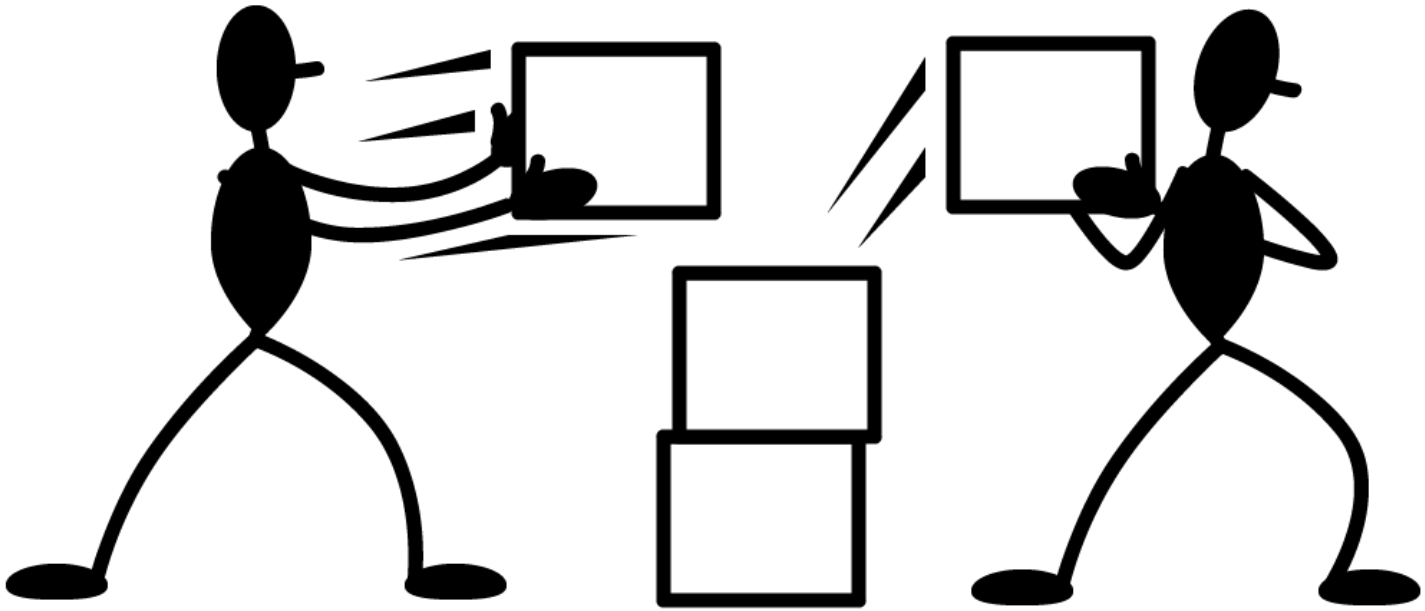
ESET 349 – Microcontroller Architecture

Stack

Dr. Muhammad Faeyz Karim

Stack



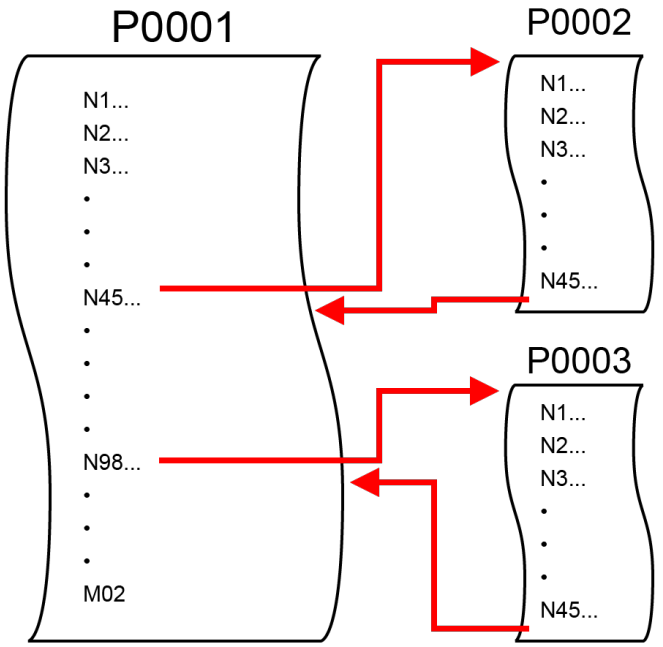
Purpose Of Using Stacks

- 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 → **STM**
- 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

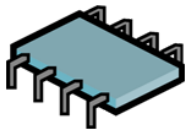
STM {<cond>}<address-mode> <Rn>{!},<reg-list>{^}

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

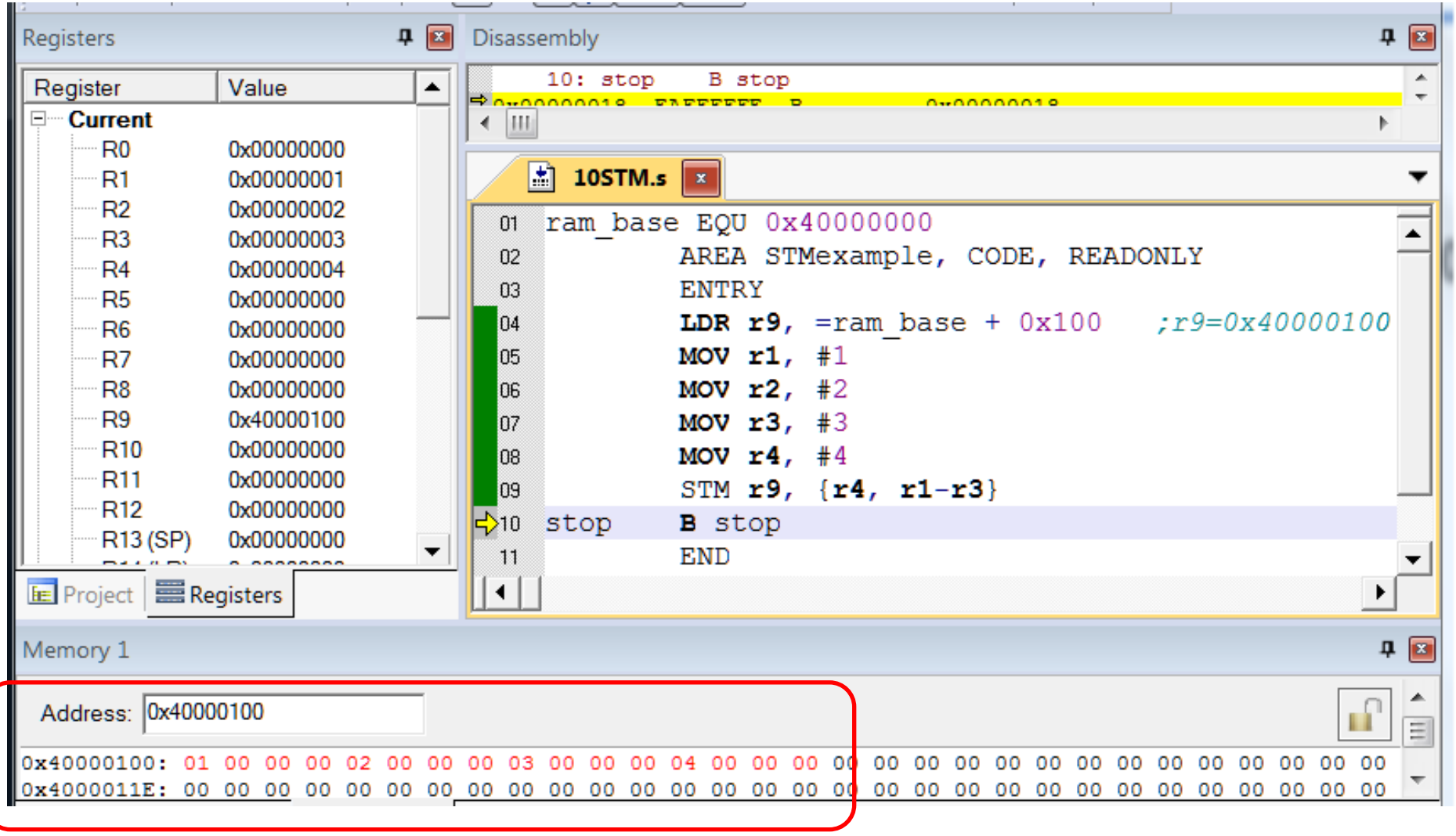
STM Instruction: Example

- **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 STR 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



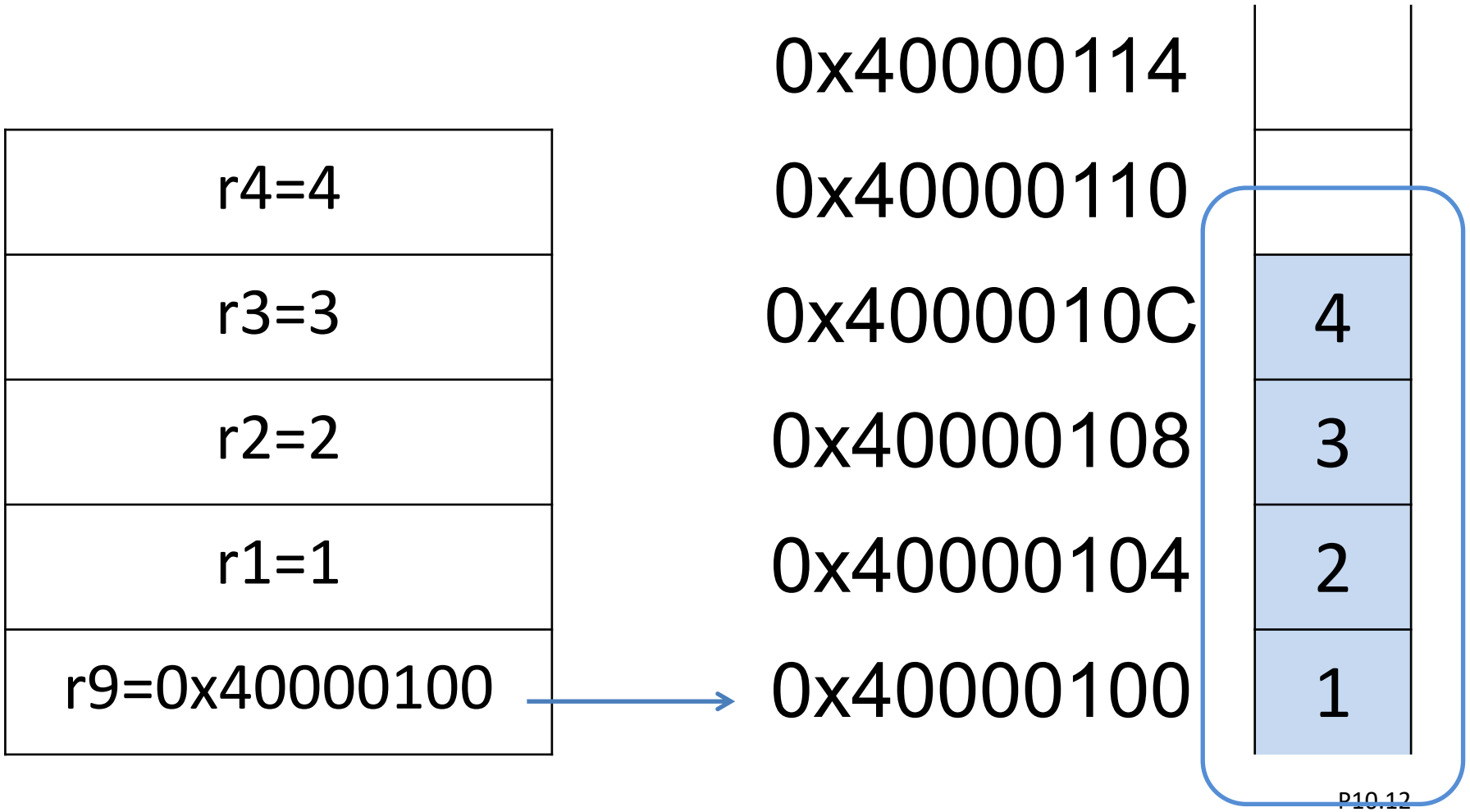
```
ram_base EQU 0x40000000
    AREA STMexample, CODE, READONLY
    ENTRY
    LDR  r9, =ram_base + 0x100   ;r9=0x40000100
    MOV r1, #1
    MOV r2, #2
    MOV r3, #3
    MOV r4, #4
    STM r9, {r4, r1-r3}
stop    B stop
    END
```



Note:

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

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



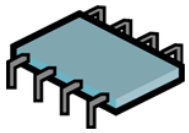
Syntax Of LDM Instruction

- 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

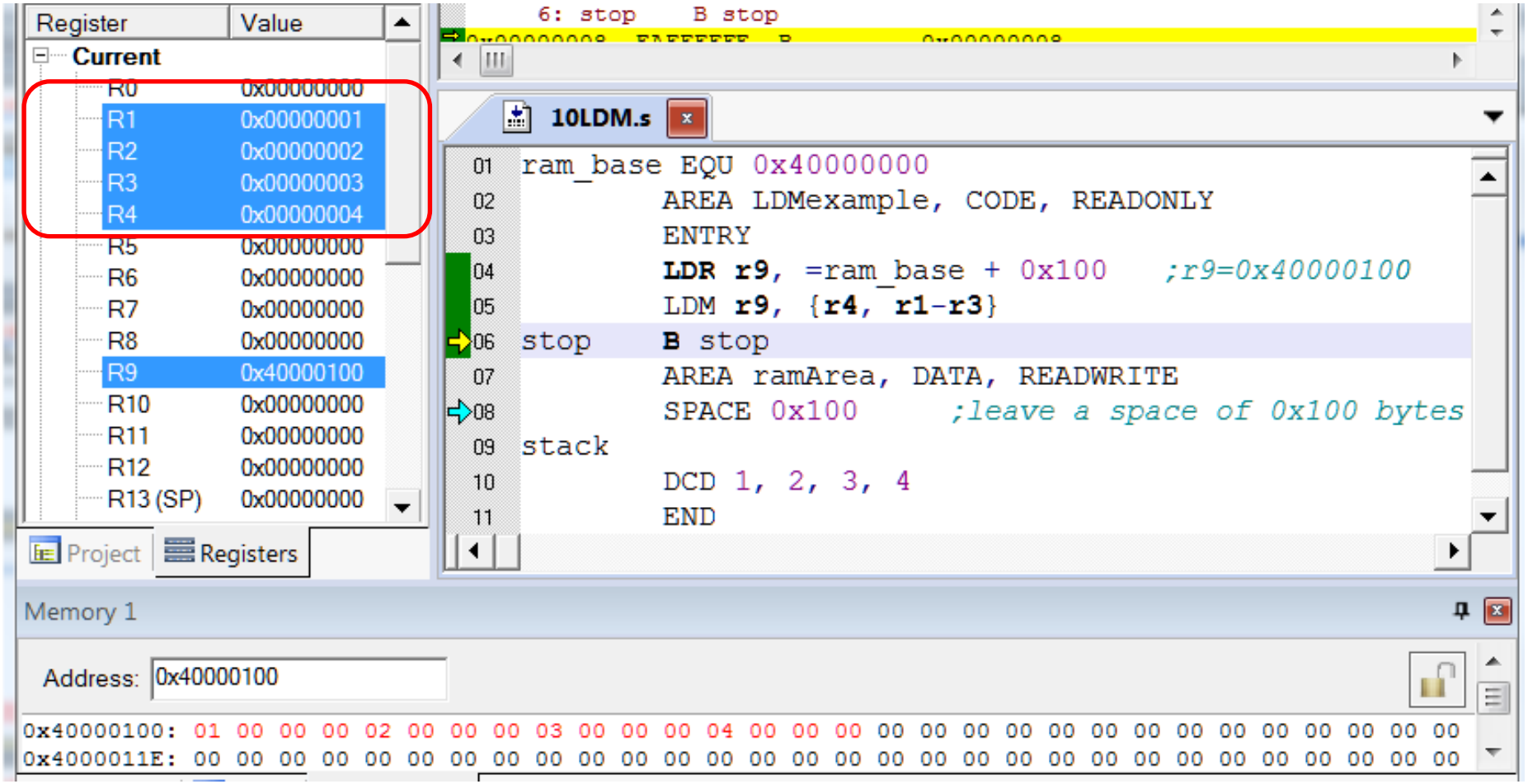
LDM Instruction: Example

- **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



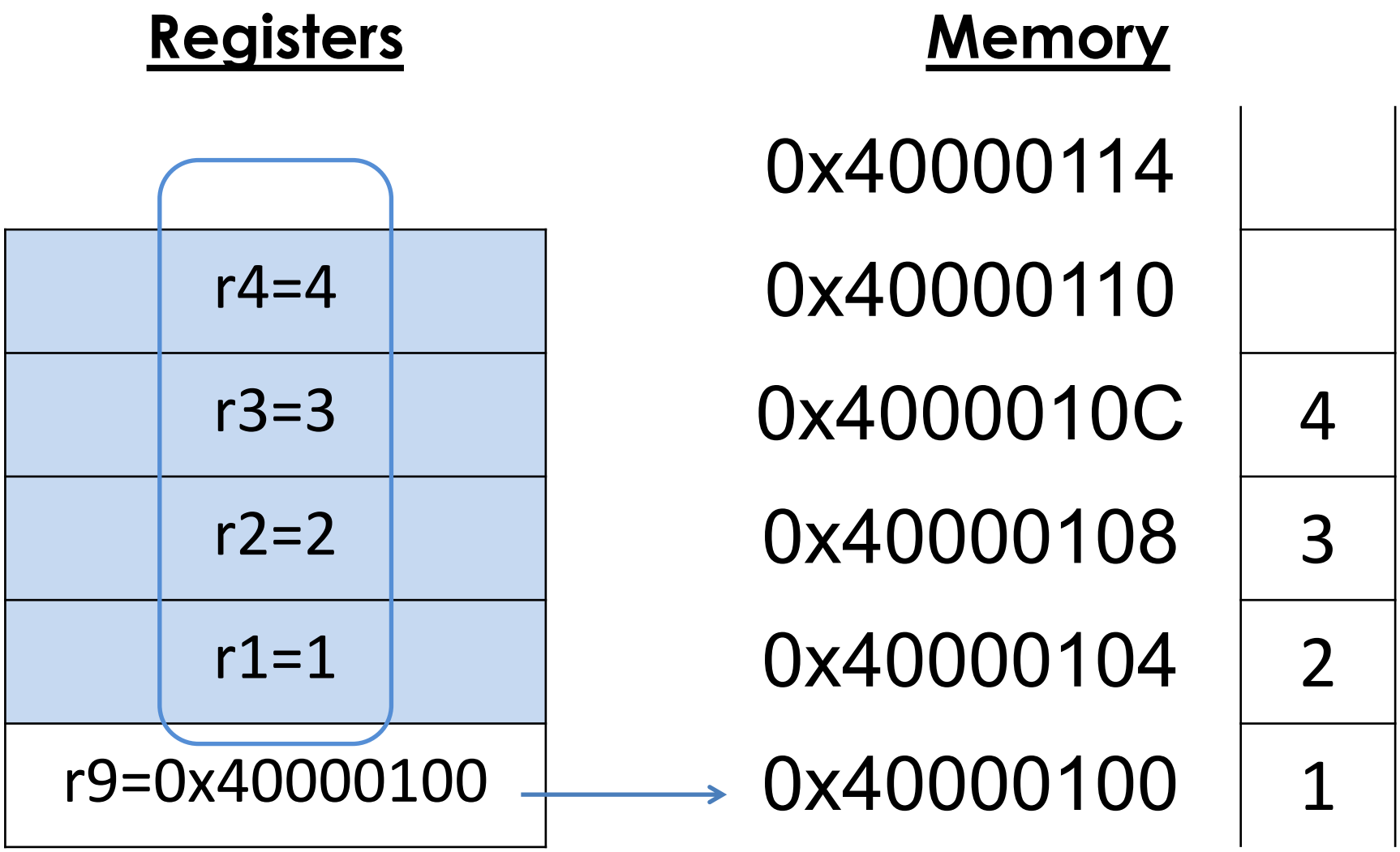
```
ram_base EQU 0x40000000
    AREA LDMexample, CODE, READONLY
    ENTRY
    LDR r9, =ram_base + 0x100    ;r9=0x40000100
    LDM r9, {r4, r1-r3}
stop    B stop
    AREA ramArea, DATA, READWRITE
    SPACE 0x100 ;leave a space of 0x100 bytes
stack
    DCD 1, 2, 3, 4
    END
```



Note:

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

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



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

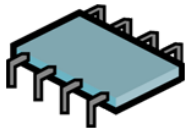
STMDB r9, {r4, r1, r2-r3}

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

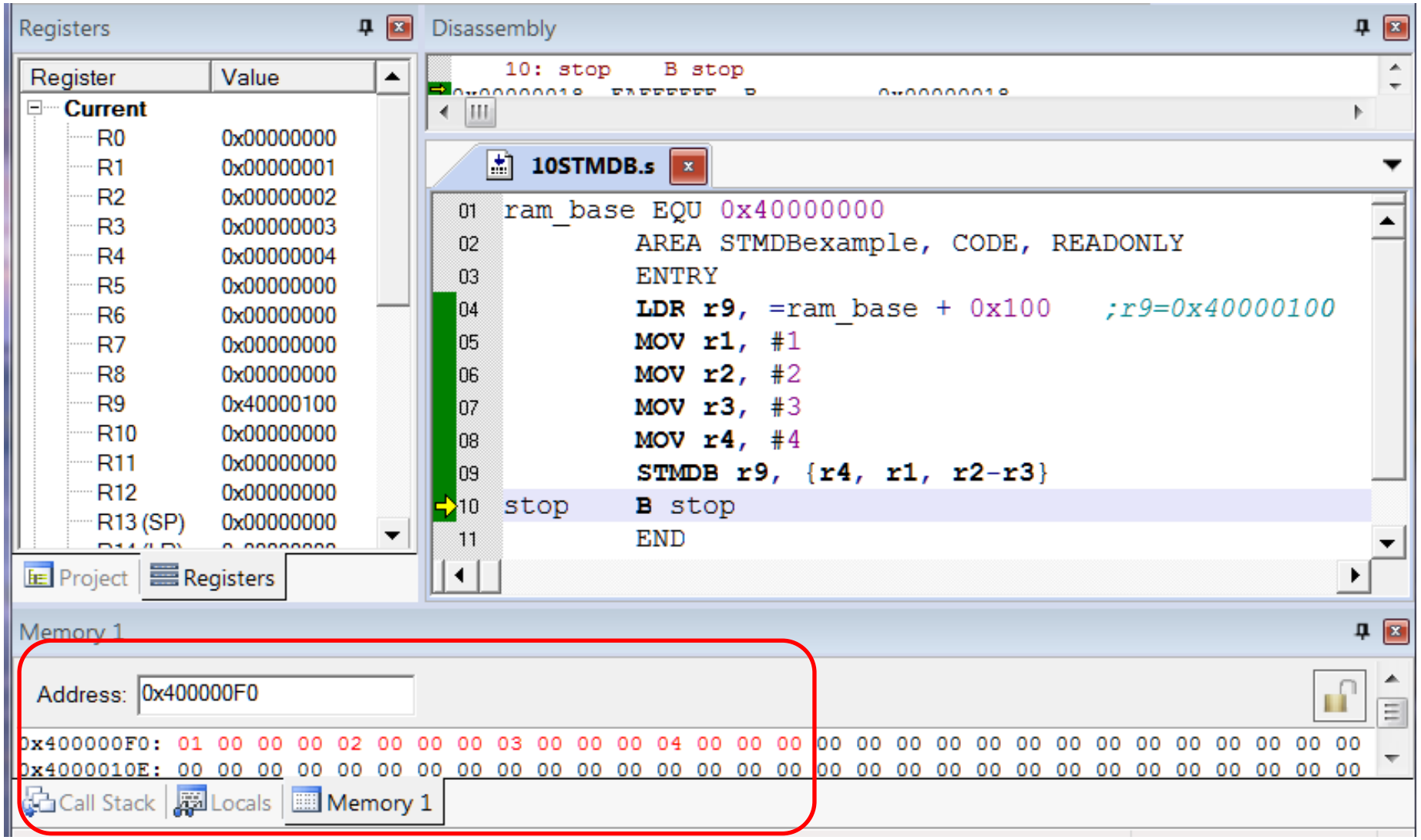
Same effect as having 4 separate STR instructions operating at the same time

- 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



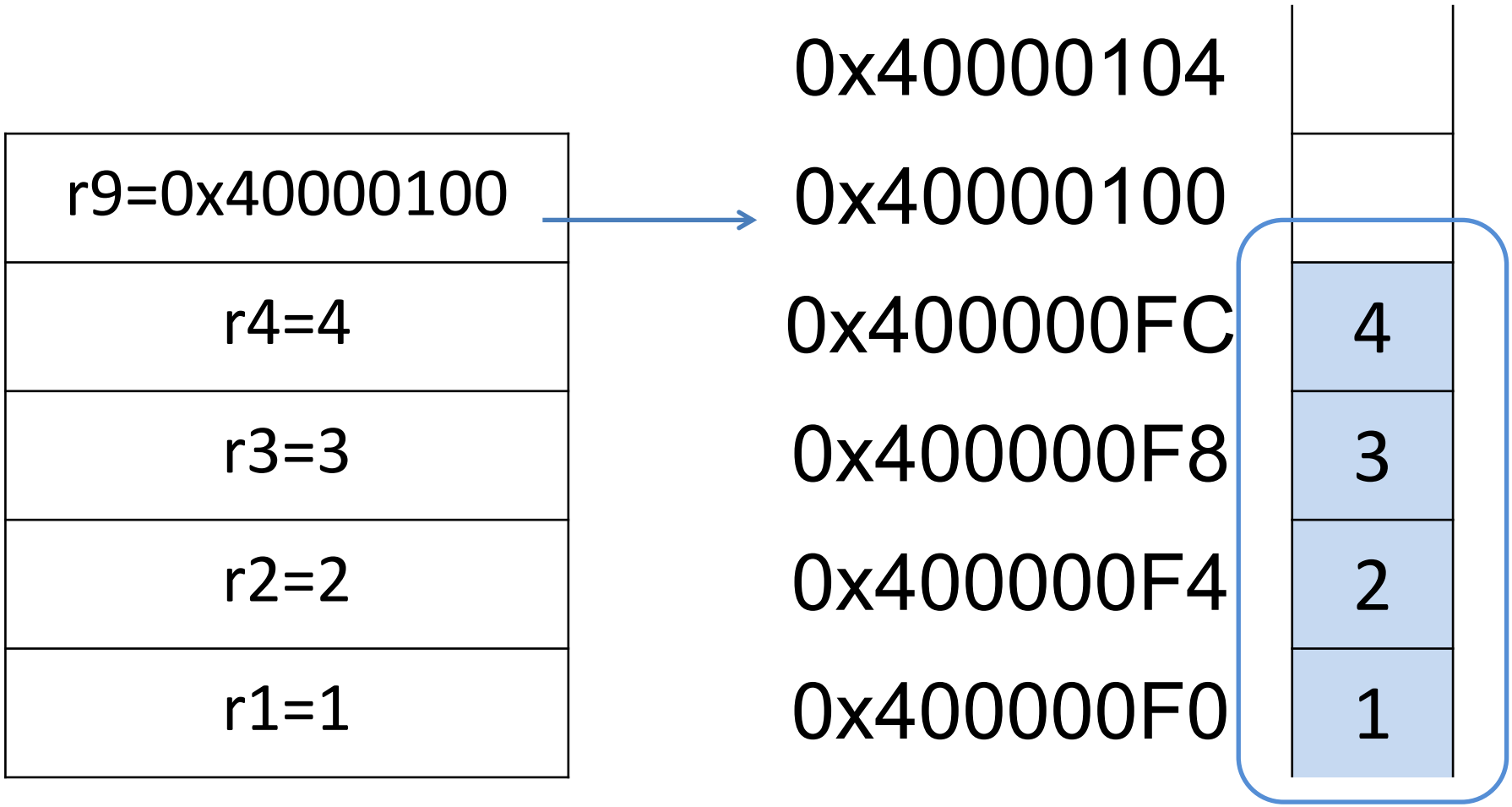
```
ram_base EQU 0x40000000
    AREA STMDBexample, CODE, READONLY
    ENTRY
    LDR r9, =ram_base + 0x100    ;r9=0x40000100
    MOV r1, #1
    MOV r2, #2
    MOV r3, #3
    MOV r4, #4
    STMDB r9, {r4, r1, r2-r3}
stop    B stop
    END
```



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

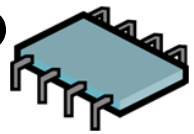
LDMDA r9, {r4, r1, r2, r3}

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

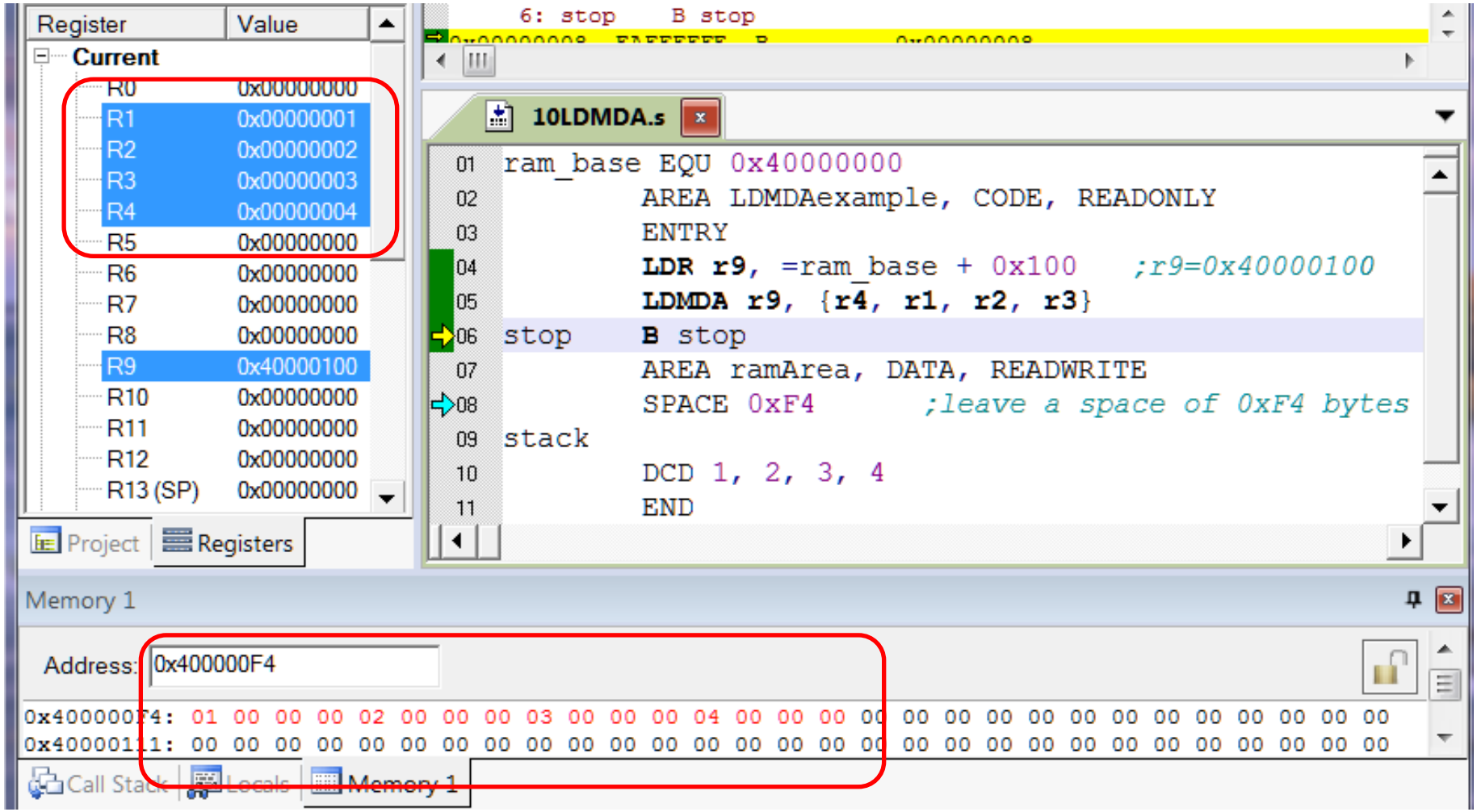
Same effect as having 4 separate LDR instructions operating at the same time

- 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



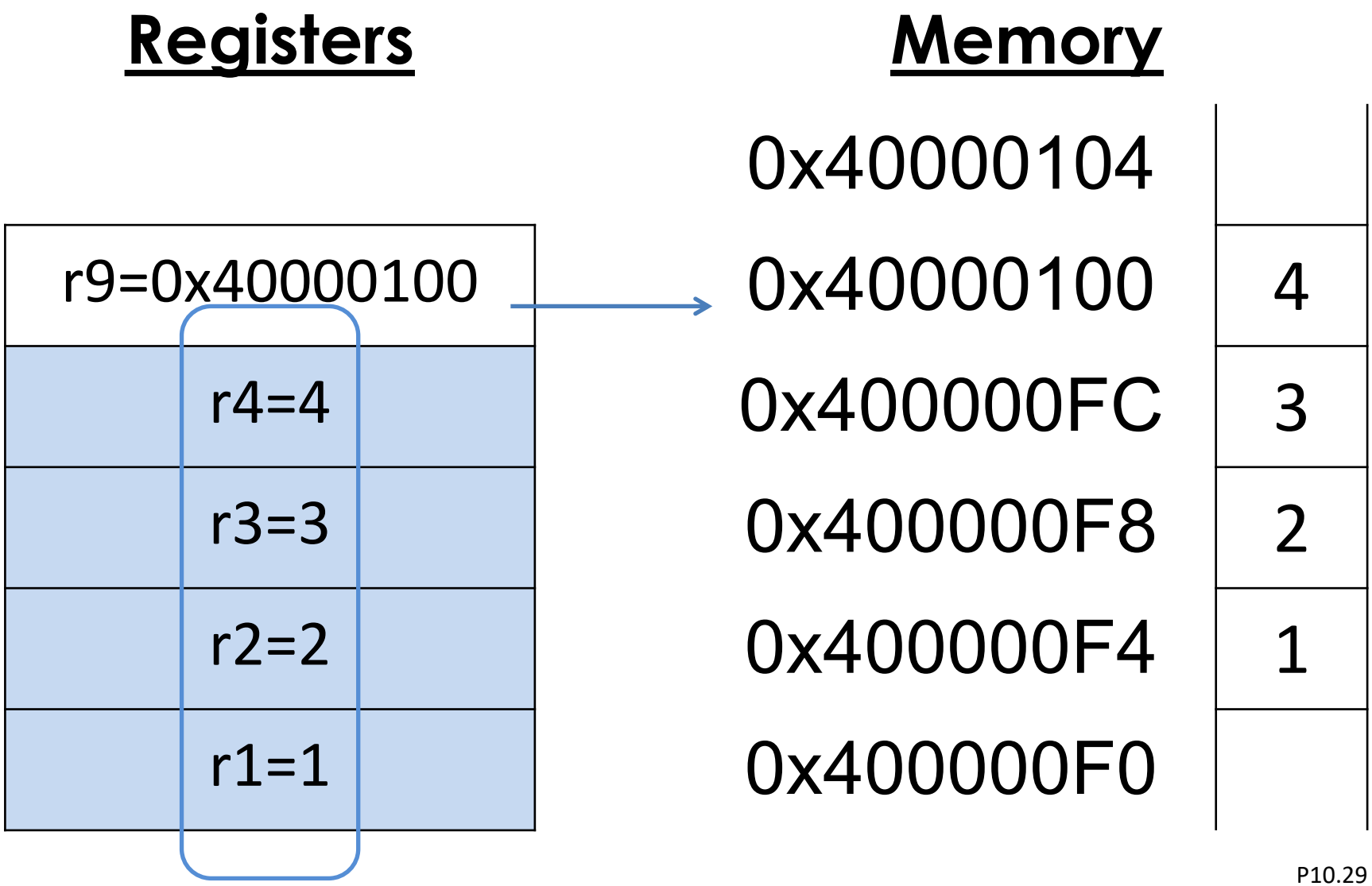
```
ram_base EQU 0x40000000
    AREA LDMDAexample, CODE, READONLY
    ENTRY
    LDR r9, =ram_base + 0x100    ;r9=0x40000100
    LDMDA r9, {r4, r1, r2, r3}
stop    B stop
    AREA ramArea, DATA, READWRITE
    SPACE 0xF4   ;leave a space of 0xF4 bytes
stack
    DCD 1, 2, 3, 4
    END
```



Note:

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

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



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)

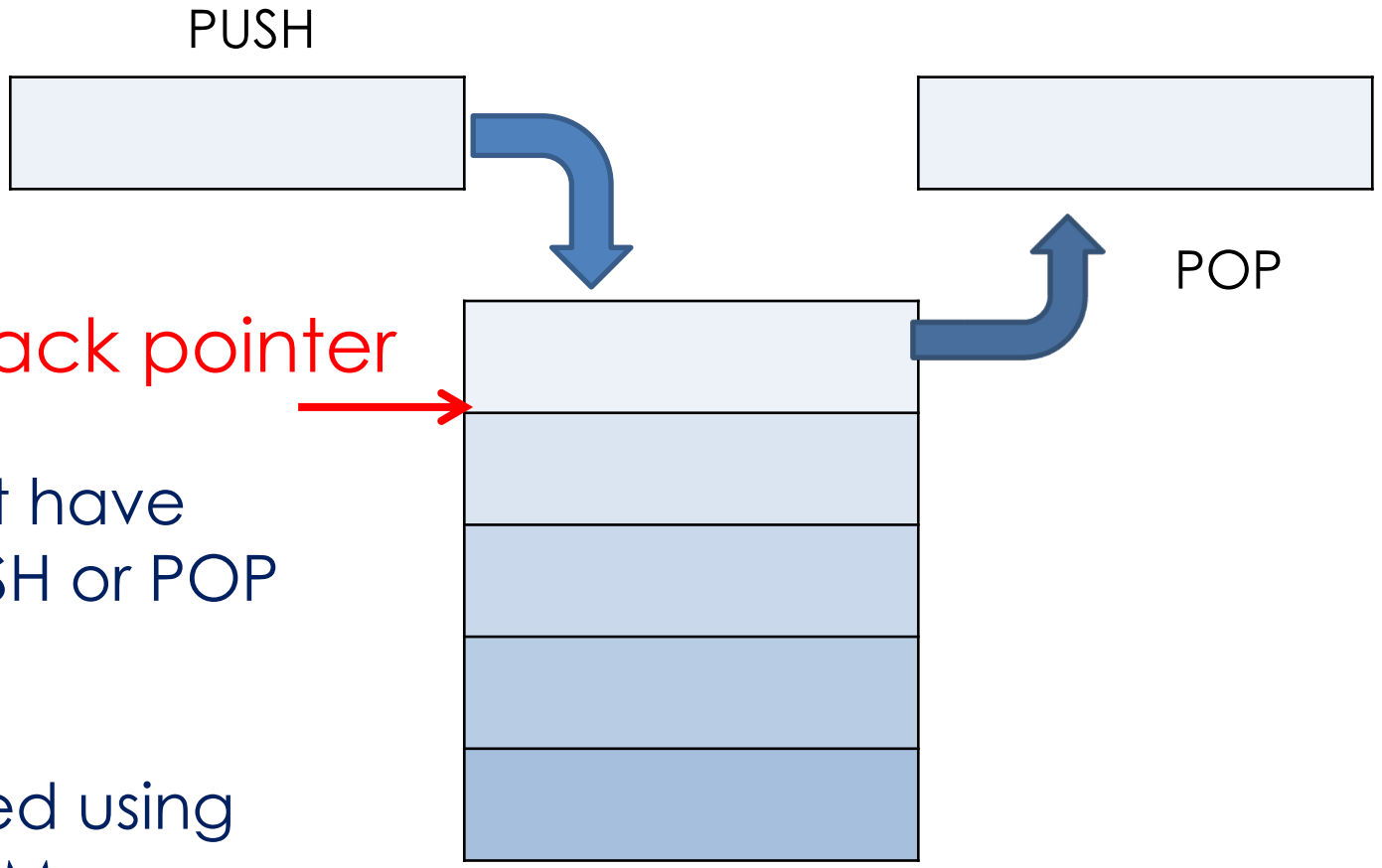
Stack (2)

- 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)

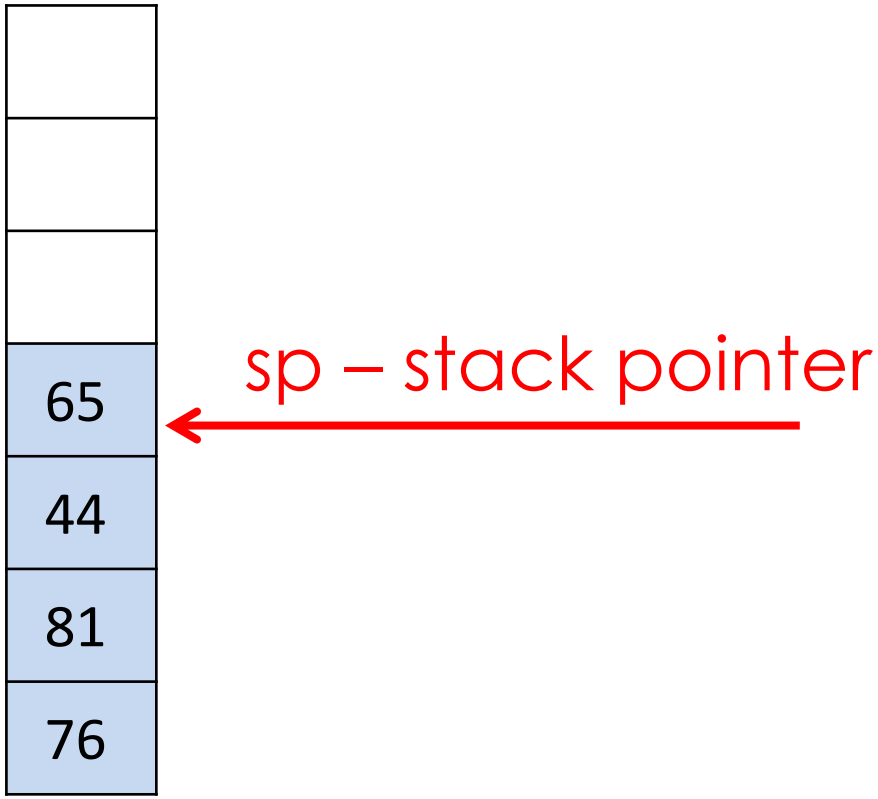
Note:

- 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?

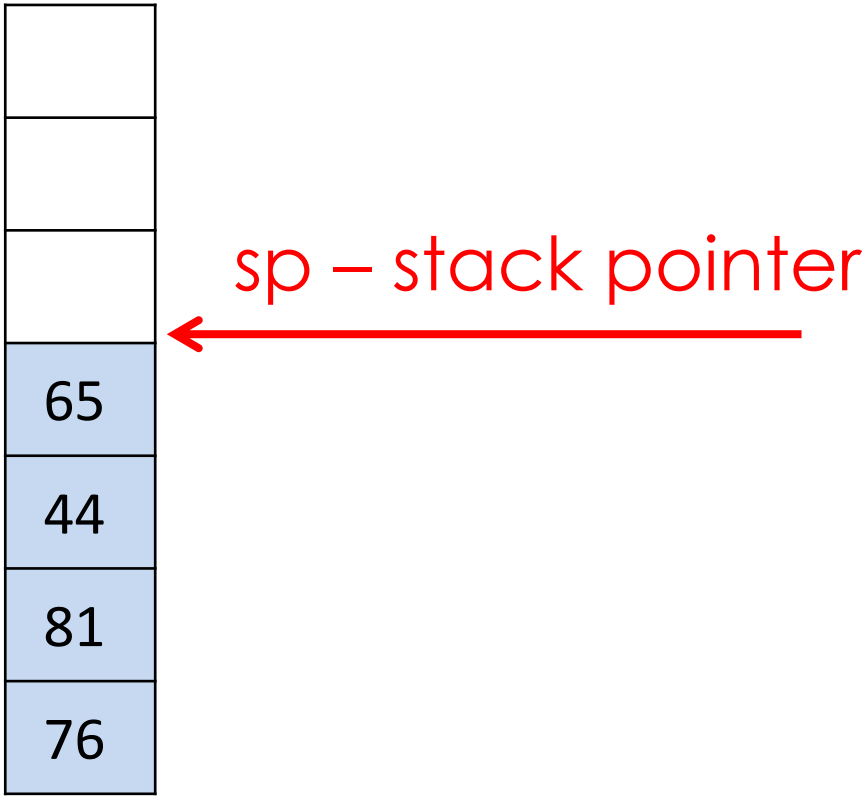
Full

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:

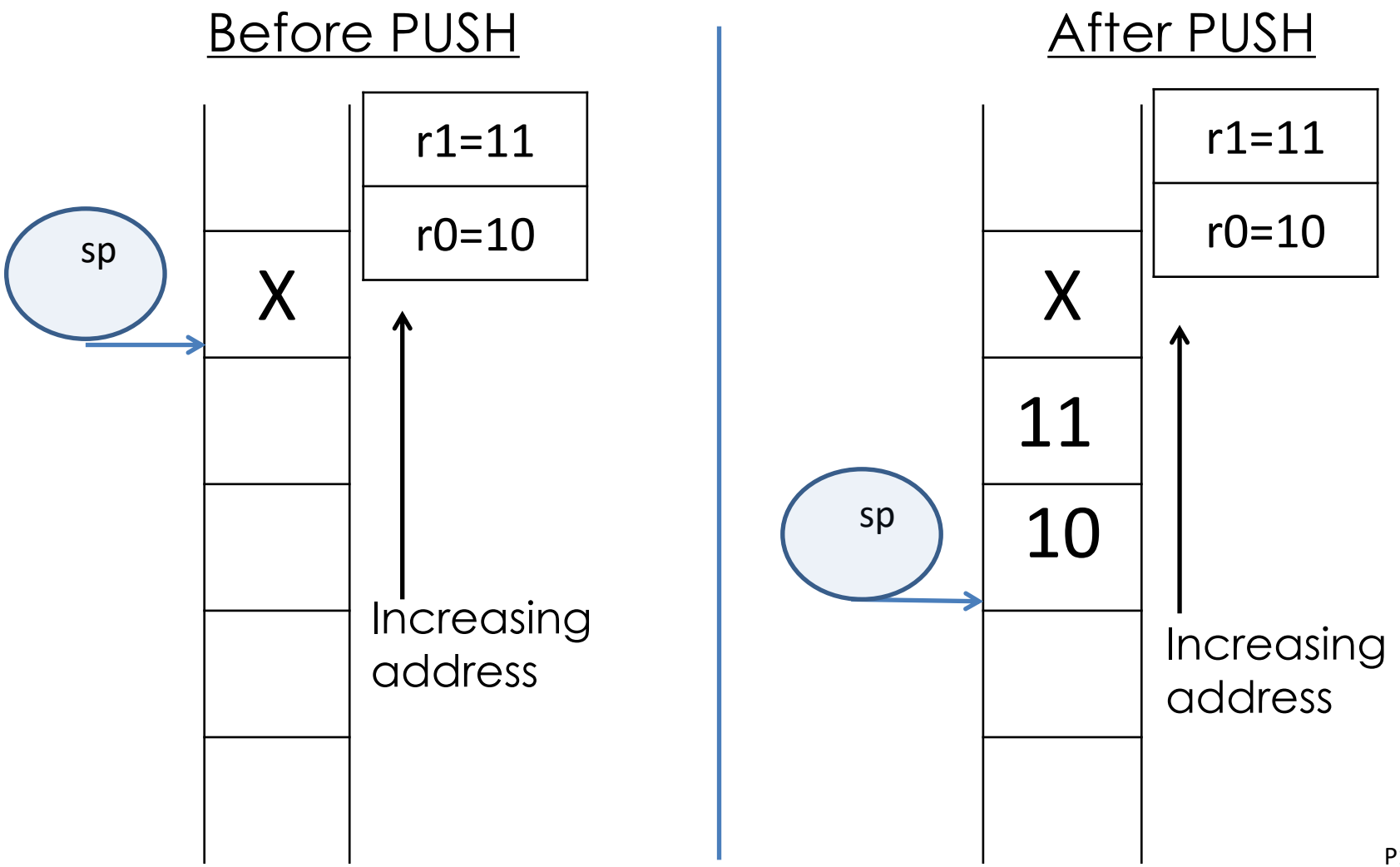
- There are SIX r13 (sp) registers available in ARM, one for each exception mode.
- In other words, we can have SIX separate stacks, one for each exception mode.
- More of these other stacks will be covered later under the exception chapter.

(1) Full Descending (FD) Stack

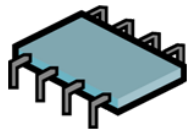
- Descending → the stack grows downward, starting with a high address and progressing to a lower address
- In other words, every times a word is pushed into the stack, it occupies a lower address (-4 bytes), hence descending
- Full → the stack pointer, sp (r13) points to the last pushed word in the stack
- **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

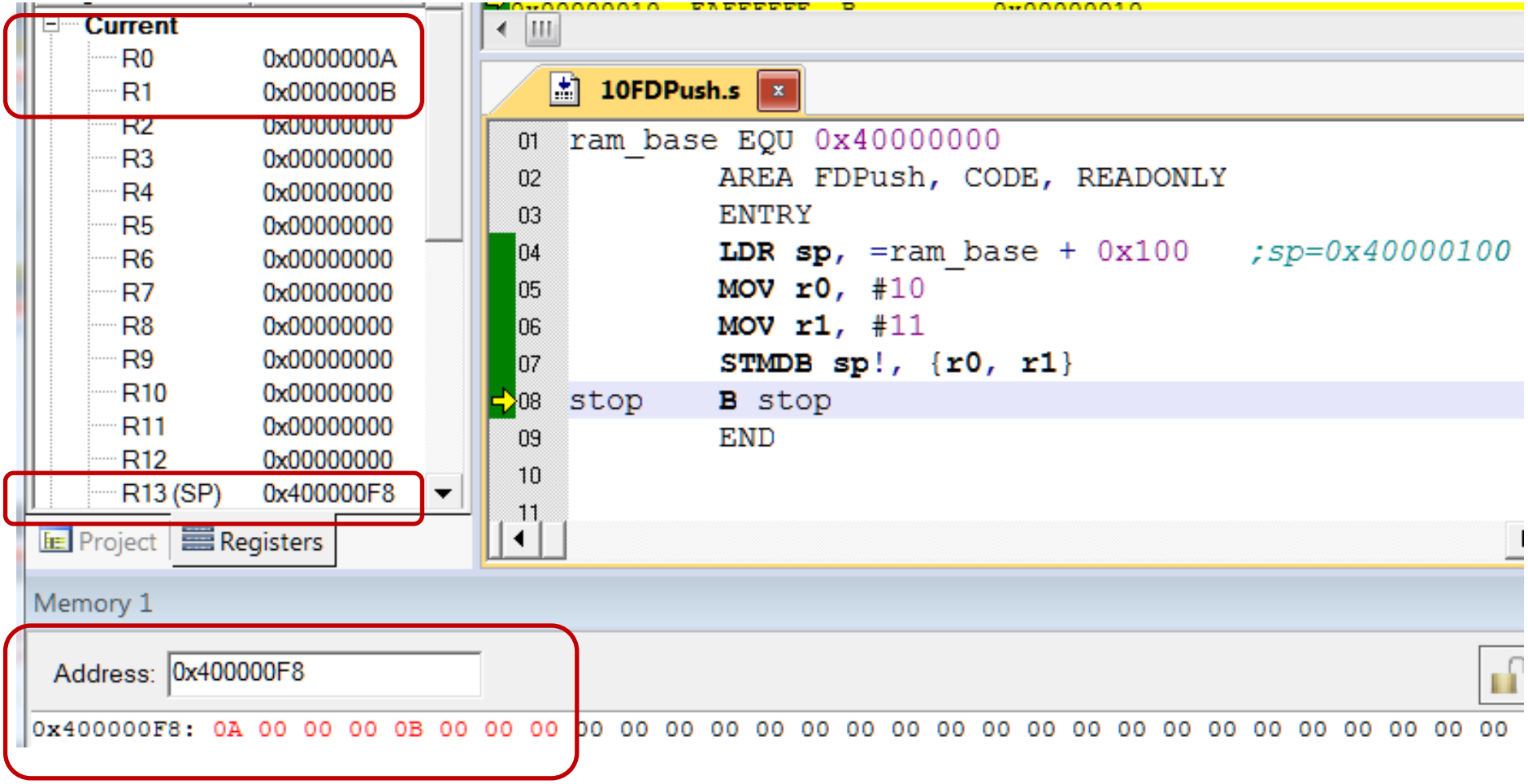
Full Descending Stack → PUSH example

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



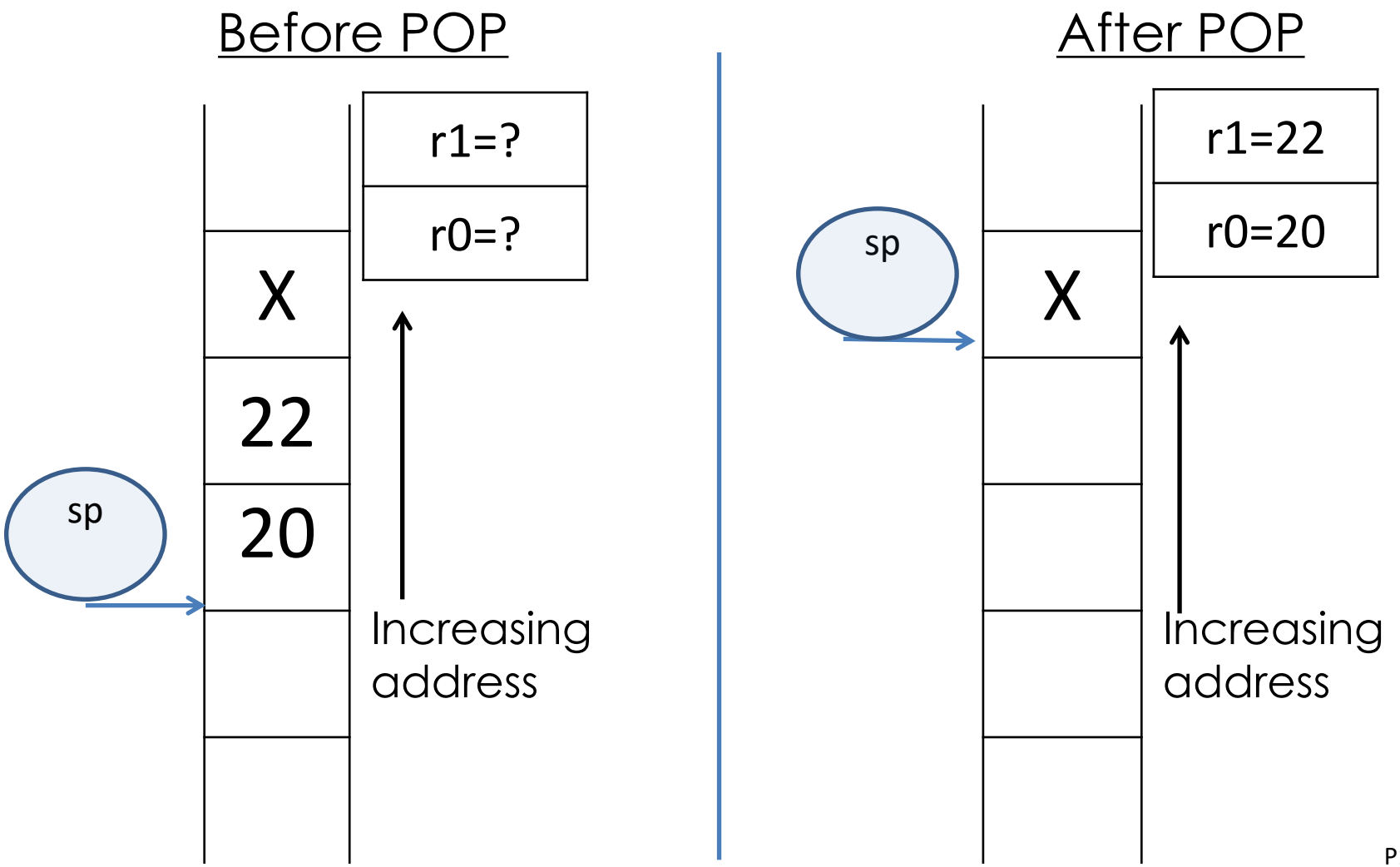
Full Descending Stack → PUSH Demo



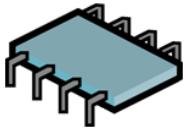


Full Descending Stack → POP Example

- 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

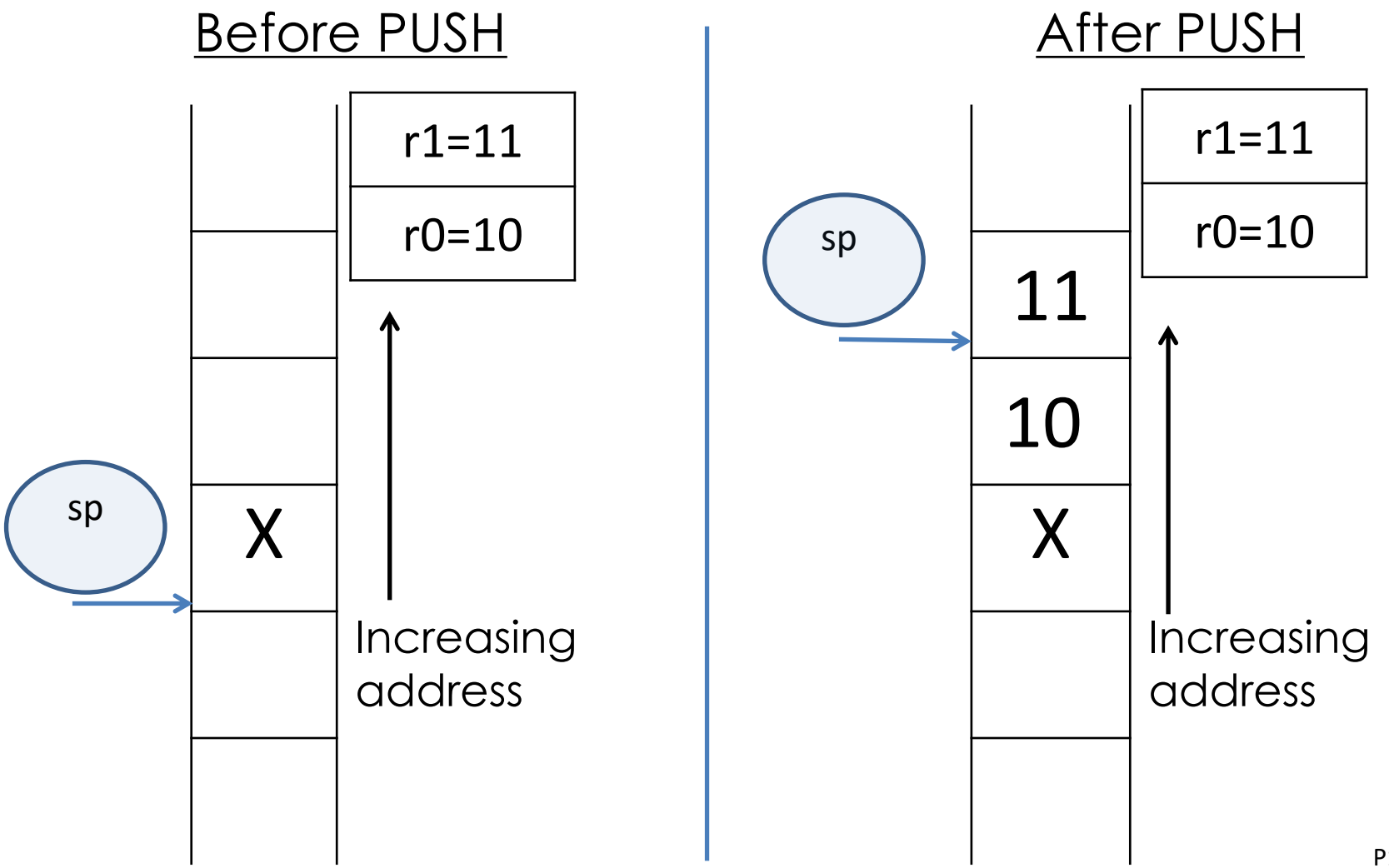
0x40000100: 14 00 00 00 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

(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

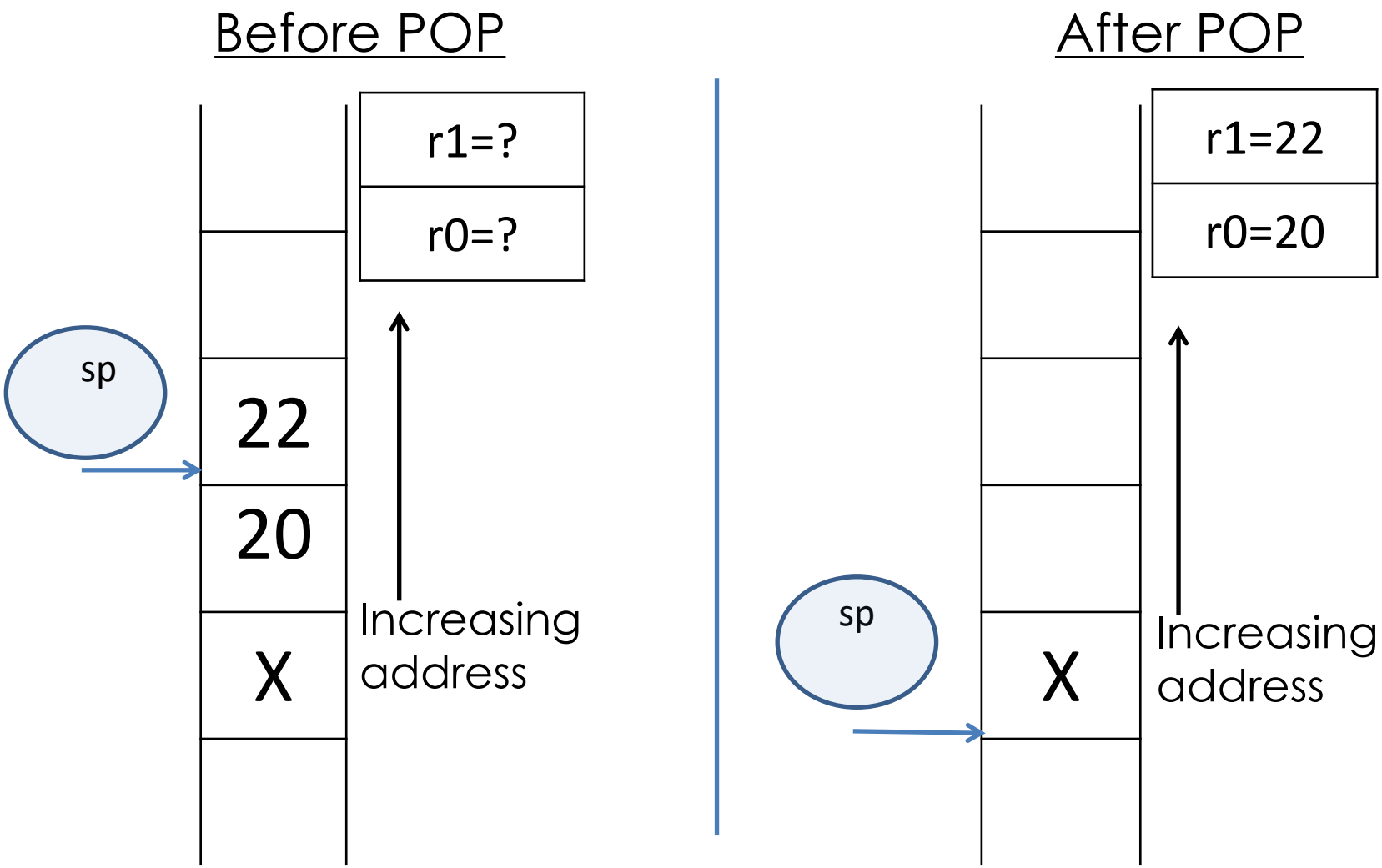
Full Ascending Stack → PUSH Example

- 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}

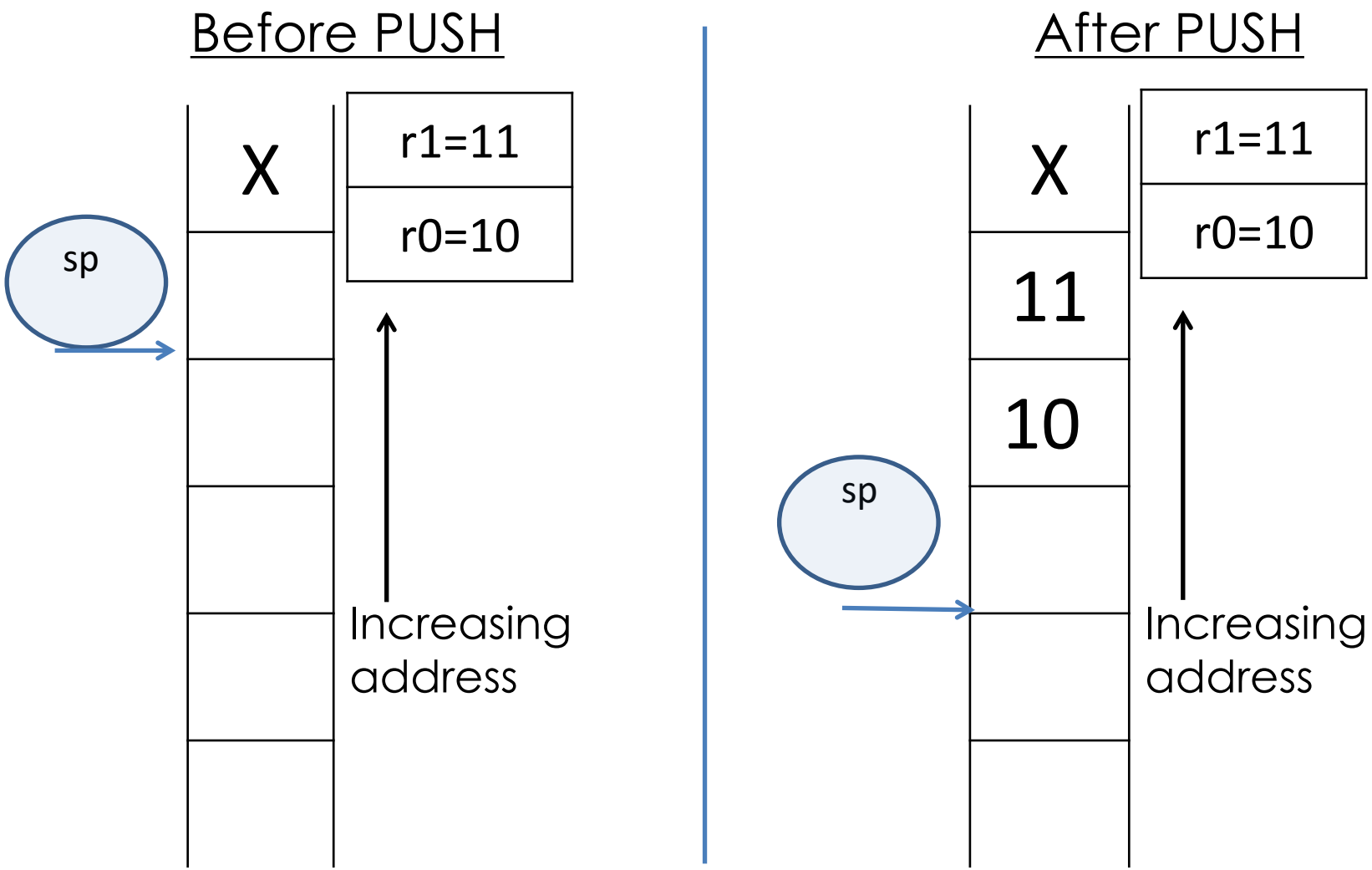


(3) Empty Descending (ED) Stack

- 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

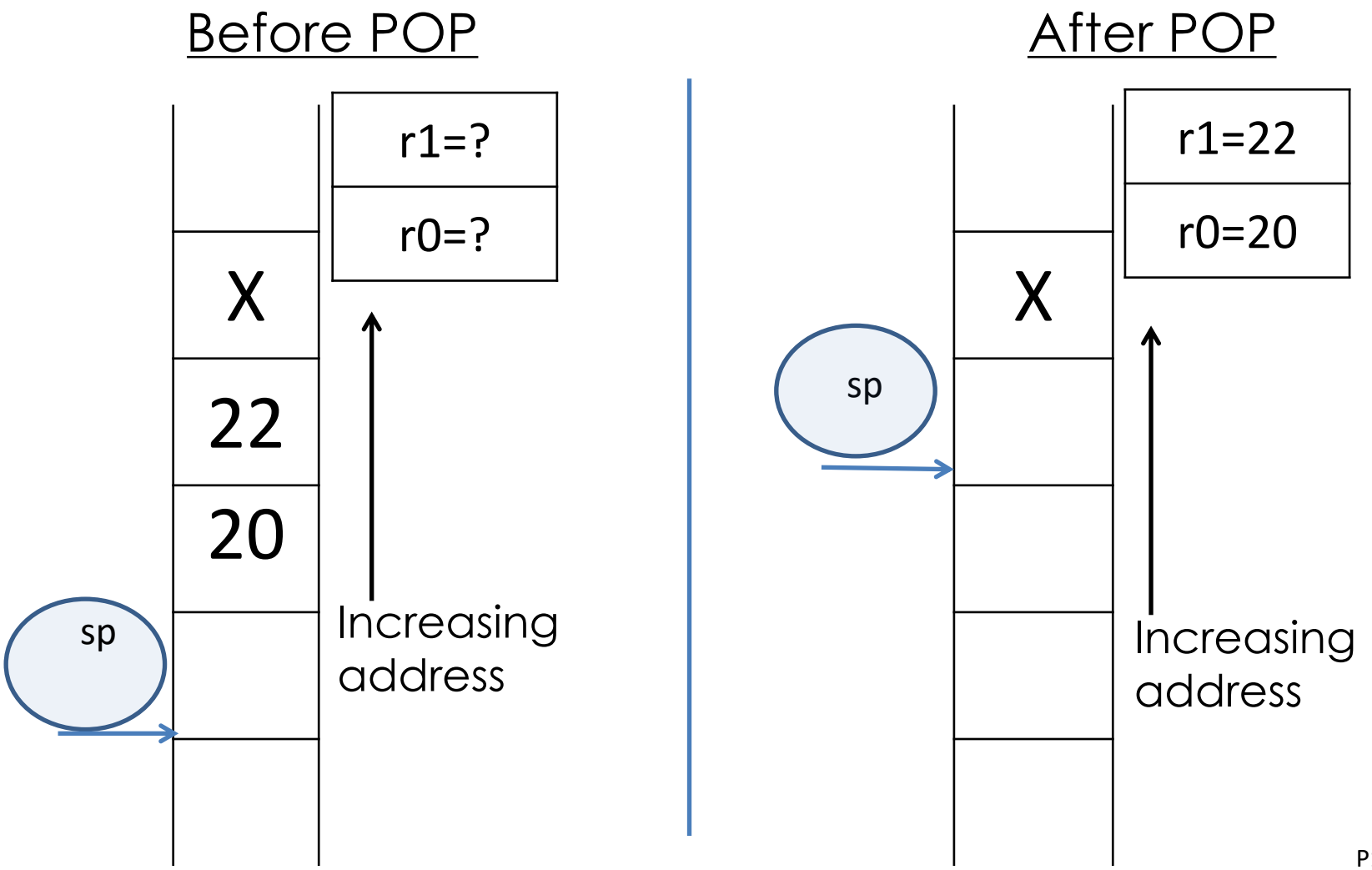
Empty Descending Stack → PUSH Example

- 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}

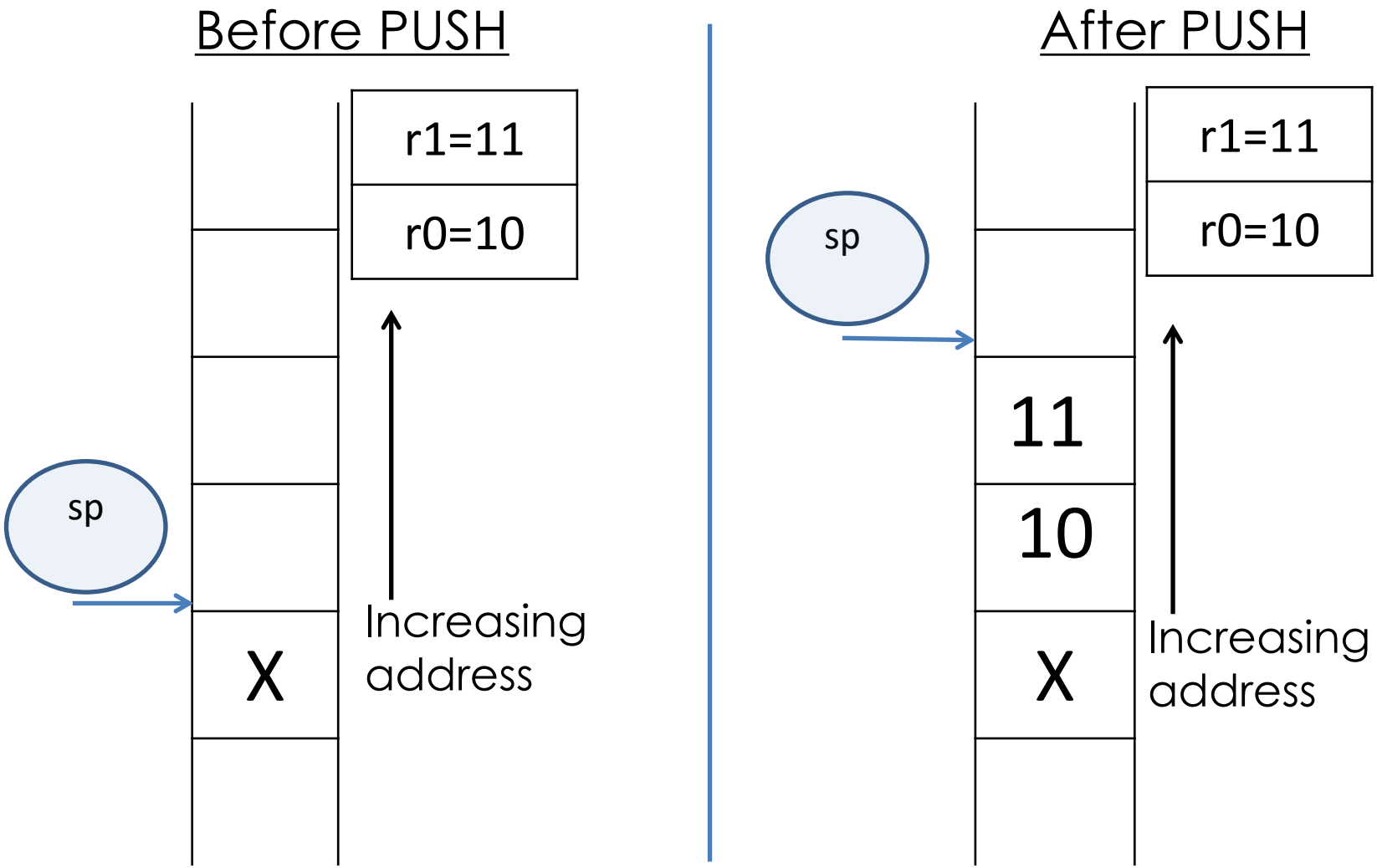


(4) Empty Ascending (EA) Stack

- 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

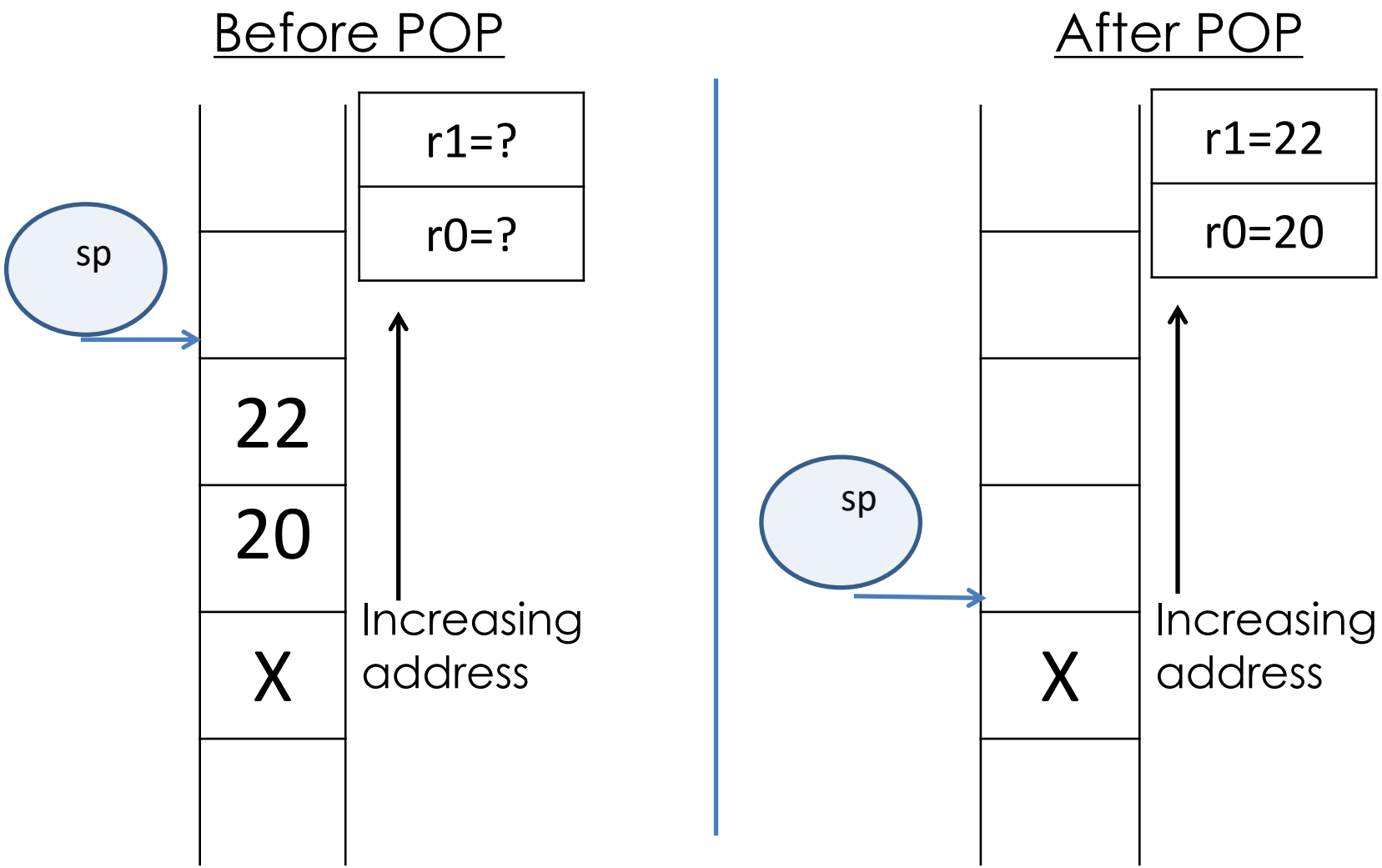
Empty Ascending Stack → PUSH example

- 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)