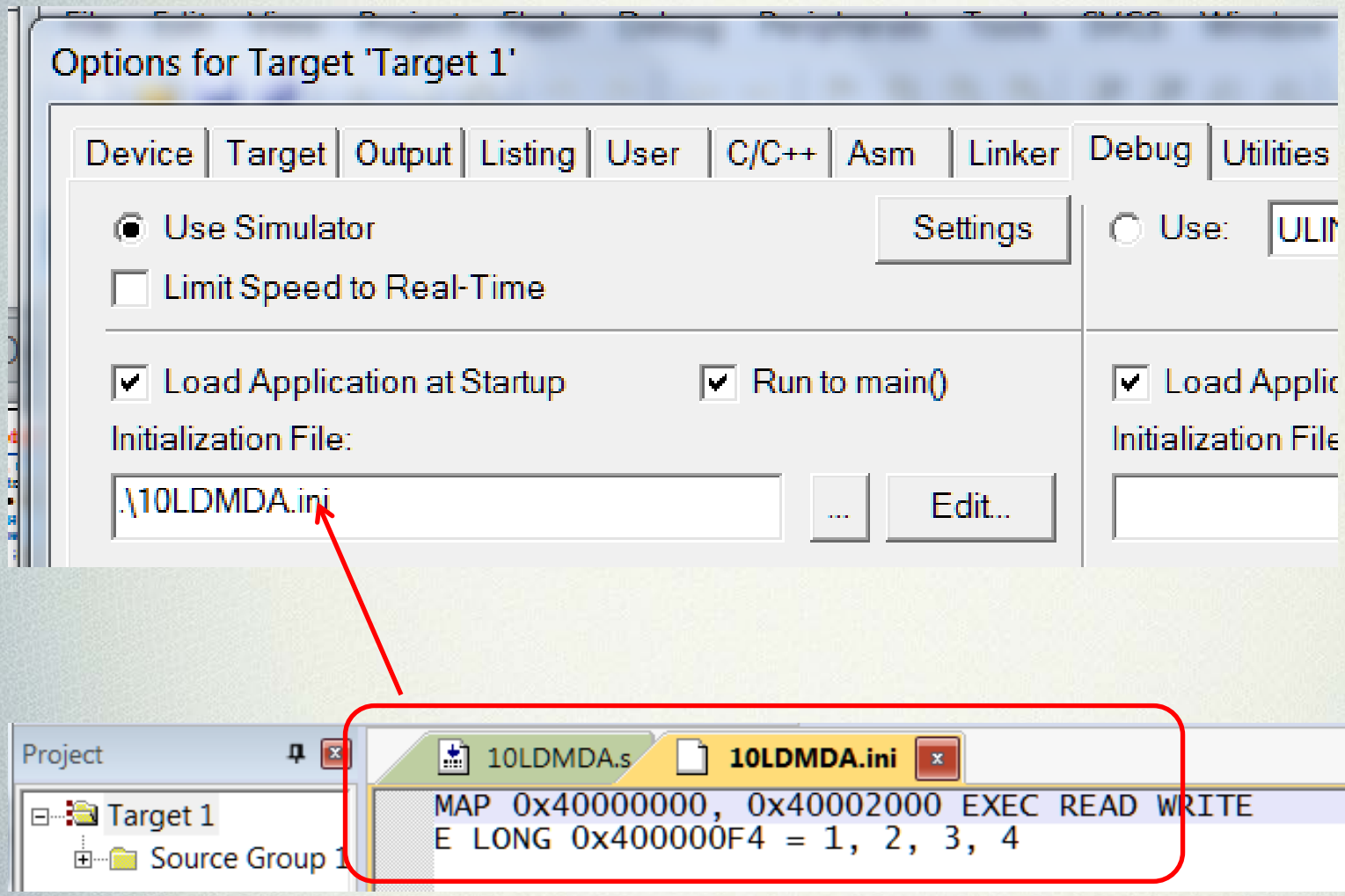
Note:

- Content of low memory address to low register.
- Order in register-list is not important.

After Executing LDMDA R9, {R4, R1, R2, R3}



LDMDA Instruction: Initial File



Other Options On LDM/STM Instructions

- Use the "!" option to force the base register to be updated
- "^" is an optional suffix for handling exception and must not be used in User mode or System mode
  - If *op* is LDM and *reglist* contains the pc (r15), in addition to the normal multiple register transfer, the SPSR is copied into the CPSR. This is for returning from exception handlers. Use this only from exception modes
  - Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers

Stack (1)

- Used widely by most programming systems, in particular to handle a call to a subroutine
- Types
  - Ascending or Descending stack
  - Full or Empty (refers to sp, stack pointer)
- Basic operations
  - PUSH (use store instruction)
  - POP (use load instruction)

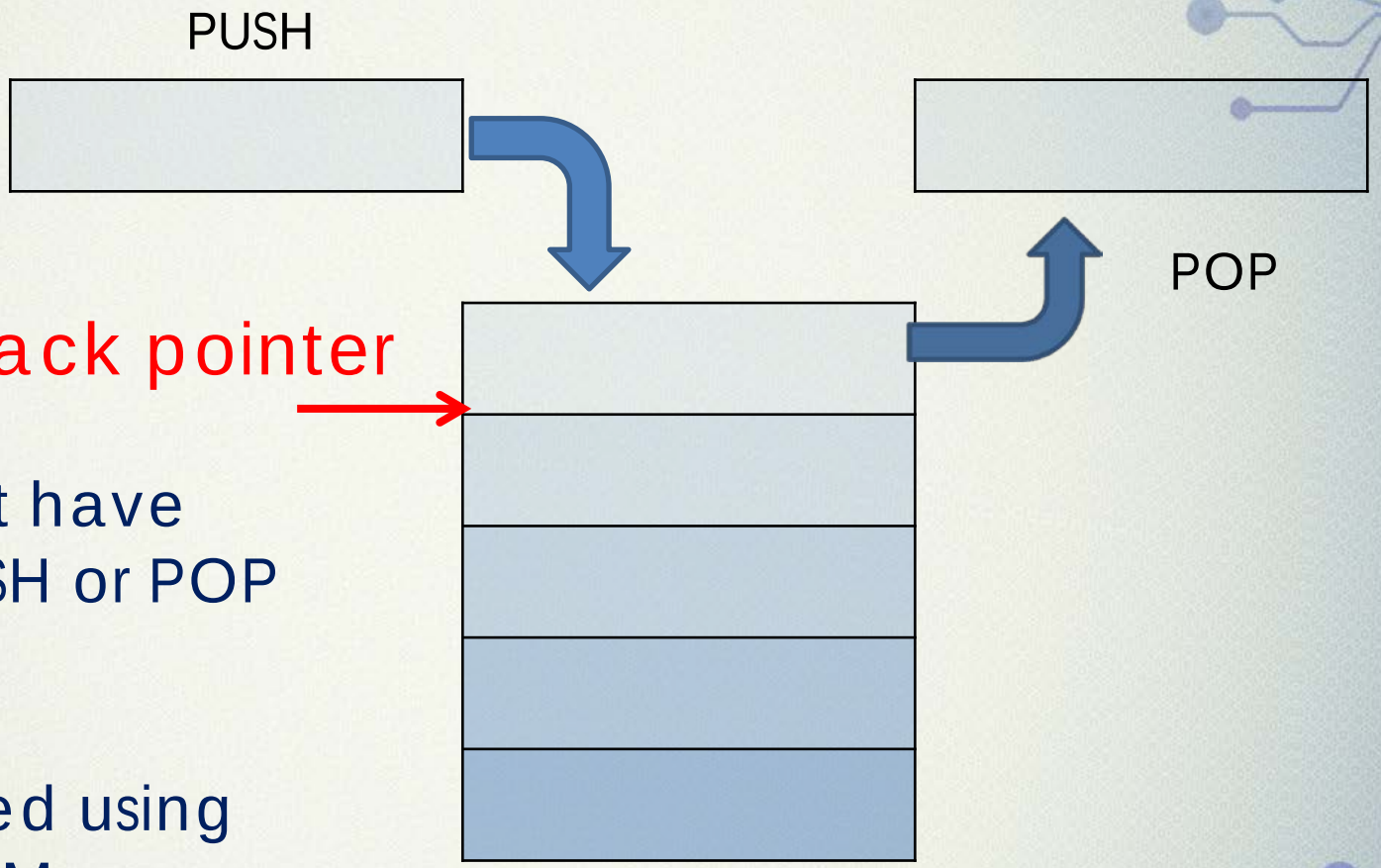
Stack (2)

- Stack is just read/write memory, use to store data temporarily
- Each mode has its own stack and sp
- ARM do not have specific PUSH or POP instructions
- Instead it is implemented using LDM and STM instructions with register r13 as base register
- r13 is the stack pointer (sp), which points to the address of either the next empty word (Empty) or the last pushed word (Full)

Main Purposes Of Using A Stack

- Save return addresses in subroutine calls
- Save registers to be preserved across subroutine calls
- Memory used to pass parameters to subroutines (including C function calls)
- Memory used for allocating space for local variables

The idea of stack – Last In - First Out (LIFO)



Note:

- ARM do not have specific PUSH or POP instructions
- Instead it is implemented using LDM and STM instructions

Type Of Stacks

1. Full Descending (FD)
2. Full Ascending (FA)
3. Empty Descending (ED)
4. Empty Ascending (EA)

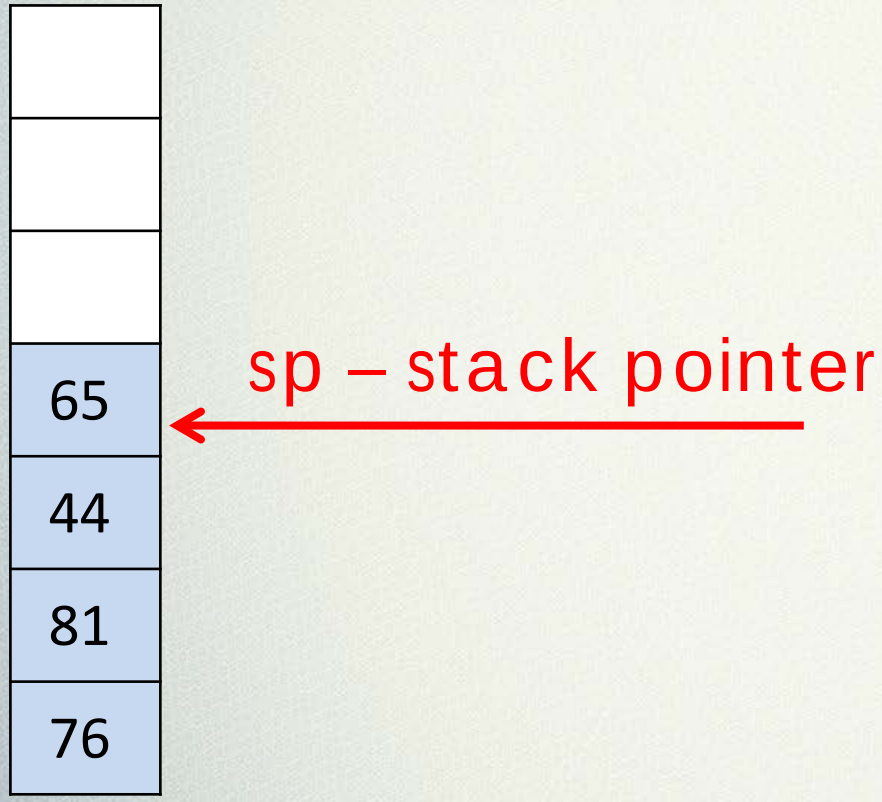
Note:

- Initially, all stacks are empty
- Full or Empty here **does not refer to a full or empty stack**, it refers to where the sp (stack pointer) is pointing
- Full if sp is pointing to the last word pushed into the stack
- Empty if sp is pointing to the next empty word

Type Of Stacks - Full Or Empty?

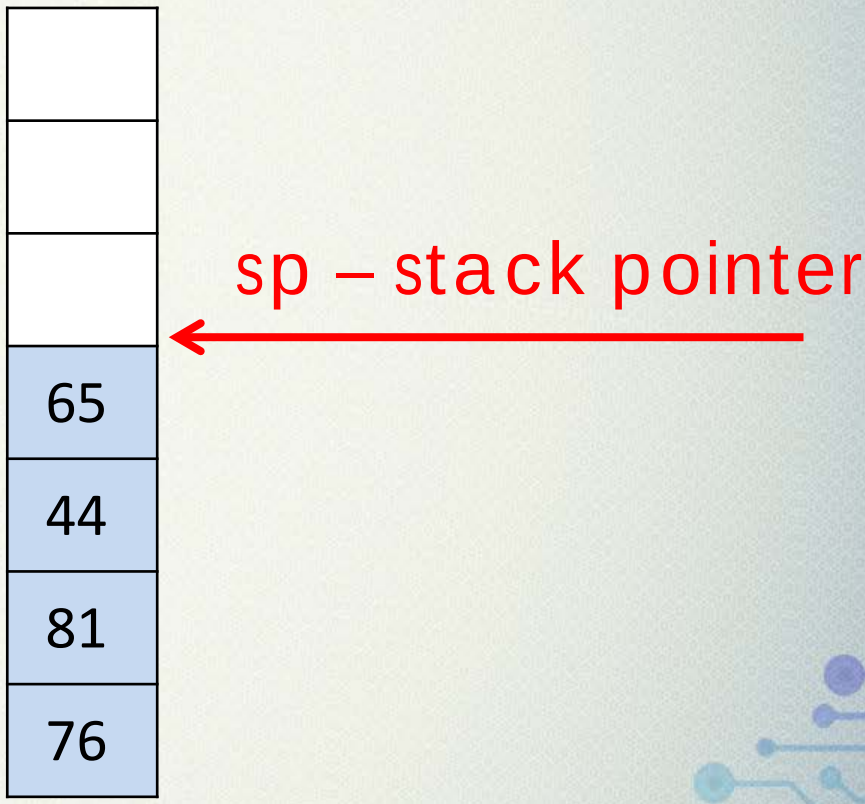
Full

sp points to the last pushed word



Empty

sp points to the next empty word



Stack Pointer – Register R13 Or Sp In ARM

Mode					
User / System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

= banked register

Note:

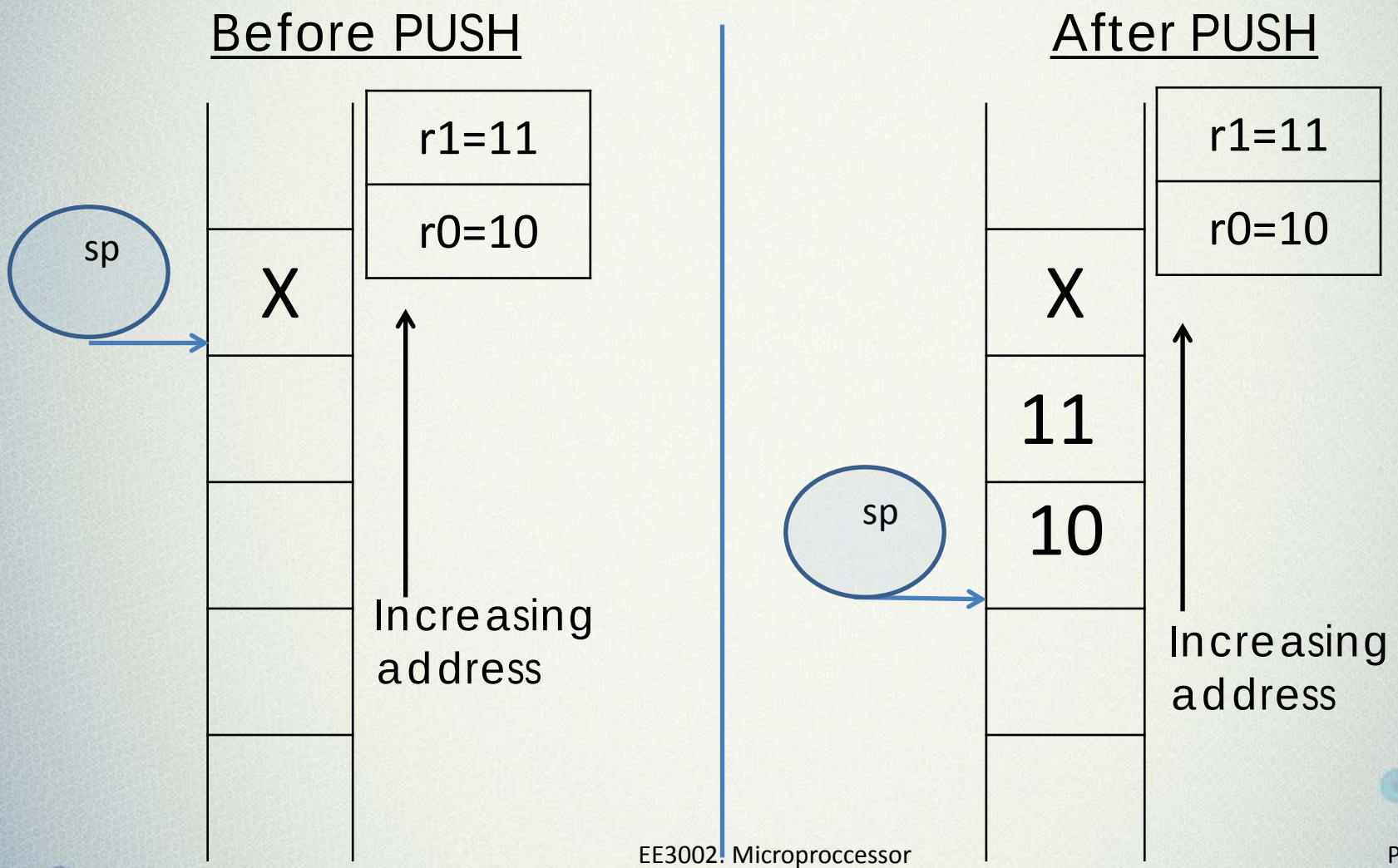
- There are SIX r13 (sp) registers available in ARM, one for each exception mode.
- In other words, we can have SIX separate stacks, one for each exception mode.
- More of these other stacks will be covered later under the exception chapter.

(1) Full Descending (FD) Stack

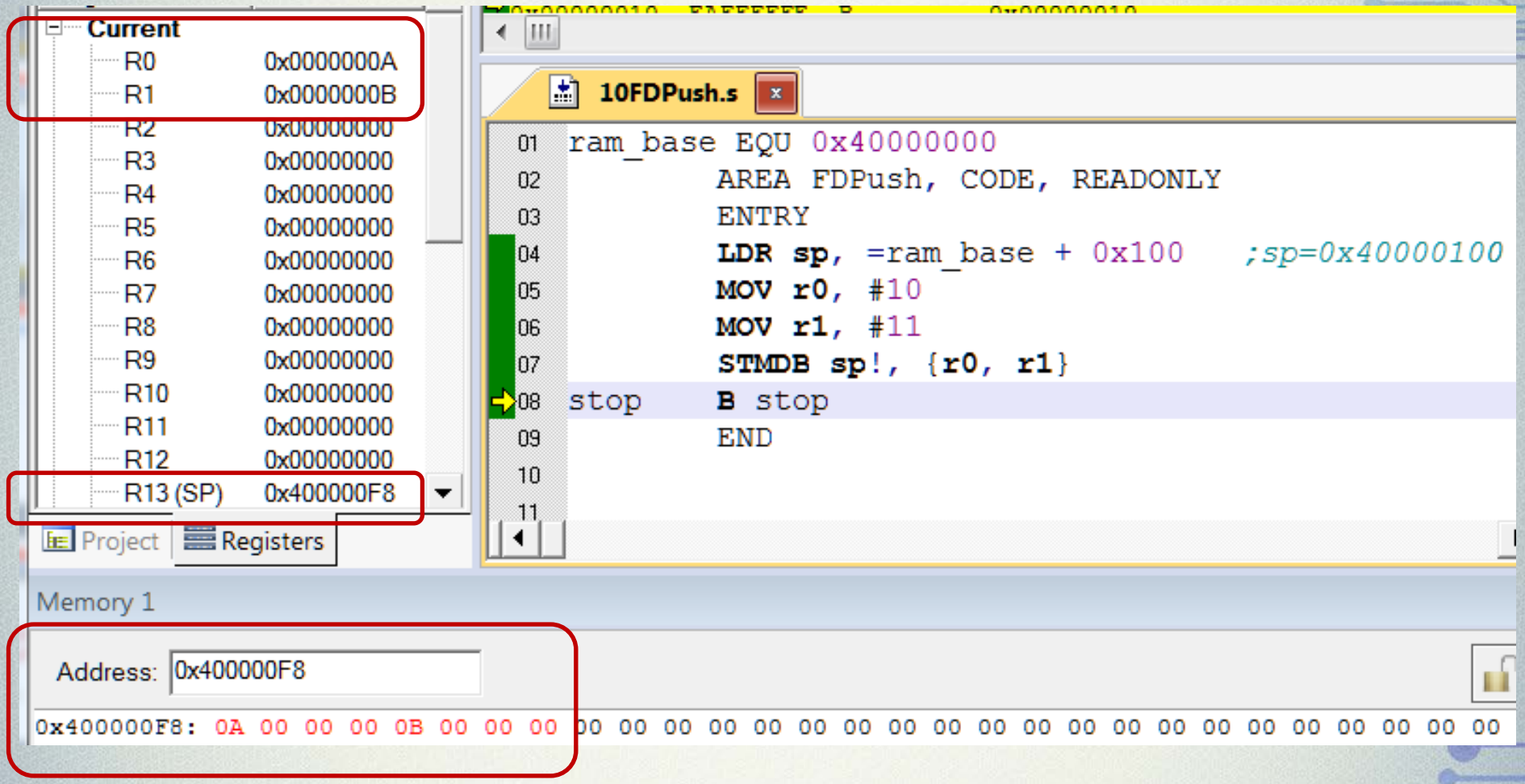
- Descending → the stack grows downward, starting with a high address and progressing to a lower address
- In other words, every times a word is pushed into the stack, it occupies a lower address (-4 bytes), hence descending
- Full → the stack pointer, sp (r13) points to the last pushed word in the stack
- **Use STMDB for PUSH (from registers to stack)**
- **Use LDMIA for POP (from stack to registers)**
- Important: use the ! option on register sp to enforce an update on it

Full Descending Stack → PUSH example

- Use STMDB sp!, {r0,r1} or STMFD sp!, {r0,r1}

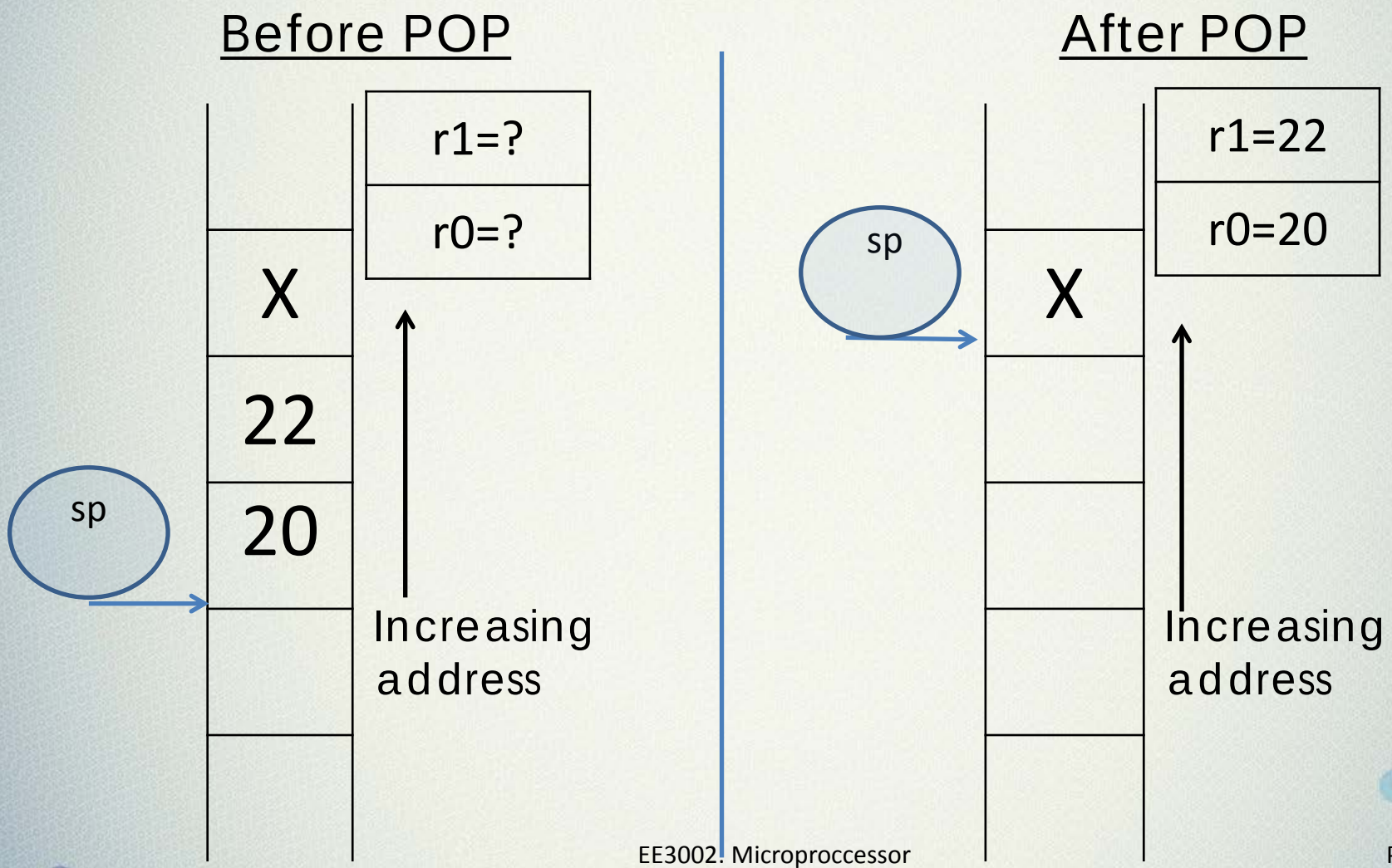


Full Descending Stack → PUSH Demo

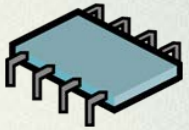


Full Descending Stack → POP Example

- Use LDMIA sp!, {r0,r1} or LDMFD sp!, {r0,r1}



Full Descending Stack → POP Demo



Current

R0	0x00000014
R1	0x00000016
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x40000108

Project | Registers

10FDPop.s

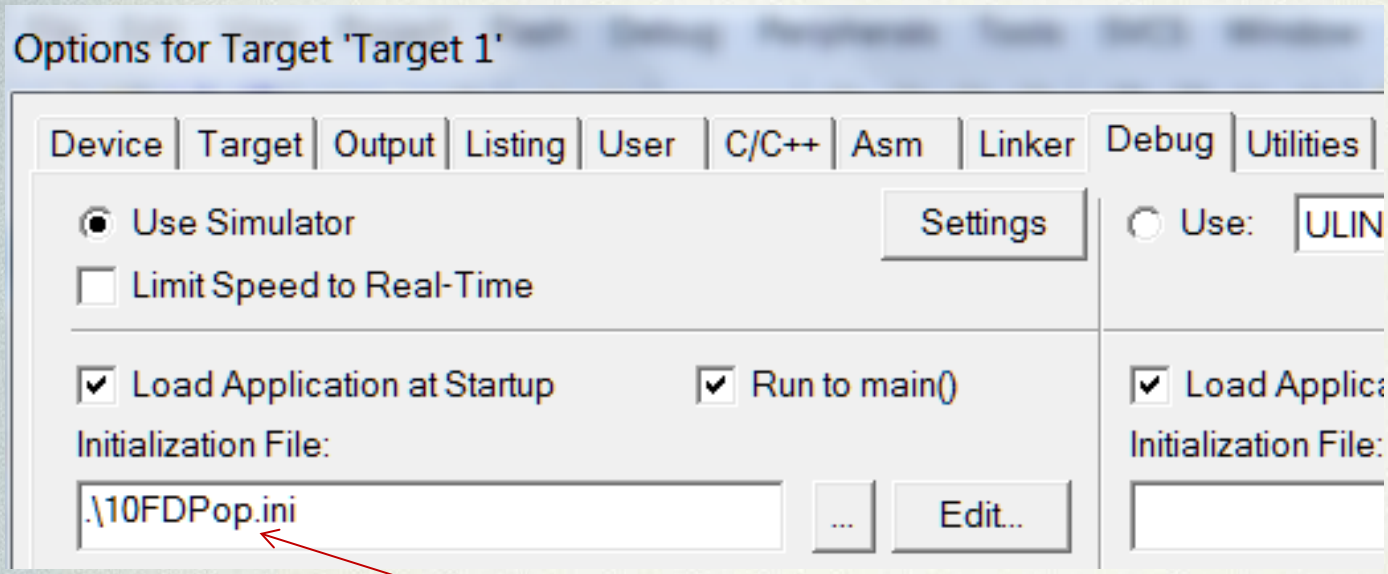
```
01  ram_base EQU 0x40000000
02          AREA FDPop, CODE, READONLY
03          ENTRY
04          LDR sp, =ram_base + 0x100    ;sp=0x40000100
05          LDMIA sp!, {r0, r1}
06  stop    B stop
07          AREA ramArea, DATA, READWRITE
08          SPACE 0x100      ;leave a space of 0x100 byt
09  stack
10          DCD 20,22
11          END
```

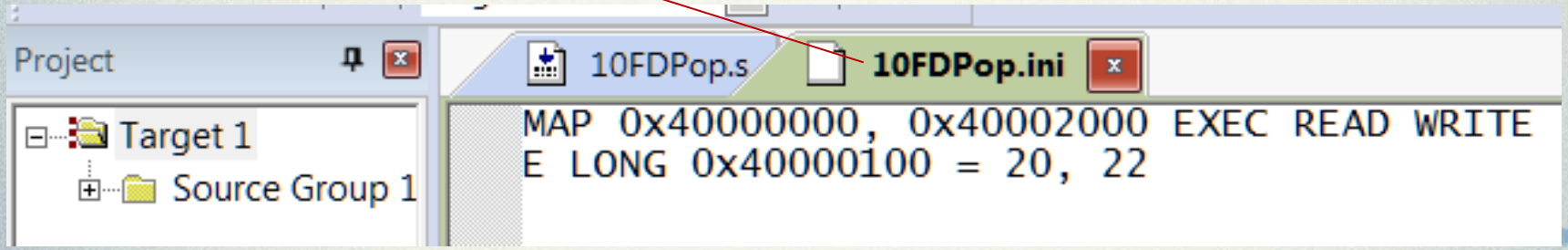
Memory 1

Address: 0x40000100

0x40000100: 14 00 00 00 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Full Descending Stack → POP Demo - Initial File



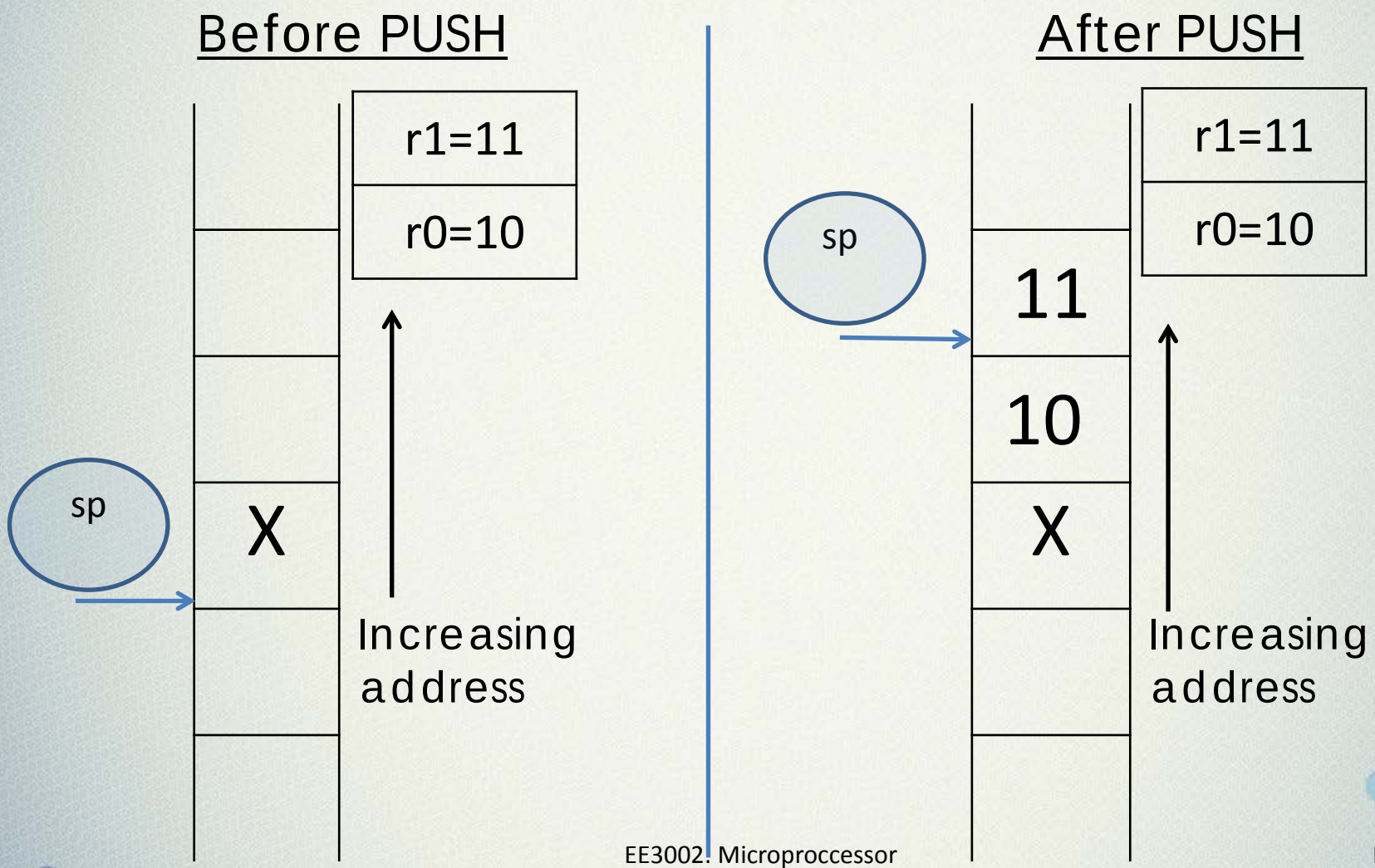


(2) Full Ascending (FA) Stack

- Ascending → the stack grows upward, starting with a low address and progressing to a higher address
- In other words, every times a word is pushed into the stack, it occupies a higher address (+4 bytes), hence ascending
- Full → the stack pointer, sp (r13) points to the last pushed word in the stack
- **Use STMIB for PUSH (from registers to stack)**
- **Use LDMDA for POP (from stack to registers)**
- Important: use the ! option on register sp to enforce an update on it

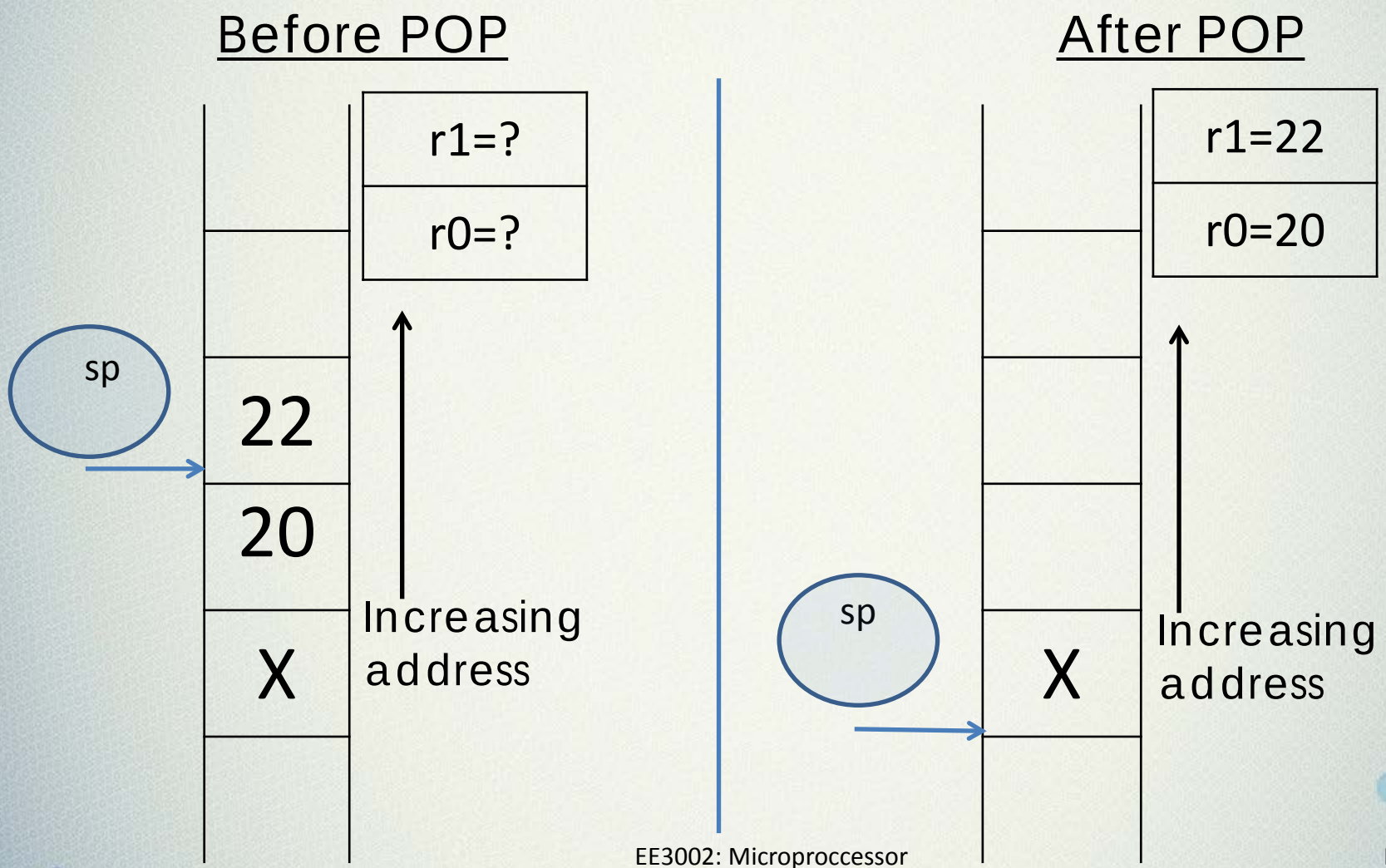
Full Ascending Stack → PUSH Example

- Use STMIB sp!, {r0,r1} or STMFA sp!, {r0,r1}



Full Ascending Stack → POP Example

- Use LDMDA sp!, {r0,r1} or LDMFA sp!, {r0,r1}

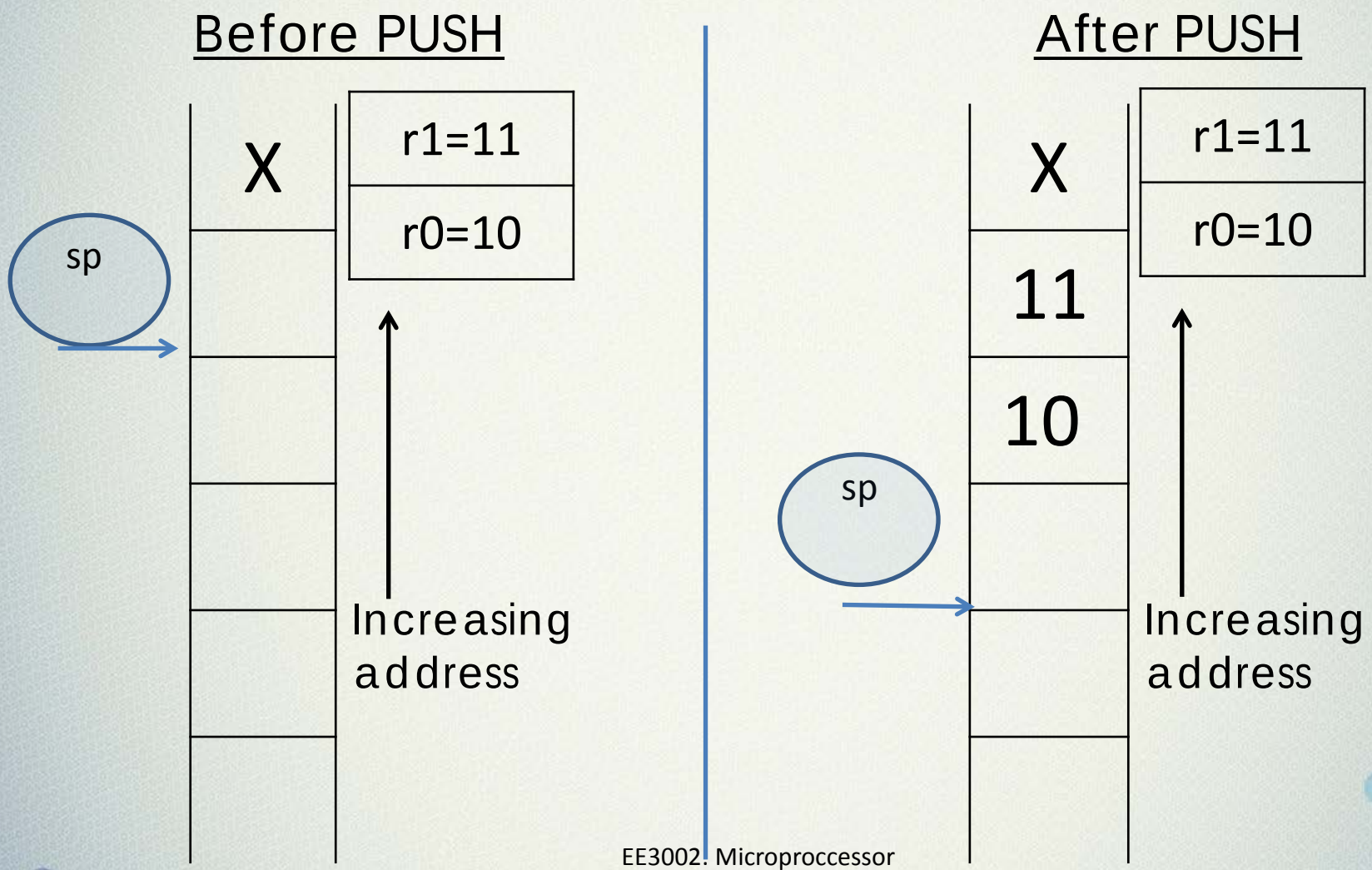


(3) Empty Descending (ED) Stack

- Descending → the stack grows downward, starting with a high address and progressing to a lower address
- In other words, every times a word is pushed into the stack, it occupies a lower address (-4 bytes), hence descending
- Empty → the stack pointer, sp (r13) points to the next empty word in the stack
- **Use STMDA for PUSH (from registers to stack)**
- **Use LDMIB for POP (from stack to registers)**
- Important: use the ! option on register sp to enforce an update on it

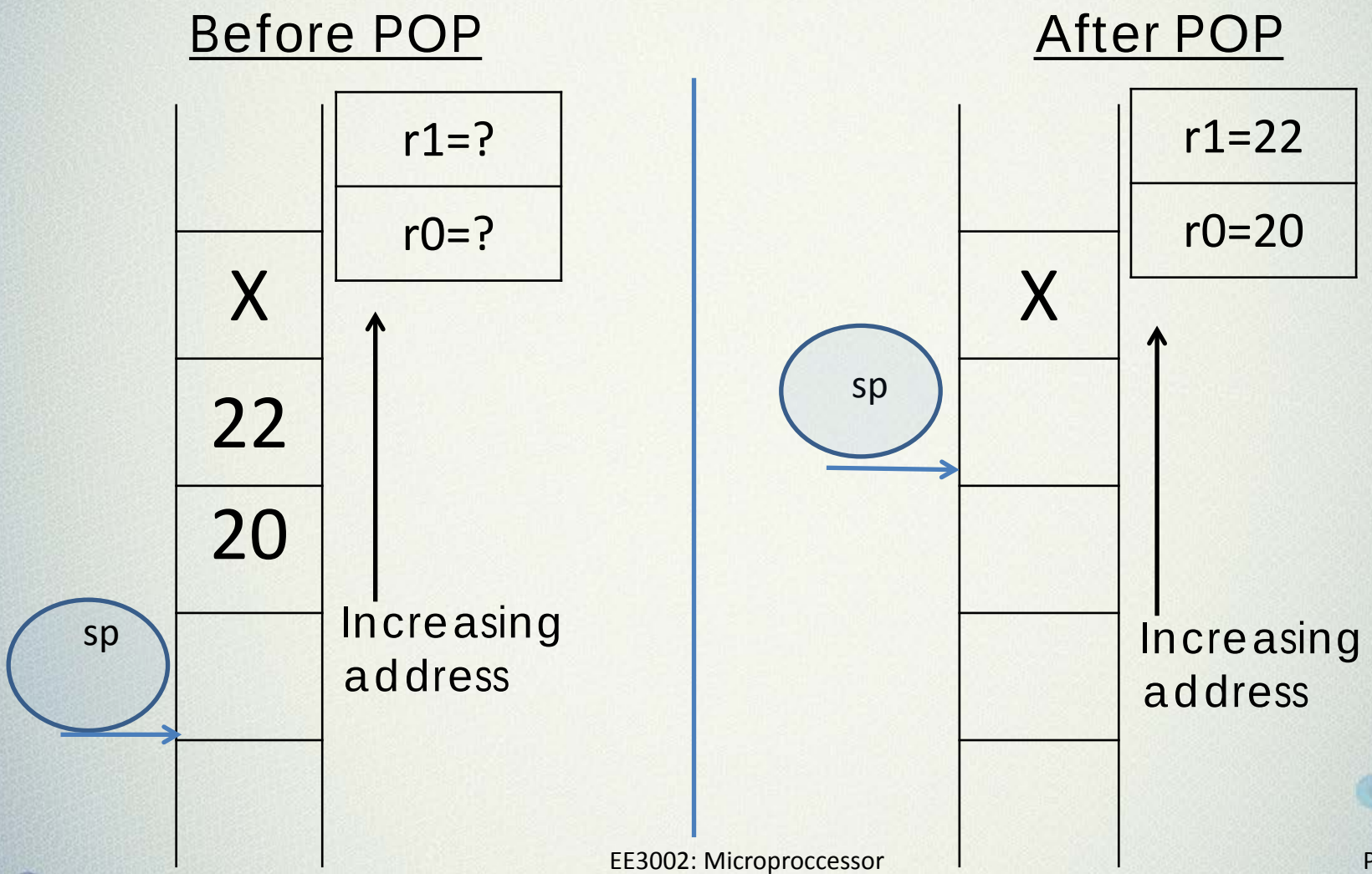
Empty Descending Stack → PUSH Example

- Use STMDA sp!, {r0,r1} or STMED sp!, {r0,r1}



Empty Descending Stack → POP example

- Use LDMIB sp!, {r0,r1} or LDMED sp!, {r0,r1}

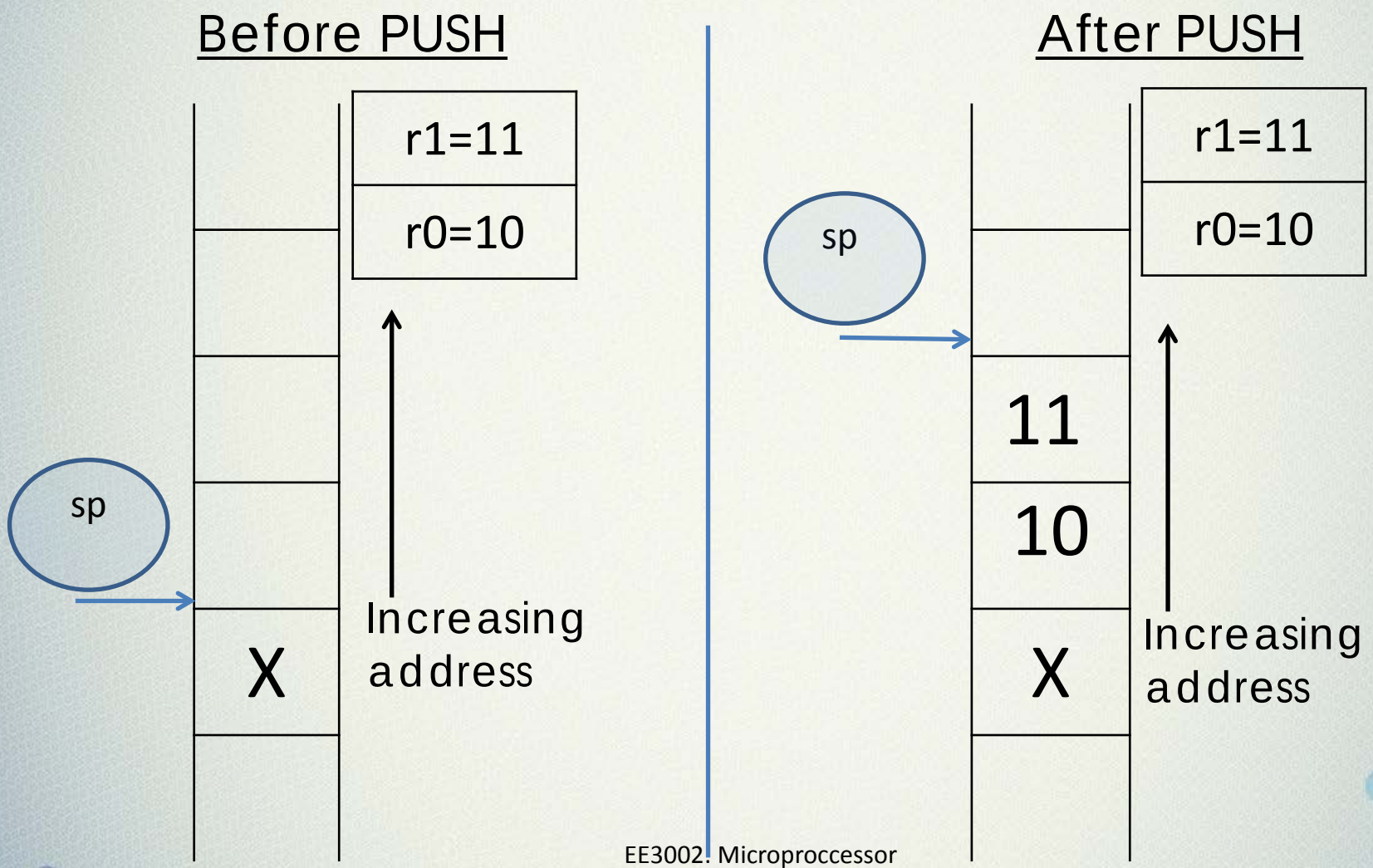


(4) Empty Ascending (EA) Stack

- Ascending → the stack grows upward, starting with a low address and progressing to a higher address
- In other words, every times a word is pushed into the stack, it occupies a higher address (+4 bytes), hence ascending
- Empty → the stack pointer, sp (r13) points to the next empty word in the stack
- **Use STMIA for PUSH (from registers to stack)**
- **Use LDMDB for POP (from stack to registers)**
- Important: use the ! option on register sp to enforce an update on it

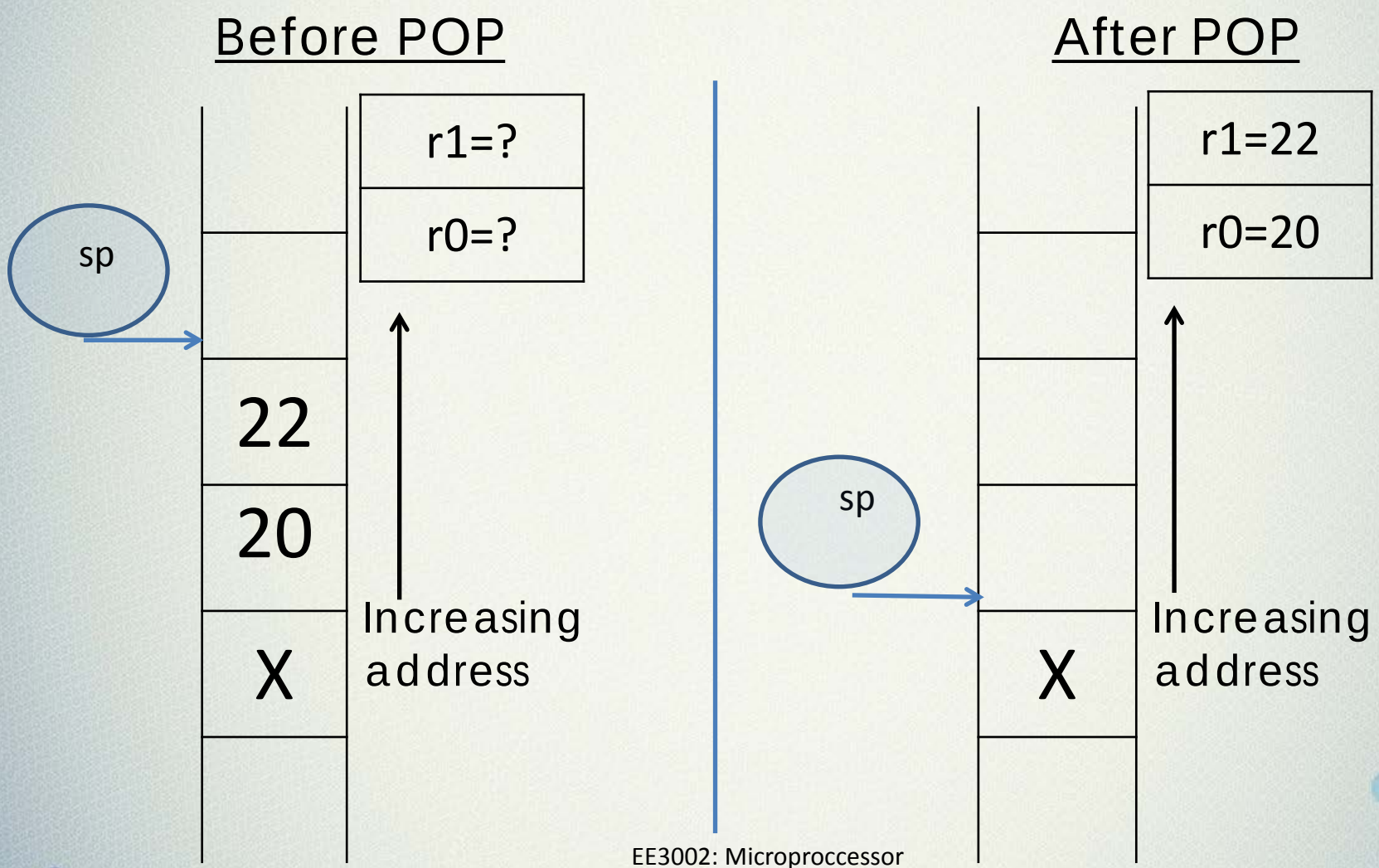
Empty Ascending Stack → PUSH example

- Use STMIA sp!, {r0,r1} or STMEA sp!, {r0,r1}



Empty Ascending Stack → POP Example

- Use LDMDB sp!, {r0,r1} or LDMEA sp!, {r0,r1}



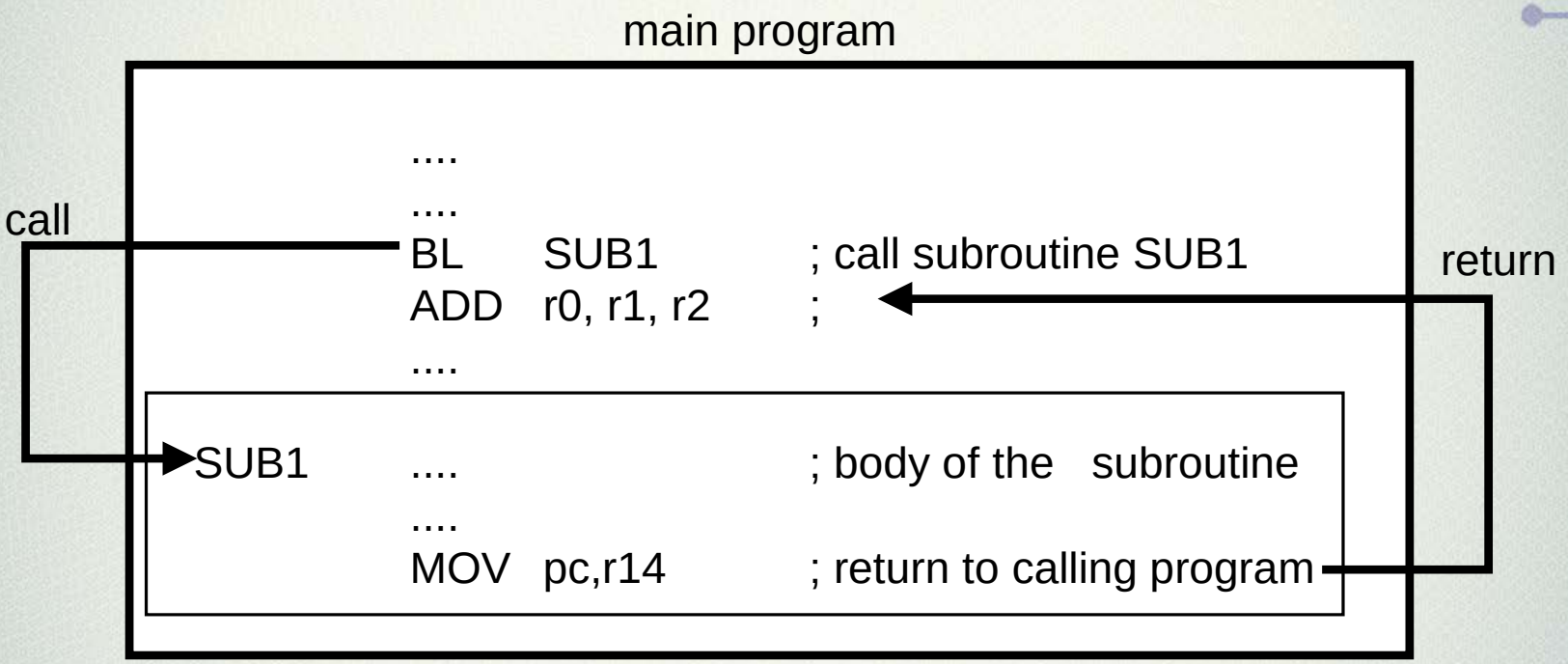
Stack-Oriented Suffixes

Stack Type	PUSH	POP
Full Descending	**STMFD**(STMDB)	**LDMFD**(LDMIA)
Full Ascending	**STMFA**(STMIB)	**LDMFA**(LDMDA)
Empty Descending	**STMED**(STMDA)	**LDMED**(LDMIB)
Empty Ascending	**STMEA**(STMIA)	**LDMEA**(LDMDB)

Subroutine (1)

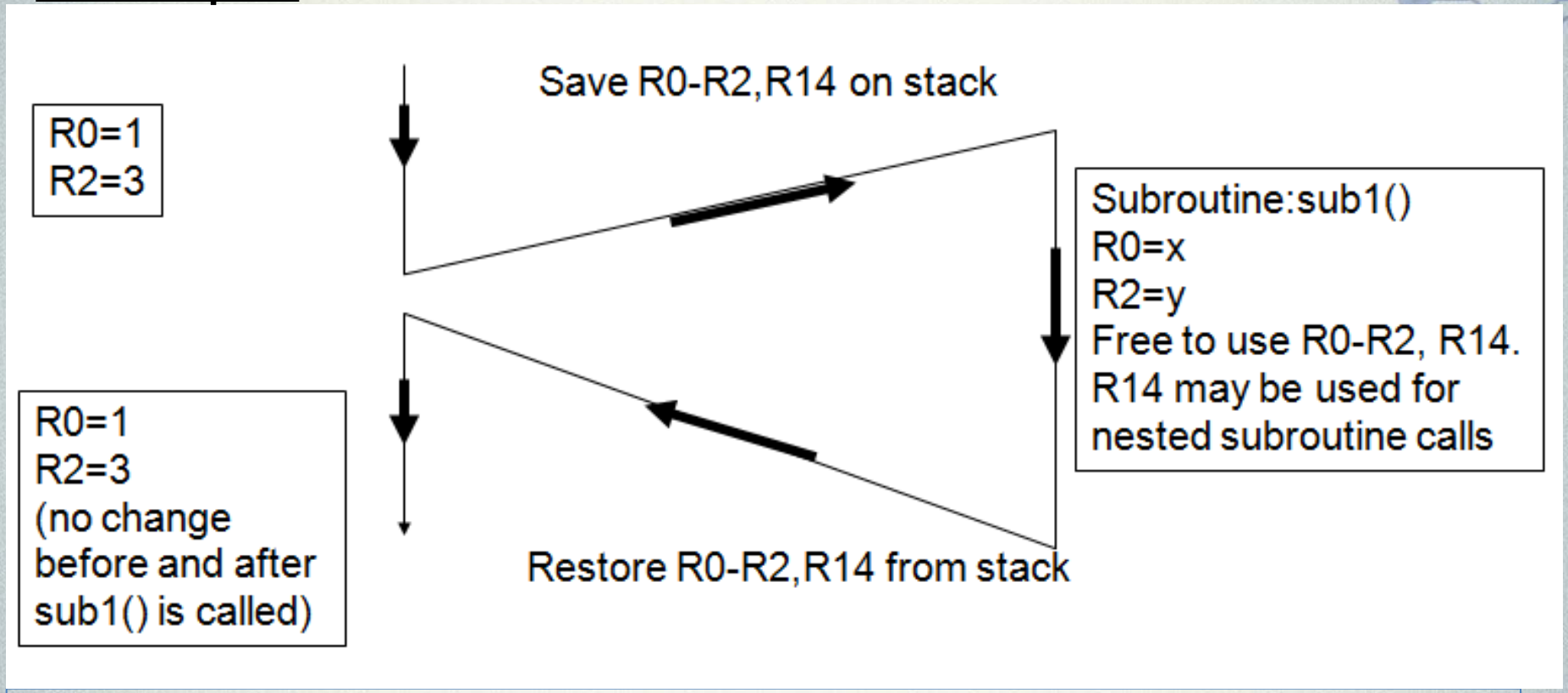
- Good software engineering strategy → divide and conquer
- Hence, large programs consist of many smaller subroutines
- Call with BL (branch and link) instruction
  - Transfers return address (address after the BL instruction) to link register lr (r14)
  - lr = pc – 4, why?
  - Transfers branch target (starting address of subroutine) to program counter pc (r15)

Subroutine (2)



Subroutine (3)

Example



Use stack to save/restore registers so as to preserve their contents before the call

Exercise On Stack Size

- For this stack size, how many nested subroutines (each sub uses 4x32bits=16 bytes) calls are allowed?
- Total stack size
- =0x488+1
  - $=4x16^2+8x16^1+8x16^0+1$
  - =1024+128+8+1
  - =1161 bytes.
- Total_size / each_call_size =1161/16 =72.6
- So 72 nested calls are allowed
- If you do 73 nested calls, it will result in stack overflow

Example:
ARM7 memory map

Addresses	AREA type
0x4000 FFFF- 0x4000 0489	Data
0x4000 0488- 0x4000 0000	**Stack**
0x0000 7FFF- 0x0000 0000	Code

Review – Branch instructions

- These instructions are used to:
  - branch backward to form loops → for, while loop
  - branch forward in conditional structure → jump
  - branch to subroutine with link option → subroutine call
  - change the processor from ARM state to Thumb state → to be covered later

Branch With Condition

- Signed numbers: B {EQ, NE, GE, LT, GT, LE}
- Unsigned numbers: B {EQ, NE, HS, LO, HI, LS}

Review Of Condition Codes – Signed Numbers

Field Mnemonic	Meaning
EQ	Equal ==
NE	Not equal !=
GE	Signed ≥
LT	Signed <
GT	Signed >
LE	Signed ≤

Review Of Condition Codes – Unsigned Numbers

Field Mnemonic	Meaning
EQ	Equal ==
NE	Not equal !=
CS/HS	Unsigned ≥
CC/LO	Unsigned <
HI	Unsigned >
LS	Unsigned ≤

Branch Link (BL) Instruction

To call a subroutine in ARM, use a Branch Link instruction.

The syntax is:

- BL *label* where *label* is usually the label (target address) on the first instruction of the subroutine
- Actions by the BL instruction
  - Copy the return address (address of the instruction after the BL instruction in the calling program) to link register lr (=pc-4)
  - Copy to register pc the target address (label) of the subroutine

Registers Lr (R14) And Pc (R15)

Mode					
User / System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

= banked register

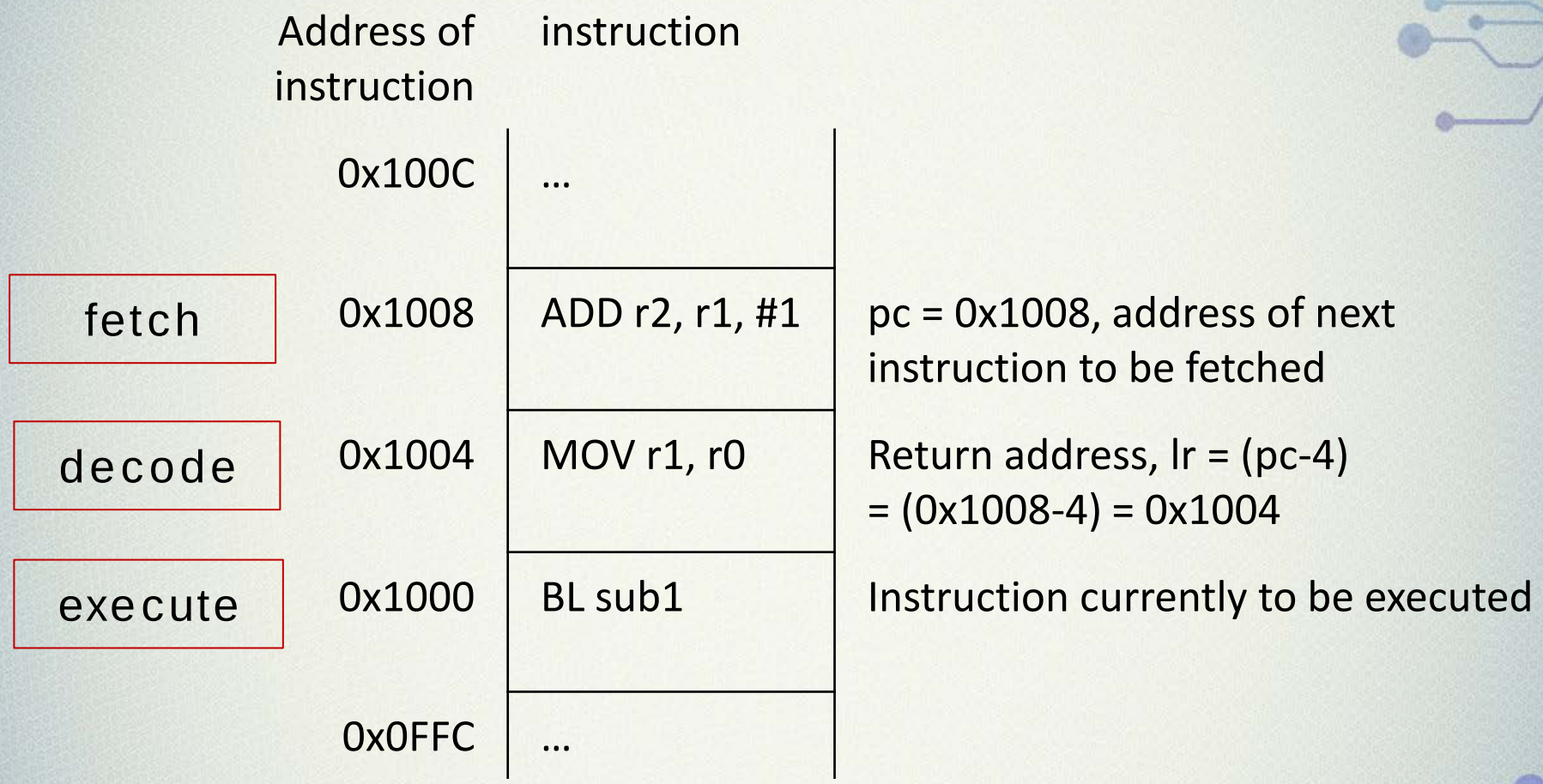
Note:

- There are SIX r14 (lr) registers available in ARM, one for each exception mode.
- There is only ONE register pc

ARM Uses Register Pc (R15) To Fetch The Next Instruction

- Register pc contains the address of the next instruction to be fetched
- The ARM processor uses it to fetch the next instruction
- Hence there is only ONE register pc!
- **By changing the content of register pc, we can change the flow of the program**

The 3-stage Fetch, Decode And Execute Pipeline



Note: each instruction is 4 bytes in size

Assume that we have 5 instructions in our program (instr1 to instr5).

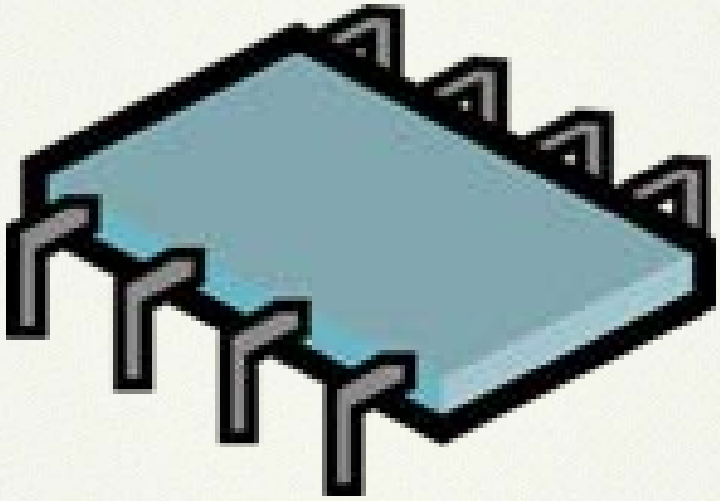
Time period	fetch (pc shows address of next instr to be fetched)	decode	execute
1	Instr1 (address=0x0FFC)	-	-
2	Instr2 (address=0x1000)	Instr1 (address=0x0FFC)	-
3	Instr3 (address=0x1004)	Instr2 (address=0x1000)	Instr1 (address=0x0FFC)
4	**Instr4 (address=0x1008)**	**Instr3 (address=0x1004)**	**Instr2 (address=0x1000)**
5	Instr5 (address=0x100C)	Instr4 (address=0x1008)	Instr3 (address=0x1004)
6	-	Instr5 (address=0x100C)	Instr4 (address=0x1008)
7	-	-	Instr5 (address=0x100C)

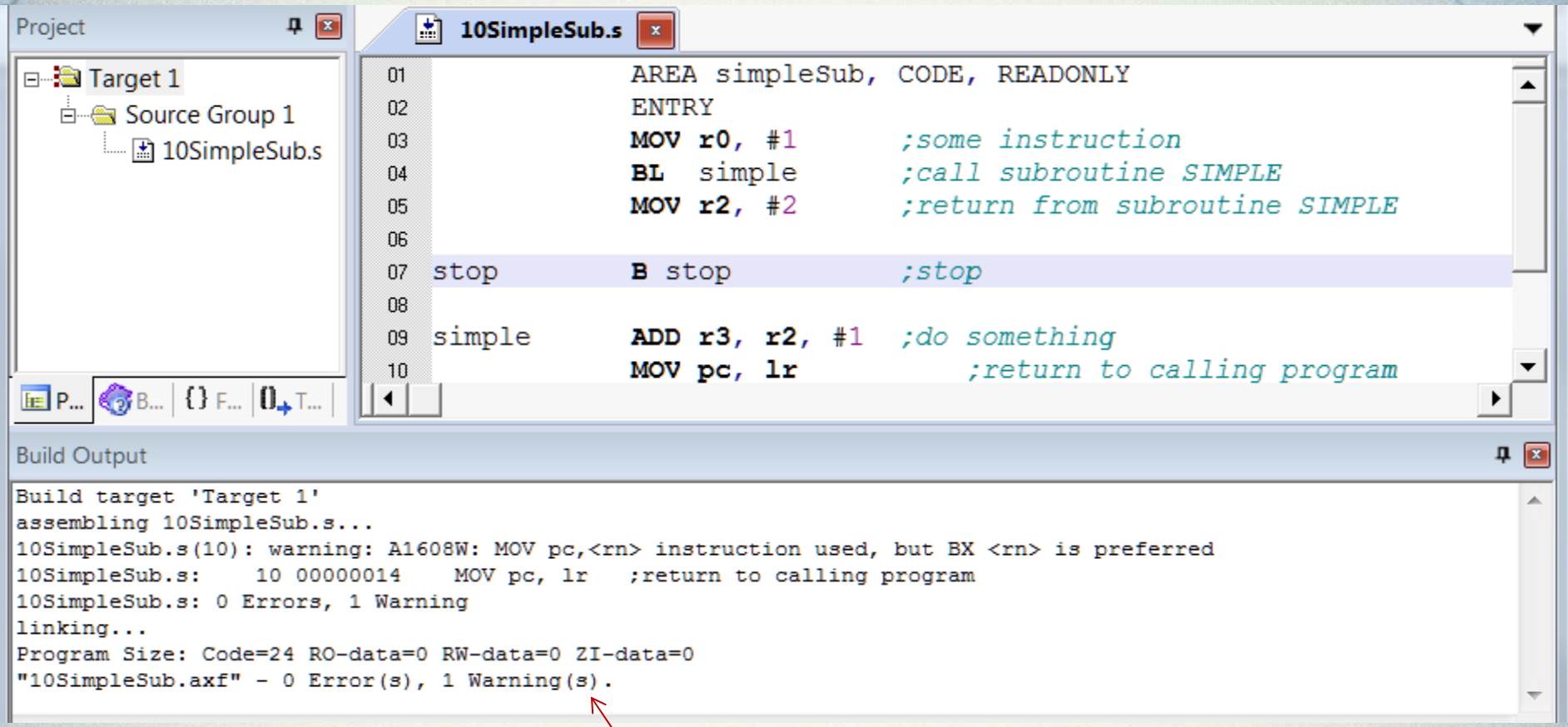
- Assume that Instr2 is our BL sub1 instruction (address = 0x1000).
- While executing this instruction, the processor is about to fetch Instr4 (address = 0x1008, +8 bytes away, 2 instructions away).
- So at this point, register pc = 0x1008 (compare with 0x1000, Instr2 address).
- After finishing the subroutine, we have to return to fetch Instr3 (address = 0x1004, +4 bytes away, 1 instruction away).
- 0x1004 is the return address and has to be stored in register lr before branching.
- Since we only have register pc for reference. Hence, return address is (=pc-4 =0x1008 - 4 =0x1004).
- Hence, before branching to the subroutine, register lr must store this return address, which is (pc-4)!

Return From Subroutine

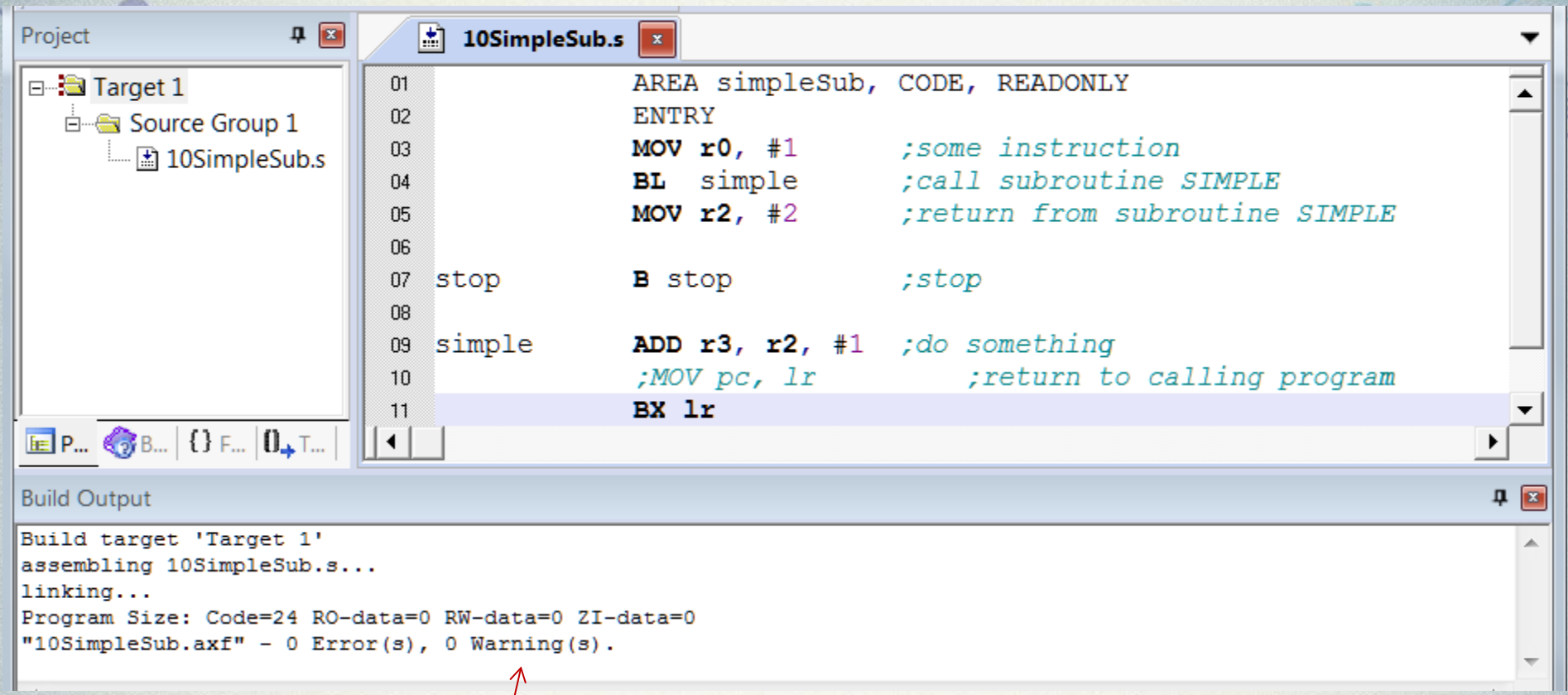
- To return from subroutine, we simply need to copy the content of register lr (r14) to register pc (r15)
- Recall that register lr contains the return address
- This will cause the next instruction to be fetched according to the address stored in pc, which is the return address
- We can use either one of the following instructions
  - MOV pc, lr ;OR
  - BX lr

Demo – A Simple Subroutine

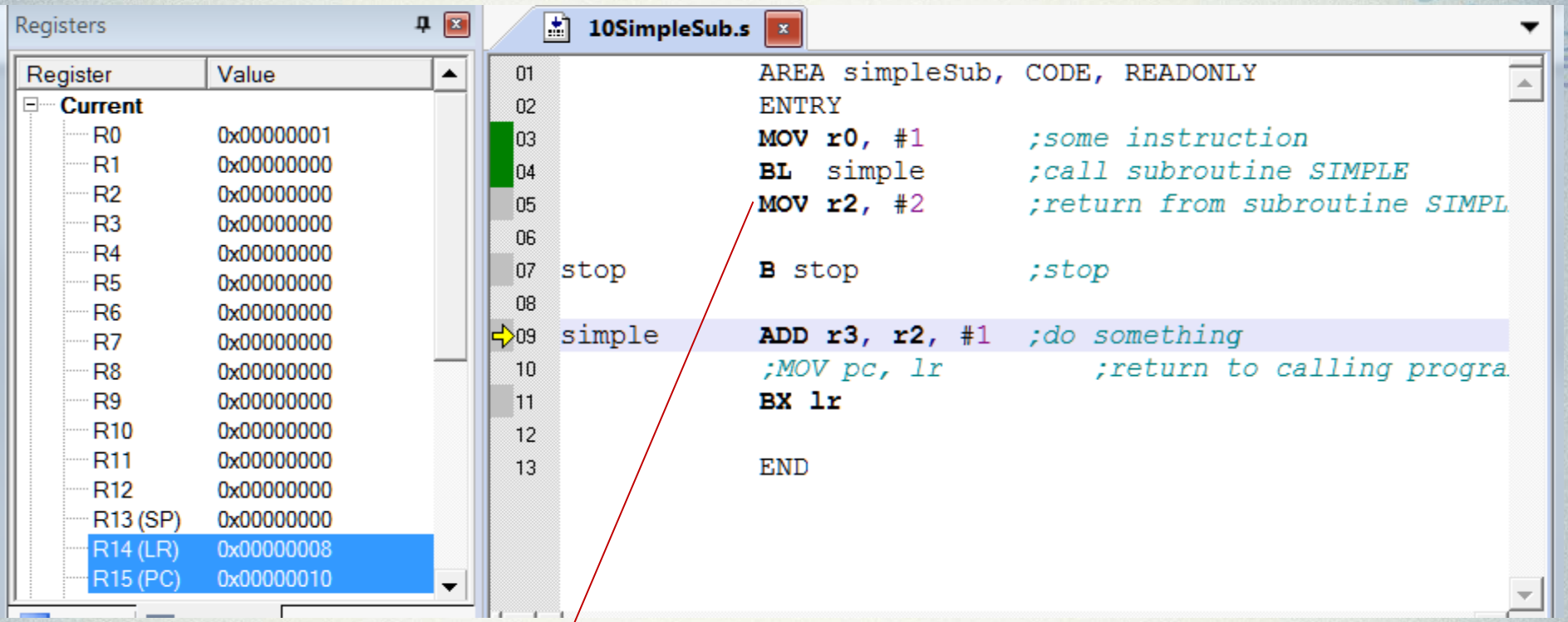




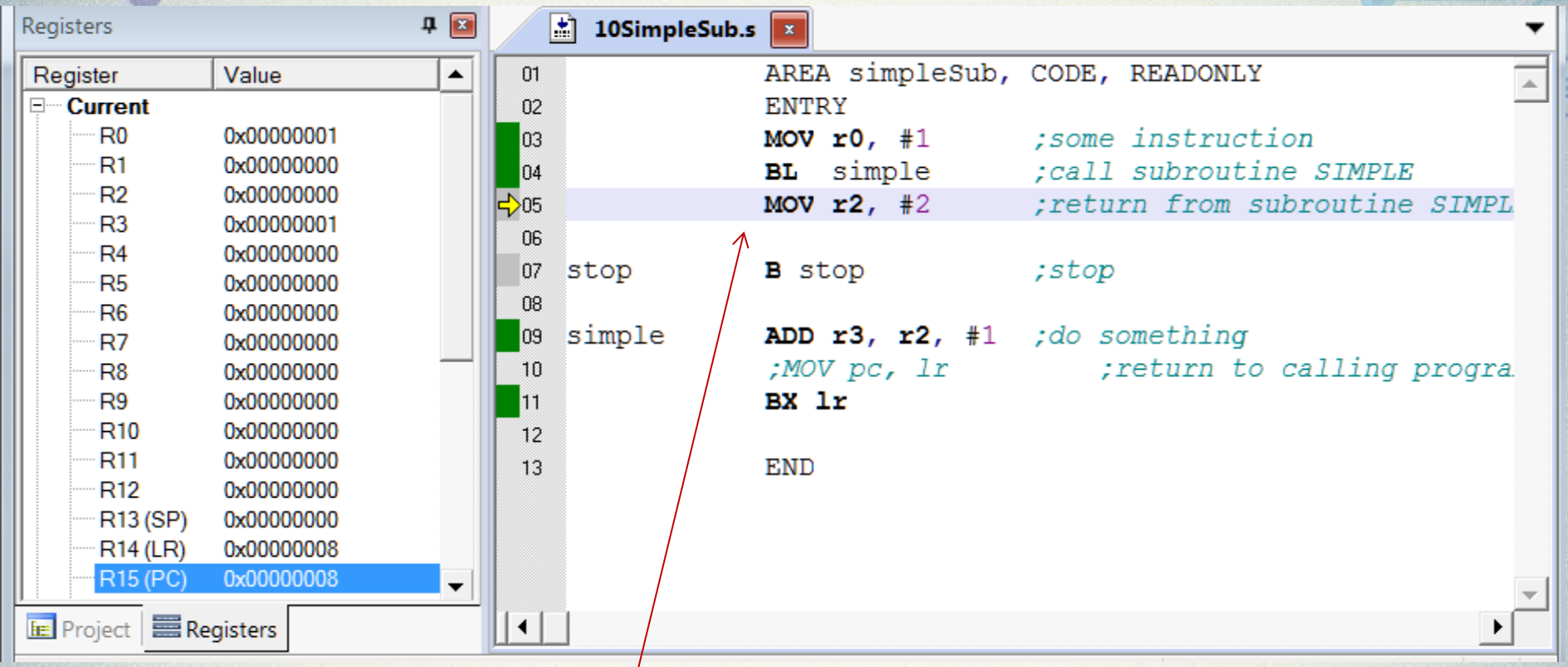
Note the warning!
However, it can still assemble and execute



No warning! If we use BX lr



Return address to lr



Return from subroutine

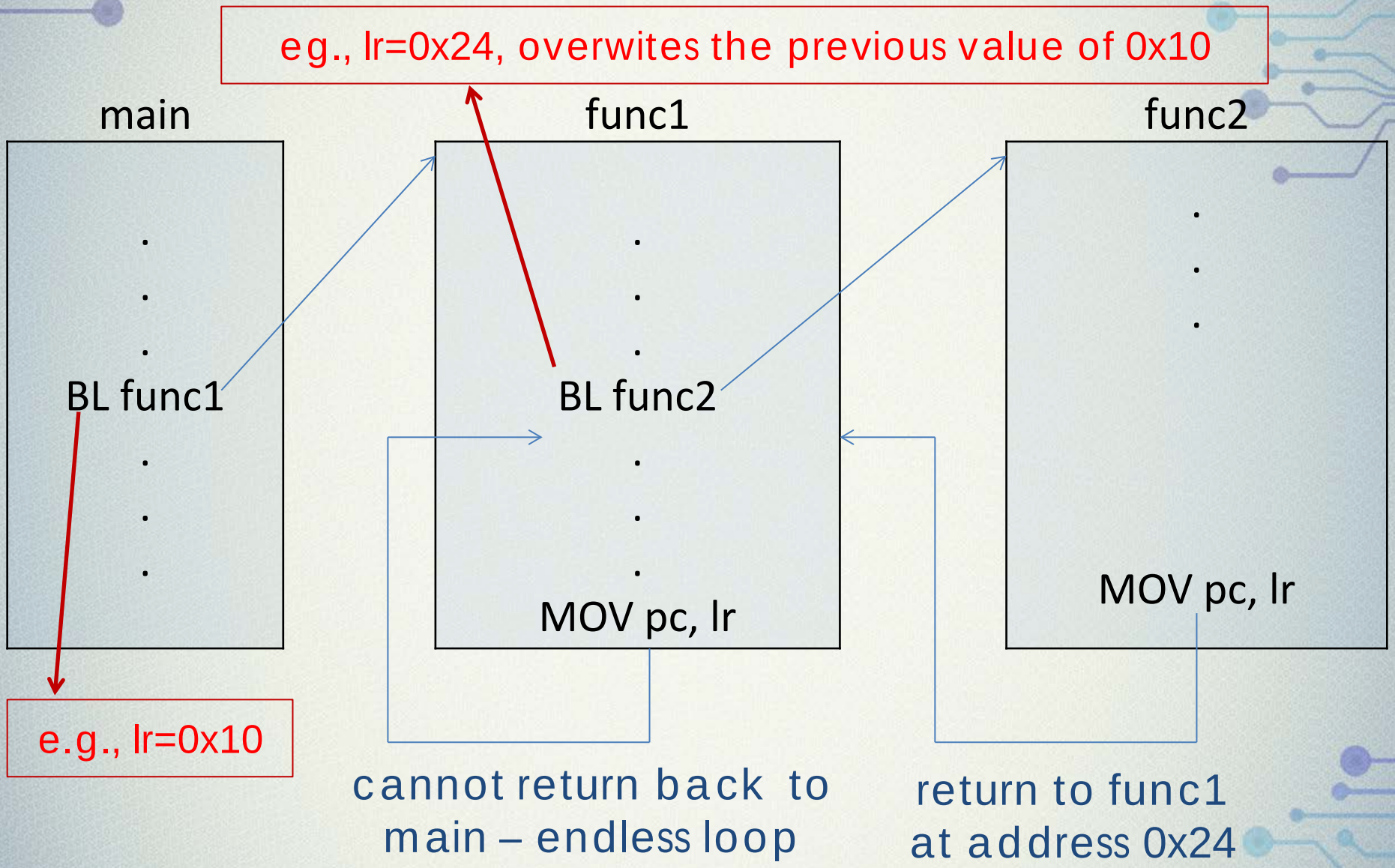
IMPORTANT!!!

- Need to PUSH the content of the link register (lr) to the stack at the start of a subroutine and POP it back to pc before return to calling program for nested subroutine calls that are more than 2 level deep.
- WHY?

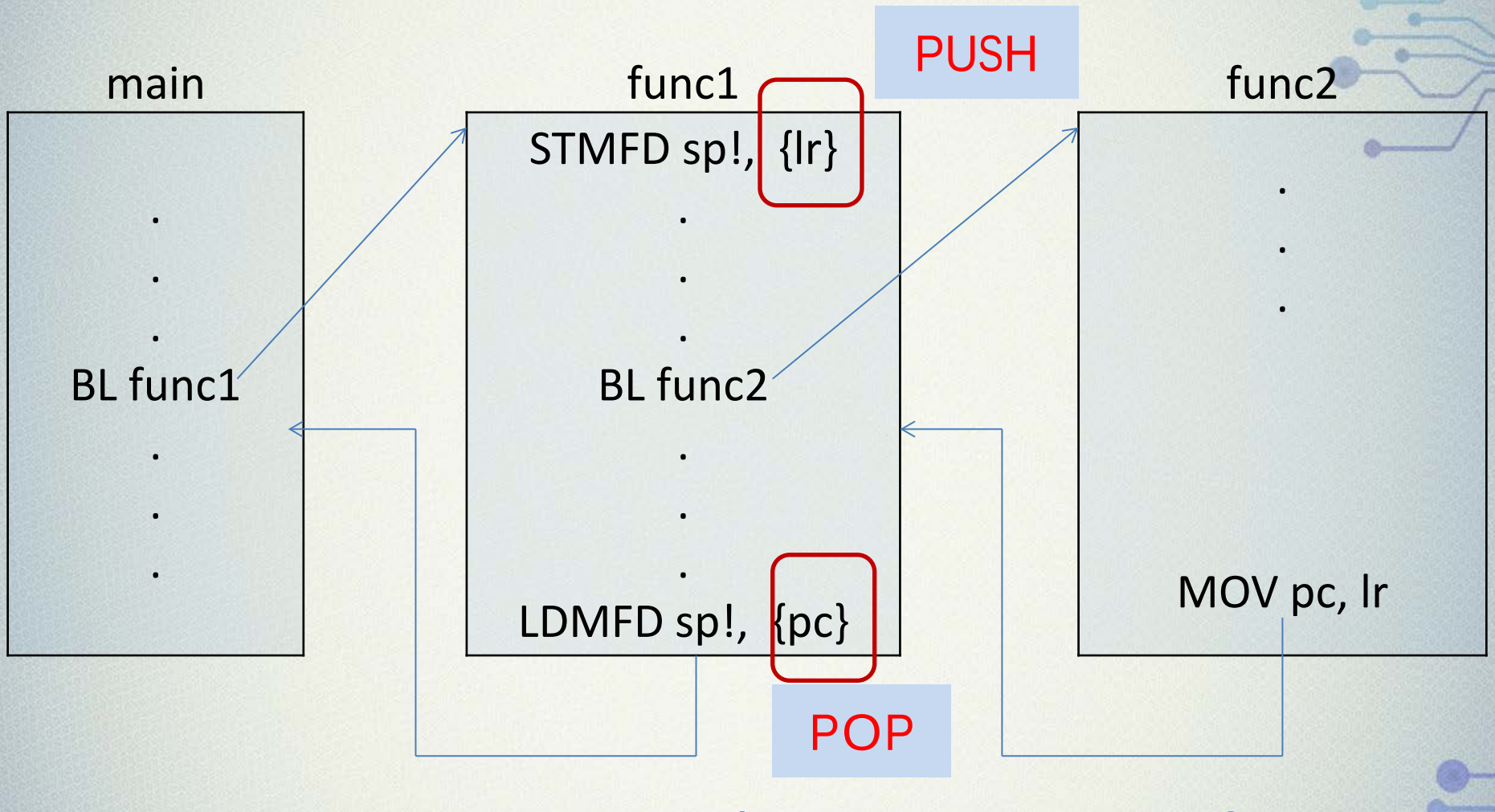
Reason: Else Result In Endless Loop

- Subroutine can call another subroutine
- Next figure shows two nested subroutines, func1 and func2
- If not careful we can get stuck in an infinite loop
- If func1 does not save the content of the link register (lr) at the start, lr will be overwritten when BL func2 is executed (call to func2) within func1
- Upon return from func2 to func1, content in lr is used as return address (address after the BL func2 instruction)
- **When func1 reach the end and wants to return back to main, it cannot return properly!**
- **Instead the instruction after BL func2 will be fetched again**
- **This will result in an endless loop**
- Solve easily by pushing the content of the link register lr at the start of subroutine func1 and pop it to pc at the end of subroutine func1!

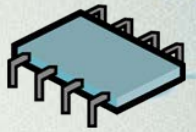
Endless Loop In Nested Subroutine Calls



PUSH The Link Register Lr In Nested Subroutine Calls



Demo – Endless Loop in Nested Subroutine Calls



Registers

Register	Value
Current	
R0	0x00000002
R1	0x00000004
R2	0x000019A1
R3	0x00000051
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000010
R15 (PC)	0x00000018

Project | Registers

10NestedSubEndlessLoop.s

```
01              AREA NestedSubEL, CODE, READONLY
02              ENTRY
03  main        ;compute (x^3 + 1)^4
04              MOV r0, #2   ;x=2
05              BL  func1    ;call subroutine func1
06  stop        B   stop     ;return from func1
07  func1       ;computer (func2+1)^4
08              BL  func2        ;call subroutine fun
09              ADD r2, r2, #1   ;return from func2, r2=func2+1
10              MUL r3, r2, r2
11              MUL r2, r3, r3   ;r2=(func2+1)^4
12              MOV pc, lr       ;return to main
13  func2       ;computer x^3
14              MUL r1, r0, r0
15              MUL r2, r1, r0   ;r2=x^3
16              MOV pc, lr       ;return to func1
17              END
```

endless loop
cannot return back to main

Demo – SOLVED!

1. PUSH lr to stack at the beginning
2. POP from stack to pc at the end

Registers

Register	Value
Current	
R0	0x00000002
R1	0x00000004
R2	0x000019A1
R3	0x00000051
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x40000100
R14 (LR)	0x00000018
R15 (PC)	0x0000000C
CPSR	0x000000D3
SPSR	0x00000000
User/System	

Project | Registers

10NestedSub.s

```
01 ram_base EQU 0x40000000
02          AREA NestedSub, CODE, READONLY
03          ENTRY
04 main     ;compute (x^3 + 1)^4
05          LDR sp, =ram_base + 0x100
06          MOV r0, #2   ;x=2
07          BL  func1    ;call subroutine func1
08 stop     B   stop     ;return from func1
09 func1    ;computer (func2+1)^4
10          STMFD sp!, {lr} ;PUSH lr to stack
11          BL  func2       ;call subroutine fun
12          ADD r2, r2, #1  ;return from func2, r2=func2+1
13          MUL r3, r2, r2
14          MUL r2, r3, r3  ;r2=(func2+1)^4
15          LDMFD sp!, {pc} ;return to main by POP to pc
16 func2    ;computer x^3
17          MUL r1, r0, r0
18          MUL r2, r1, r0  ;r2=x^3
19          MOV pc, lr      ;return to func1
20          END
```

1

2

Passing Parameters To Subroutine

- Subroutines are written as general as possible
- Data or addresses need to be able to move in/out of subroutines → parameters for the subroutine
- Parameters can be passed through registers or stack or a mixture of both
- By convention
  - registers r0-r3 are used to pass input parameters to subroutine
  - register r0 to return result back to the caller
- Use of any other registers (r4-r12) in subroutine
  - PUSH to stack before use → save environment
  - POP from stack before return to caller → restore environment

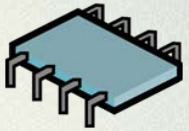
Passing Parameters Through Registers

Calling program	Subroutine
1) Sets some registers to contain inputs	
2) Calls subroutine	
	3) Extract contents from registers that contains inputs
	4) Performs the action of the subroutine
	5) Places the outputs in some registers and return
6) Read from those registers containing outputs	

Passing Parameters Through Registers

- Example: in a subroutine
  - r0 = option: “0” → shift left; “1” → shift right
  - r1 = operand to be shifted
  - r2 = shift m bits
  - r3 = result
  - r4 = temporary register
- 3 parameters (r0, r1, r2) need to be passed to the subroutine
- So its value need to be defined before calling subroutine
- r4 is used as a temporary register in the subroutine, so it must first be stacked before it is used in the subroutine, else the previous value before the call will be corrupted by the subroutine

Demo - Passing Parameters Through Registers (1)



```
ram_base EQU 0x40000000
     AREA PassParaReg, CODE, READONLY
    ENTRY
main
    LDR  sp, =ram_base + 0x100
    MOV r0, #0   ;shift left
    LDR  r1, =0x00012340   ;operand
    MOV r2, #4   ;shift 4 bits
    BL    shiftfunc    ;call subroutine shiftfunc
stop B   stop      ;return from shiftfunc
```

Demo - Passing Parameters Through Registers (2)

```
shiftfunc
    STMFD  sp!, {r4, lr}   ;PUSH r4, lr to stack
    MOV    r4, #0   ;r4 - temporary register
    CMP    r0, r4
    MOVEQ r3, r1, LSL r2    ;r0=0 implies shift left
    MOVNE r3, r1, LSR r2    ;else shift right
    LDMFD  sp!, {r4, pc}    ;POP r4 to restore r4 and
        ;POP lr to pc to return
    END
```

Passing Parameters Through The Stack

- Data is PUSH before the subroutine call
- Subroutine grabs the data from the stack for processing
- Results are stored back into the stack to be retrieved later by the calling program

Passing Parameters through the Stack

Calling program	Subroutine
1) Pushes inputs onto the Stack	
2) Calls subroutine	
	3) Extract inputs from the stack (pops)
	4) Performs the action of the subroutine
	5) Pushes outputs on the stack and return
6) Stack contain outputs (pop)	
7) Balance stack	

Demo - Passing Parameters Through The Stack (1)

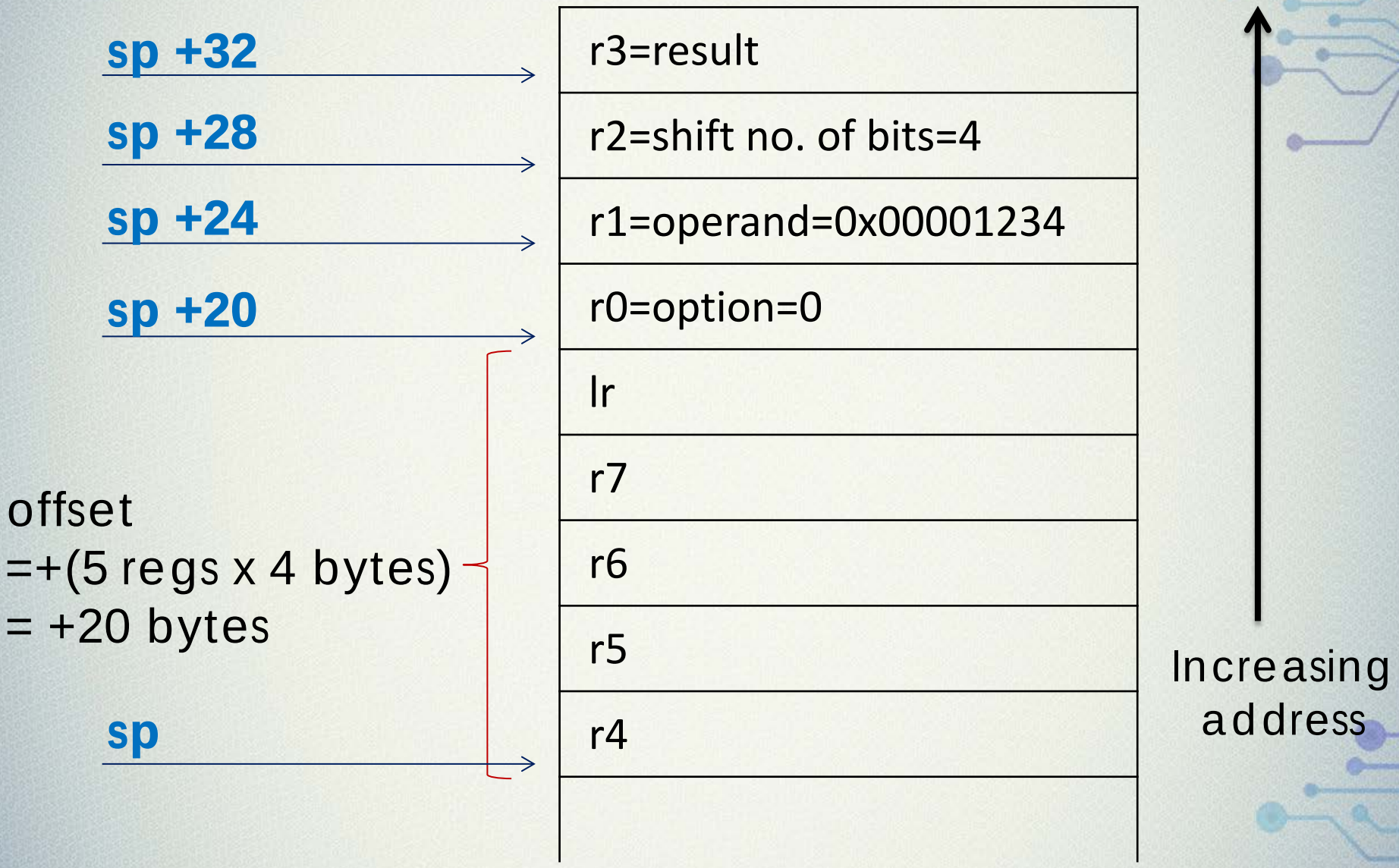
```
ram_base EQU 0x40000000
    AREA PassParaStack, CODE, READONLY
    ENTRY
main
    LDR  sp, =ram_base + 0x100
    MOV r0, #0   ;option=0 implies shift left
    LDR  r1, =0x00001234   ;operand
    MOV r2, #4   ;shift 4 bits

    STMFD sp!, {r0-r3}    ;PUSH in/out parameters to stack
    BL    shiftfunc    ;call subroutine shiftfunc
    LDMFD sp!, {r0-r3}   ;POP in/out parameters from stack
        ;result in r3
stop B   stop      ;return from shiftfunc
```

Demo - Passing Parameters Through The Stack (2)

```
shiftfunc
    STMFD sp!, {r4-r7, lr};PUSH r4-r7, lr to stack (5 regs)
    LDR     r4, [sp, #20]    ;r4=option, offset=5*4=20 bytes
    LDR     r5, [sp, #24]    ;r5=operand
    LDR     r6, [sp, #28]    ;r6=shift no. of bits
    MOV     r7, #0   ;r7 - temporary register
    CMP     r4, r7
    MOVEQ r7, r5, LSL r6    ;r0=0 implies shift left
    MOVNE r7, r5, LSR r6    ;else shift right
    STR     r7, [sp, #32];store result in stack
    LDMFD sp!, {r4-r7, pc} ;POP r4-r7 to restore them and
        ;pop lr to pc to return
    END
```

Content Of Stack In Shiftfunc After 1st Instruction, STMFD Sp!, {R4-r7, Lr}



The ARM APCS (AAPCS)

- Application Procedure Call Standard → a standard
- Defines how subroutines can be separately written, separately compiled and separately assembled
- Contract between subroutine callers and callees
- Standard specifies
  - how parameters be passed to subroutines
  - which registers must have their content preserved (which are corruptible)
  - special roles for certain registers
  - a Full Descending stack pointed by r13 (sp) etc

AAPCS Simplified Specifications

Register	Notes
r0 – r3	Parameters to and results from subroutines. Otherwise may be corrupted.
r4 – r11	Variables. Must be preserved.
r12	Scratch register (corruptible), use by linkers
r13	Stack pointer (sp)
r14	Link register (lr)
r15	Program counter (pc)

Example: Design And Write An ARM Subroutine That Fills A Sequence Of Words In Memory With Same 32-bit Value

- Pseudo-code solution

```
fillmem (address, length, value) {
   count = 0;
   while (count < length) {
      memory.word[address] = value;
      address = address + 4;
      count = count + 1;
   }
}
```

- 3 parameters
  - address – start address in memory
  - length – number of words to store
  - value – value to store

A Program To Test The Subroutine (1)

```
main
  LDR     r0,  =tstarea          ;Load address to be filled
  LDR     r1,  =32               ;Load number of words to be
   filled
  LDR     r2,  =0xA0B0C0D0 ;Load value to fill
  LDR     sp,  =ram_base + 0x1000
  STMFD  sp!, {r0-r2}            ;push parameters on stack

  BL       fillmem   ;call fillmem subroutine to fill up a
        ;portion of memory with a value

  LDMFD  sp!, {r0-r2}   ;pop parameters off the stack

  stop      B stop
```

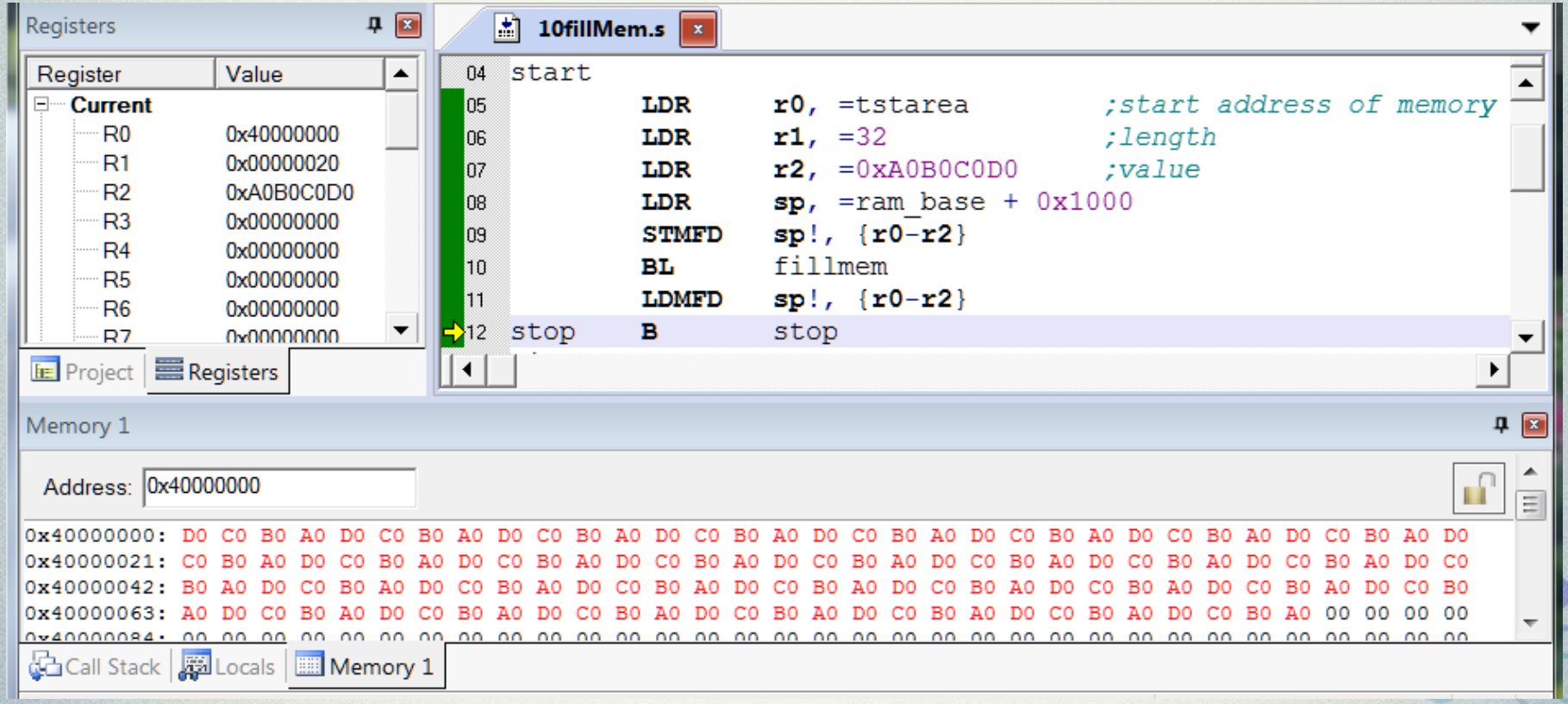
Fillmem Subroutine (2)

```
fillmem
        STMFD  sp!, {r4-r7, lr}          ;save registers, 5 altogether

        LDR      r4, [sp, #20]           ;load address parameter
        LDR      r5, [sp, #24]               ;load length parameter
        LDR      r6, [sp, #28]               ;load value parameter
        MOV      r7, #0                      ;count = 0
loop
        CMP      r7, r5                  ;count < length
        BHS      endloop                 ;ends if higher or same
        STR      r6, [r4, r7, LSL #2]  ;memory[count*4]=value
        ADD      r7, r7, #1               ;count = count + 1
        B        loop
endloop
        LDMFD  sp!, {r4-r7, pc}          ;pop parameters off the stack
```

Fillmem Subroutine (3)

```
    AREA test, DATA, READWRITE
tstarea     SPACE 256       ;reserve 256 bytes
    END
```



Summary

- LDM/STM instructions
- Stacks – FD, FA, ED, EA
- Implementation of PUSH, POP operations using STM and LDM instructions
- Subroutine, calls and return
  - Call instruction – BL label
  - Return instructions
  - Registers sp, pc, lr
- Nested subroutine call
- Parameters Passing
  - Through registers
  - Through stack
  - Standards, AAPCS