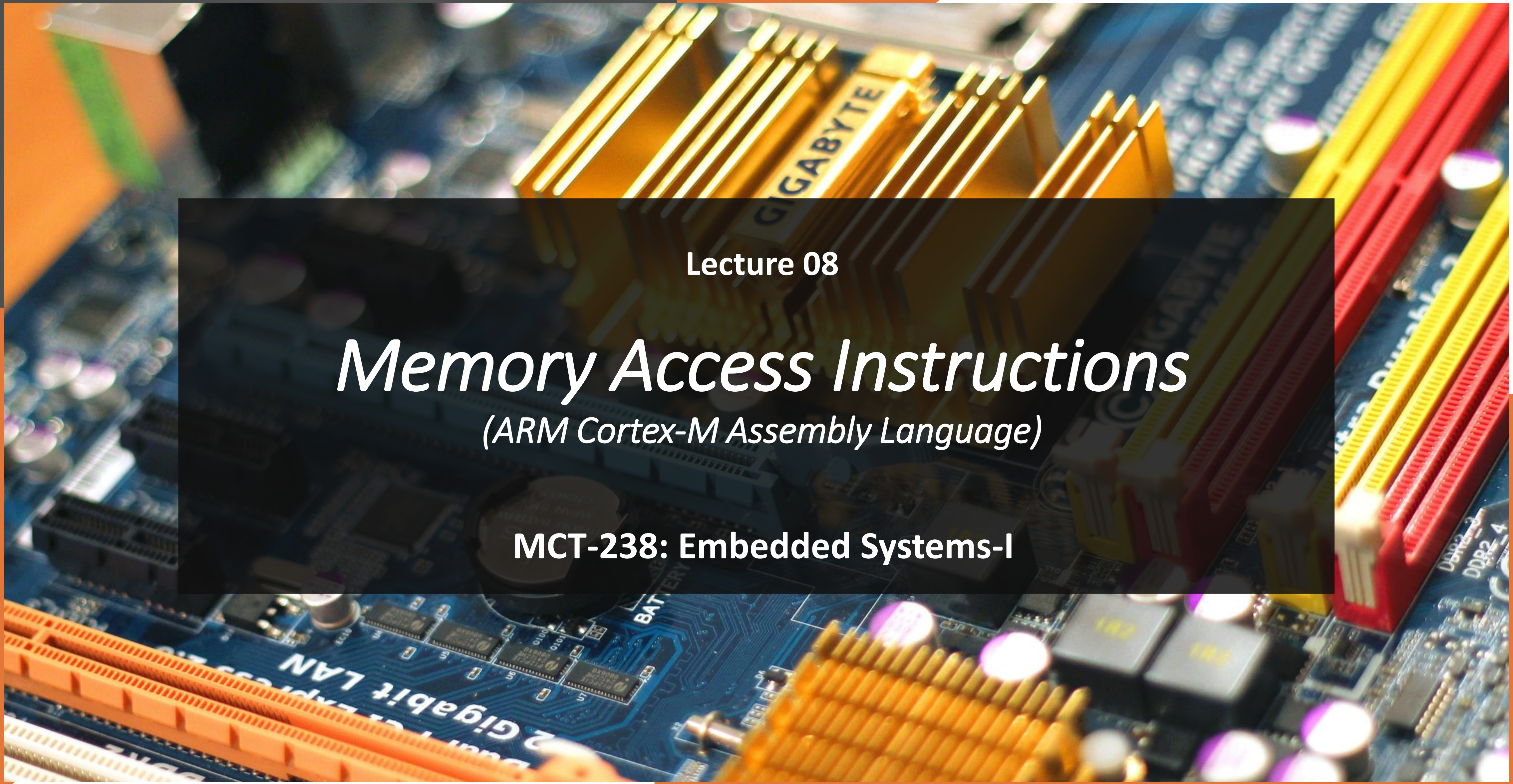
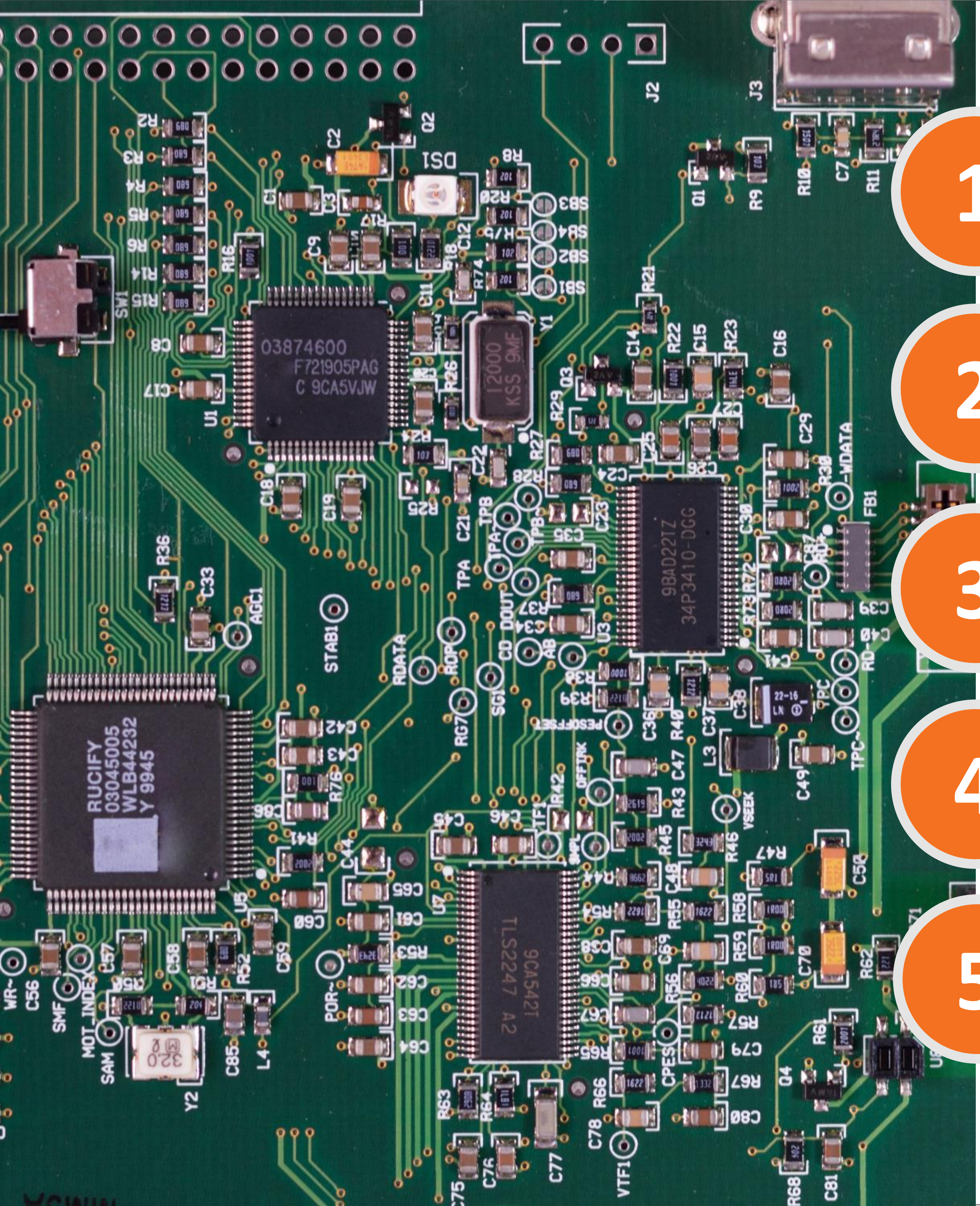


Lecture 08

Memory Access Instructions

(ARM Cortex-M Assembly Language)

MCT-238: Embedded Systems-I



1 WHY WE NEED TO ACCESS MEMORY?

2 LOAD AND STORE INSTRUCTIONS

3 LDR WITH PC-RELATIVE ADDRESSING MODE

4 MULTIPLE WORD MEMORY ACCESSES

5 STACK MEMORY ACCESS

WHY WE NEED TO ACCESS MEMORY?

Fundamentals of Embedded Systems

Fundamentals of Embedded Systems

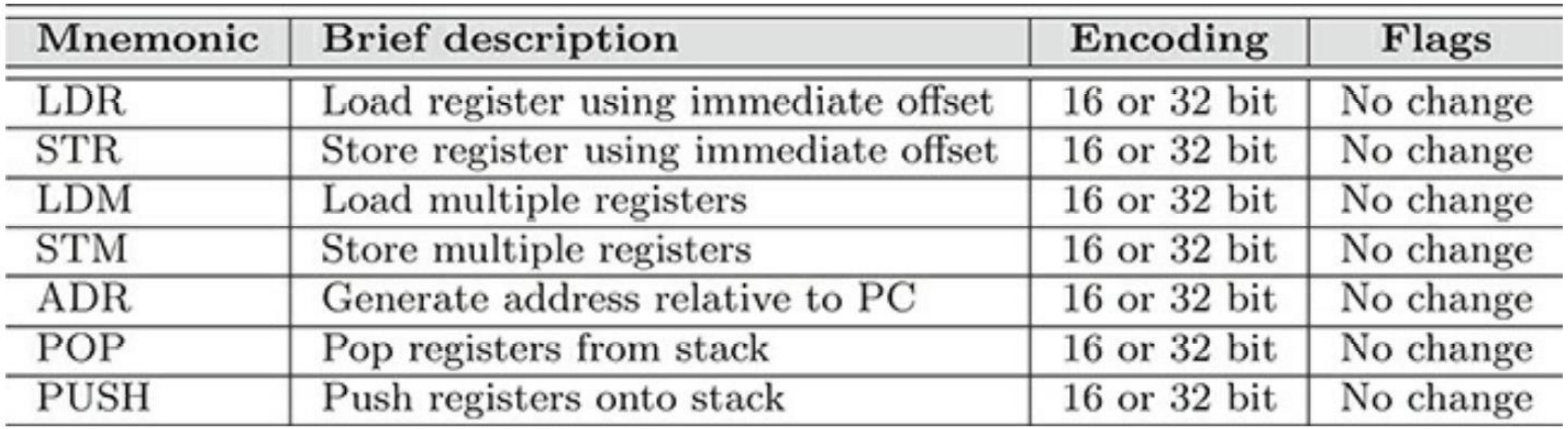
WHY WE NEED TO ACCESS MEMORY?

- In embedded systems, data transfers among the memory processor as well as peripherals is an essential task that is performed quite frequently.
    - **Load-Process-Store Model**: Existing data can be loaded to the memory during the application initialization, which can then be processed by the user program during the associated task execution.
    - **Data Storage**: In another scenario, the raw data can be obtained from a peripheral device by the processor and then can be stored to the memory after processing.
    - **Memory Mapped Peripherals**: In Cortex-M architecture, the peripherals are memory mapped. In case of memory mapped peripherals, data read and write operations are performed like the memory read/write operations. The only distinguishing aspect is that a different address space, based on memory address map, is used to interact with the peripherals.
- These data transfers are carried out using **memory access instructions**

Memory Access Instructions

WHY WE NEED TO ACCESS MEMORY?

- Being a RISC architecture, Cortex-M processors allow only a few instructions for accessing the memory, including load-store and push-pop
- Memory access instructions are crucial as transfer of data occurs frequently between memory and processor in digital systems
- So, we can state that memory access instructions are the heart of the instruction set



Mnemonic	Brief description	Encoding	Flags
LDR	Load register using immediate offset	16 or 32 bit	No change
STR	Store register using immediate offset	16 or 32 bit	No change
LDM	Load multiple registers	16 or 32 bit	No change
STM	Store multiple registers	16 or 32 bit	No change
ADR	Generate address relative to PC	16 or 32 bit	No change
POP	Pop registers from stack	16 or 32 bit	No change
PUSH	Push registers onto stack	16 or 32 bit	No change

LOAD AND STORE INSTRUCTIONS

ARM Cortex-M4 Assembly Language

Offset Addressing

LOAD AND STORE INSTRUCTIONS

- In ARM Cortex-M processor, each memory location is accessed by using **offset addressing** which involve base address and offset.
- **Base Address**: a pointer to the starting address of the memory segment, maintained in one of the processor registers
- **Offset Address** is used to generated a relative address
- When the based address is derived as the current value of program counter register, then this relative addressing is specifically termed **PC-relative addressing** and will be discussed later
- Offset based addressing can be implemented in the following two possible ways:
    - Immediate offset addressing
    - Register offset addressing

Immediate Offset Addressing

LOAD AND STORE INSTRUCTIONS

- **LDR** (load register) instruction is used to **load** 32-bits information from a memory location to CPU register
- **STR** (store register) instruction is used to **store** 32-bits information to a memory location from CPU register
- Address of memory location is provided in the form of offset addressing i.e., base address + offset address
- When the offset is an immediate value from the base address stored in a register, the resulting addressing mode is called **immediate offset addressing**

Immediate Offset Addressing

LOAD AND STORE INSTRUCTIONS

$$\begin{array}{lll} \mathrm{LDR}\{type\}\{cond\} & Rt, & [Rn\ \{,\ \#offset\}] \\ \mathrm{STR}\{type\}\{cond\} & Rt, & [Rn\ \{,\ \#offset\}] \end{array}$$

- **{}** Represents the optional field
- ***type***: *Access Type* Specifier, used to access 8 or 16-bits of data access instead of 32-bits
- ***cond***: *Conditional Code Suffix*, used to make instruction conditional based on status of flags of APSR
- ***Rt***: a CPU register, destination operand for **LDR** and source operand for **STR** instruction
- ***Rn***: a CPU register holding base address of memory location

Immediate Offset Addressing

LOAD AND STORE INSTRUCTIONS

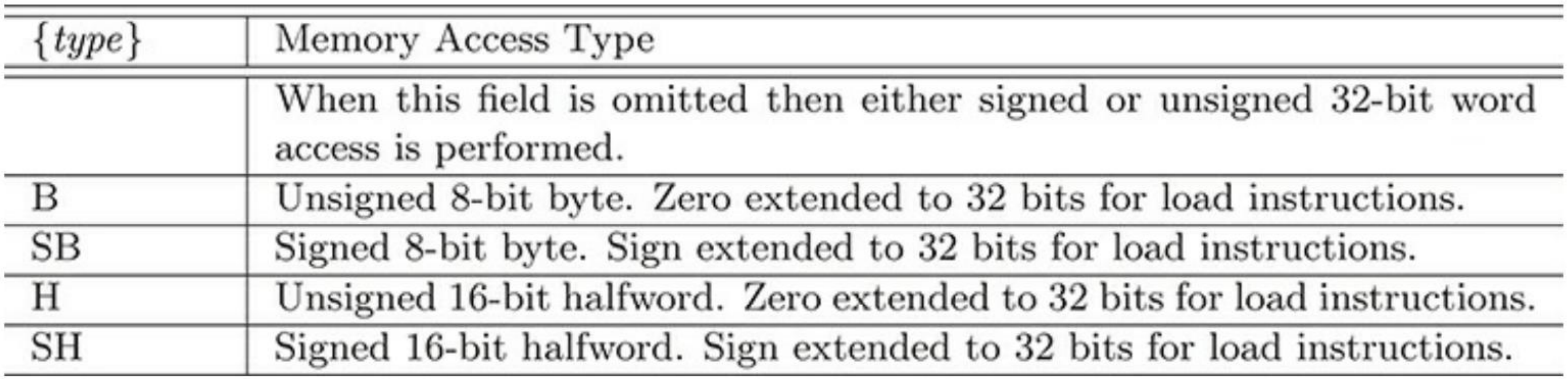
$$\begin{array}{lll} \mathrm{LDR}\{type\}\{cond\} & Rt, & [Rn\ \{,\ \#offset\}] \\ \mathrm{STR}\{type\}\{cond\} & Rt, & [Rn\ \{,\ \#offset\}] \end{array}$$

- The optional **#offset** field is used to specify an offset from the base address register **Rn**.
- The **#offset** field can also be zero in which case the register **Rn** value is used as the memory address.
- The immediate offset can take any value between **-255** and **+255** (for 16-bit instruction encoding) between **-4095** and **+4095** (for 32-bit instruction encoding)
- **Rn** holding base address along with immediate offset value (**#offset**) determines the destination address in the memory. To explicitly identify this special usage of the register, it is surrounded by square brackets **[]**.

Optional Type Specifiers

LOAD AND STORE INSTRUCTIONS

- By default, the **LDR** and **STR** instructions perform 32-bit data transfers, but it is also possible for 8, 16 bits by using optional memory **access type specifier**
- Using the optional type specifier **B** and **SB** allows to perform 8-bit unsigned and signed data transfers, respectively, between the register and memory.
- Similarly, the type specifier **H** and **SH** is used to perform unsigned and signed halfword data transfers.



{*type*}	Memory Access Type
	When this field is omitted then either signed or unsigned 32-bit word access is performed.
B	Unsigned 8-bit byte. Zero extended to 32 bits for load instructions.
SB	Signed 8-bit byte. Sign extended to 32 bits for load instructions.
H	Unsigned 16-bit halfword. Zero extended to 32 bits for load instructions.
SH	Signed 16-bit halfword. Sign extended to 32 bits for load instructions.

Optional Type Specifiers

LOAD AND STORE INSTRUCTIONS

ZERO/SIGN EXTENSION

- When 8-bit or 16-bit data is transferred from memory to CPU register (which is always 32- bit), we need to pay attention of the sign while extending the data size to 32 bits; otherwise, data will be corrupted
- When an 8- or 16-bit unsigned value is loaded to a register, the most significant bits of the register are filled with 0s. This is termed **zero extending** the data.
  - If we load an unsigned number, say 6 (or 0x06) to a 32-bit register, then zero extended 32-bit value will result in 0x00000006
- On the other hand, when we load a register with 8- or 16-bit signed value, the sign bit is extended to fill the most significant bits. This is called **sign extension**.
  - When an 8-bit number, say -6 (or 0xFA) is loaded to a 32-bit register, the number needs to be sign extended. The sign extension will yield 0xFFFFFFFA in the register, which is still -6 in decimal
- Transferring 32 bits or its multiples between memory and processor registers, however, does not raise any issues regarding data size extension
- On the other hand, no extension is required when an 8-bit or 16-bit value is stored from the register to the memory. But why?

Addressing Modes

LOAD AND STORE INSTRUCTIONS – Immediate Offset Addressing

- It should be noted that since the data memory, code memory as well as the peripherals are the part of the memory map for Cortex-M, hence the address register in LDR instruction can be used for pointing toward any object in a valid memory region of the memory map.
- In Thumb2 instruction set architecture, the following two different variants of this addressing mode are defined, which will be discussed next.
    - Pre-indexed immediate offset addressing
    - Post-indexed immediate offset addressing

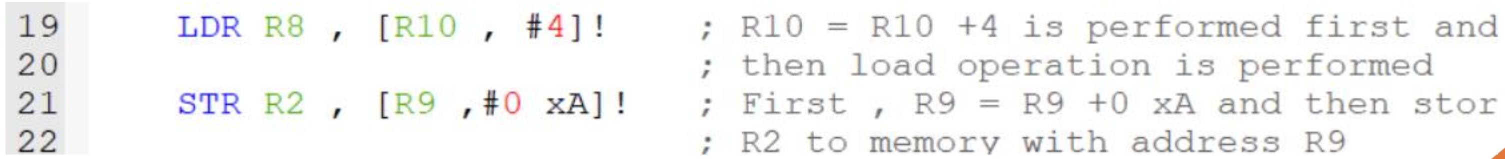
Pre-Indexed Offset Addressing Mode

LOAD AND STORE INSTRUCTIONS – Immediate Offset Addressing

- In this addressing mode, the register containing the base address is updated first and then used by the LDR or STR instruction

$$\begin{array}{lll} \mathrm{LDR}\{type\}\{cond\} & Rt, & [Rn,\ \#offset]! \\ \mathrm{STR}\{type\}\{cond\} & Rt, & [Rn,\ \#offset]! \end{array}$$

- In this addressing mode, the base address is updated unlike the previous addressing mode, where the register holding the base address is not changed during instruction execution
- The presence of '!' after the closing or right square bracket ']' indicates that the contents of the register containing the base address are permanently updated and distinguishes the pre-indexed addressing mode from the immediate addressing mode.
- It should be noted that #offset is a mandatory field for the pre-indexed offset addressing mode



```
LDR R8 , [R10 , #4]!      ; R10 = R10 +4 is performed first and
                          ; then load operation is performed
STR R2 , [R9 ,#0 xA]!     ; First , R9 = R9 +0 xA and then stor
                          ; R2 to memory with address R9
```

Post-Indexed Offset Addressing Mode

LOAD AND STORE INSTRUCTIONS – Immediate Offset Addressing

- If the base address contained in the address register is updated after the load or store operation has been performed, then this addressing mode is called the post-index offset addressing mode.

$$\begin{array}{lll} \mathrm{LDR}\{type\}\{cond\} & Rt, & [Rn],\ \#offset \\ \mathrm{STR}\{type\}\{cond\} & Rt, & [Rn],\ \#offset \end{array}$$

```
        STR R1 , [R3], #5    ; STR is performed with R3 and then
                             ; R3 is updated as R3 = R3 +5.
        LDR R0 , [R6], #4    ; First perform LDR with R6 and then
                             ; update R6 as R6 = R6 +4.
```

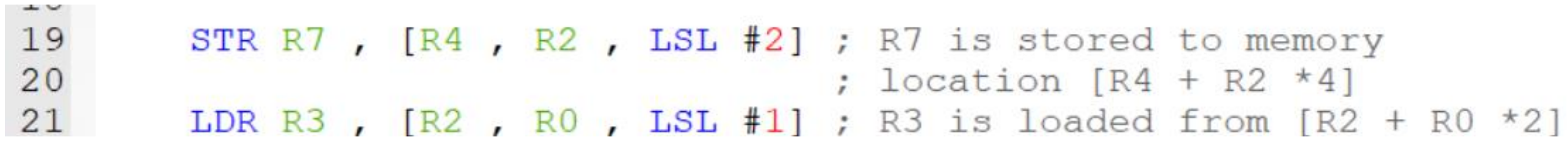
Register Offset Addressing

LOAD AND STORE INSTRUCTIONS

- In this addressing mode, the base address is contained in a register and unlike the immediate offset addressing, the offset value is also in a register

$$\begin{array}{lll}\text{LDR}\{\mathit{type}\}\{\mathit{cond}\} & \mathit{Rt}, & [\mathit{Rn},\ \mathit{Rm}\ \{,\ \mathit{LSL}\ \#n\}] \\ \text{STR}\{\mathit{type}\}\{\mathit{cond}\} & \mathit{Rt}, & [\mathit{Rn},\ \mathit{Rm}\ \{,\ \mathit{LSL}\ \#n\}]\end{array}$$

- **Rn** contains the base address of the memory, to or from which an offset contained in **Rm** is added or subtracted
- The offset can be shifted by up to 3 bits by left shift logical operation, i.e., The {, **LSL#n**} field is optional where the parameter **n** can take values in the range 0 to 3
- The registers **Rn** and **Rm** preserve their original values after the instruction execution



```
        STR R7 , [R4 , R2 , LSL #2] ; R7 is stored to memory
                                    ; location [R4 + R2 *4]
        LDR R3 , [R2 , R0 , LSL #1] ; R3 is loaded from [R2 + R0 *2]
```

Aligned/Unaligned Memory Accesses

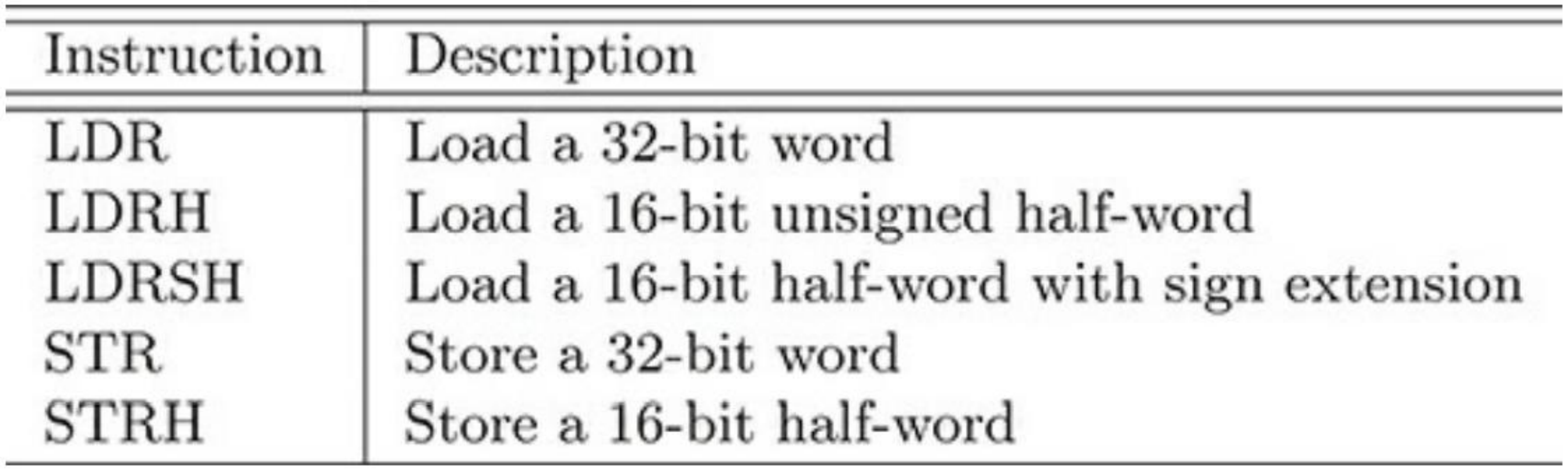
LOAD AND STORE INSTRUCTIONS

- For Cortex-M processor, memory accesses can be of 8, 16, 32 or multiples of 32 bits. The starting memory address of a data element is crucial, which can result in aligned or unaligned access.
- A 32-bit word object will result in an aligned word access if the first two bits of its starting address are zero or if it is divisible by 4. Otherwise, the 32-bit data access is unaligned.
- Similarly, if the starting address of a 16-bit halfword object is divisible by 2 or its LSB is zero, we say the 16-bit object is half-word aligned. Otherwise, its access will be unaligned.
- Memory accesses of size 1 byte are, however always aligned.
- Usually aligned memory accesses are faster than the unaligned accesses. But memory is better utilized with unaligned access.
- As a result, there is a tradeoff between memory size required for certain data and the speed at which this data can be accessed by the processor.

Aligned/Unaligned Memory Accesses

LOAD AND STORE INSTRUCTIONS

- In Cortex-M processor, not all the read and write memory instructions can perform unaligned access operations. Instructions, which support unaligned access are given below.
- We can observe that no sign extension is required when storing a 16-bit signed halfword to the memory. Therefore, unlike LDRH and LDRSH, only STRH is available.



Instruction	Description
LDR	Load a 32-bit word
LDRH	Load a 16-bit unsigned half-word
LDRSH	Load a 16-bit half-word with sign extension
STR	Store a 32-bit word
STRH	Store a 16-bit half-word

Aligned/Unaligned Memory Accesses

LOAD AND STORE INSTRUCTIONS

- The set of those assembly instructions that are used for data transfers of size 1 byte.

Instruction	Description
LDRB	Load 8-bit unsigned byte
LDRSB	Load 8-bit signed byte (sign extend bit 7 over bit 8 to 31 positions)
STRB	Store 8-bit byte

- If the address values in registers R3 and R4 (for second and fourth instructions below)
  - are odd numbers, then these instructions result in an aligned access.
  - are even numbers, then these instructions result in an unaligned access.

```
LDRB R8, [R5, #12]  ; no problem at all with byte transfers
LDRH R0, [R3, #3]   ; LDRH can be unaligned access
STRB R0, [R1, #1]   ; no problem at all with byte transfers
STRH R0, [R4, #3]   ; STRH can be unaligned access
```

Unprivileged Load/Store Instructions

LOAD AND STORE INSTRUCTIONS

- Cortex-M processor can access the memory either in privileged or unprivileged mode.
- Load and store instructions with unprivileged access can be used to mandate the applications running on top of the operating system to have only unprivileged access.
- Unprivileged access only permits a restricted memory access to the application program.

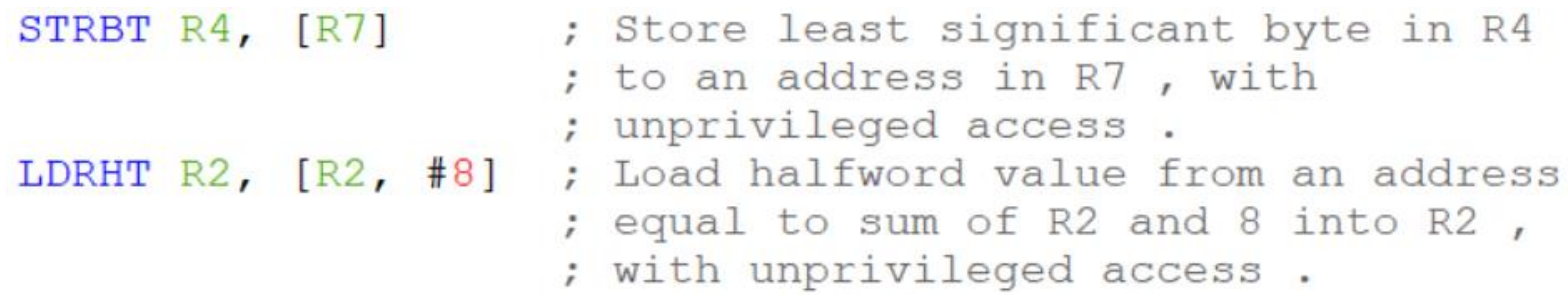
$$\begin{array}{lll}\mathrm{LDR}\{type\}\mathrm{T}\{cond\} & Rt, & [Rn\ \{,\#offset\}] \\ \mathrm{STR}\{type\}\mathrm{T}\{cond\} & Rt, & [Rn\ \{,\#offset\}]\end{array}$$

- These instructions perform the same function as the memory access instructions with immediate offset

Unprivileged Load/Store Instructions

LOAD AND STORE INSTRUCTIONS

- Even when these instructions are part of a software with privileged access rights, these remain unprivileged
- And when used in unprivileged software, these instructions behave as normal memory access instructions with immediate offset



```
STRBT R4, [R7]        ; Store least significant byte in R4
                      ; to an address in R7 , with
                      ; unprivileged access .
LDRHT R2, [R2, #8]    ; Load halfword value from an address
                      ; equal to sum of R2 and 8 into R2 ,
                      ; with unprivileged access .
```

LDR WITH PC-RELATIVE ADDRESSING MODE

ARM Cortex-M4 Assembly Language

What is PC-Relative Addressing

LDR WITH PC-RELATIVE ADDRESSING MODE

- In PC-relative addressing mode, the memory address is generated using an offset from the current value of PC.
- PC-relative memory addresses can be specified using labels, which are also known as **PC-relative expressions**.
- A label is used for representing an instruction address or literal data in code memory region.
- General syntax of LDR with PC-relative addressing mode is given below.

$$\mathrm{LDR}\{type\}\{cond\} \quad Rt, \quad label$$

What is PC-Relative Addressing

LDR WITH PC-RELATIVE ADDRESSING MODE

$$\mathrm{LDR}\{type\}\{cond\} \quad Rt, \quad label$$

- The LDR instruction, using the above-mentioned syntax, is called **pseudo assembly** instruction. As it is converted to an equivalent PC-relative addressing based assembly instruction, during the compilation process
- When assembled, the assembler calculates the label address offset with respect to this instruction location (i.e., current value of PC)
- It is required that the label must be within an address offset of ±4095 from the current instruction address. Otherwise, the assembler generates an error.
- The assembler may also permit us to directly write the label as [PC, #number]. The expression [PC, #number] generates the label address by adding or subtracting an offset equal to #number to the current value of PC.

How does this Addressing Mode Work?

LDR WITH PC-RELATIVE ADDRESSING MODE

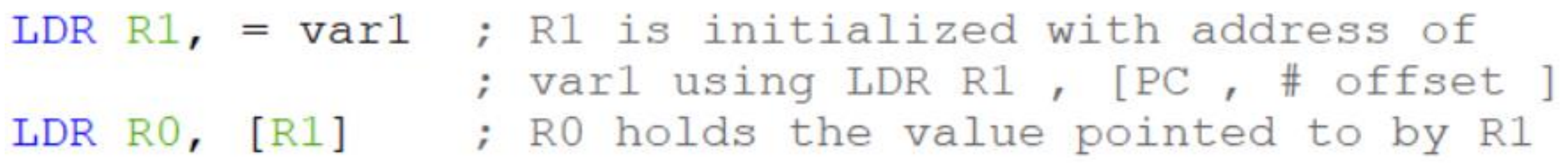
- Let’s assume that var1 is a 32-bit data object declared in an area with READWRITE attribute while var2 with two elements, each of 32 bits, is declared in an area with READONLY attribute, then the following instructions are valid.

```
LDR R0, = var1        ; loads the address of var1 in R0
LDR R7, var2          ; loads R7 with first element of var2
LDR R4, = var2        ; loads R4 with the address of var2
LDR R3, var2 + 4      ; loads R3 with second element of var2
```

- In the above example, reading data from memory with READONLY attribute (or ROM) space can be done directly with an instruction such as ***LDR R7, var2***, provided the data (var2 in this case) is within an offset of ±4095 from the address of the corresponding instruction.
- However, loading registers with the data from memory with READWRITE (RAM) attribute or I/O will in general violate the offset value restriction and requires two instructions for data loading (explained on next slide)

How does this Addressing Mode Work?

LDR WITH PC-RELATIVE ADDRESSING MODE



```
LDR R1, = var1   ; R1 is initialized with address of
                 ; var1 using LDR R1 , [PC , # offset ]
LDR R0, [R1]     ; R0 holds the value pointed to by R1
```

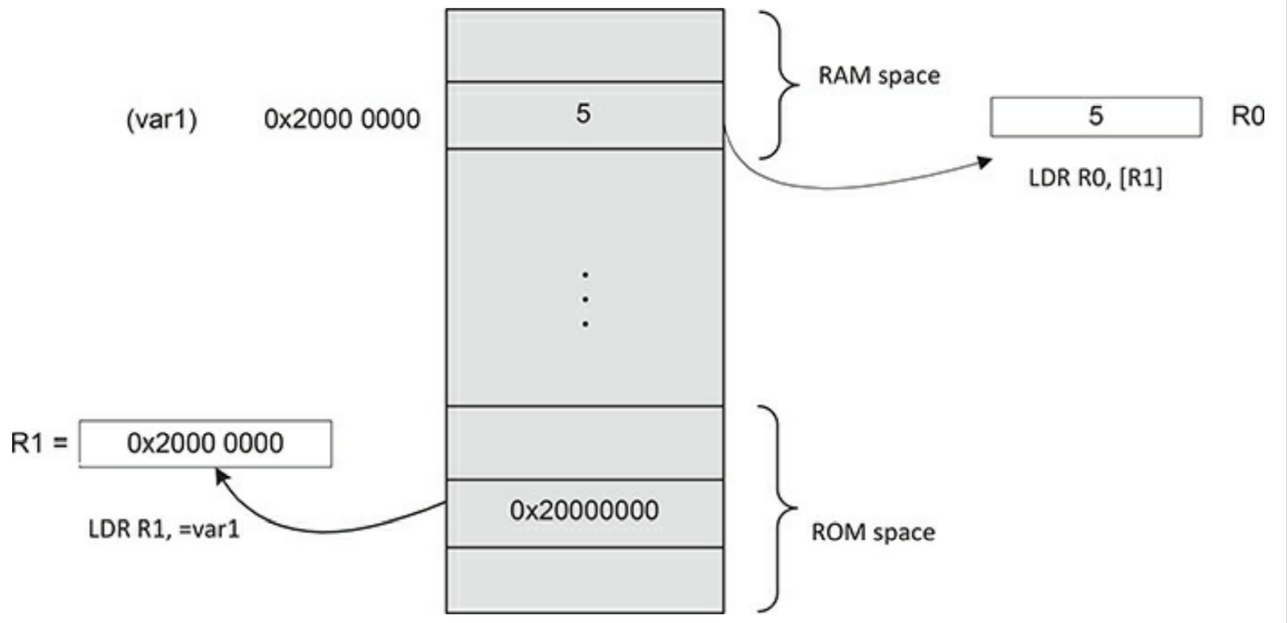
- The first instruction uses PC-relative addressing and loads register **R1** with the address of the variable named var1.
- The 32-bit address, loaded to **R1**, is physically stored in the code memory region at an address obtained by adding the offset to the current value of PC.
- This address in **R1** is used in the second instruction to read the current value of var1 from the RAM memory.

How does this Addressing Mode Work?

LDR WITH PC-RELATIVE ADDRESSING MODE

- Let’s assume that var1 is 32 bit, with current value 5 and is in the RAM region at address 0x20000000.
- First, we initialize **R1** with the address of var1 using LDR **R1**, =var1.
- The pseudo instruction LDR **R1**, =var1 is replaced by the corresponding assembly instruction LDR **R1**, [PC, #offset].
- During this process, the address of var1, (i.e., 0x20000000) will be stored in the form of literal data at an address obtained as PC+offset in the code memory region.

```
LDR R1, = var1   ; R1 is initialized with address of
                 ; var1 using LDR R1 , [PC , # offset ]
LDR R0, [R1]     ; R0 holds the value pointed to by R1
```



Generating 32-Bit Constants with LDR Instruction

LDR WITH PC-RELATIVE ADDRESSING MODE

- How can we generate a 32-bit constant value by using MOV instructions?
- MOV instructions (i.e., MOV, MVN, MOVW, MOVT) generate only 16-bit constant value and for generating a 32-bit immediately value we may use MOV and MOVT instructions in sequence
- The **LDR Rd, =const** is a pseudo assembly instruction that can be used to construct any arbitrary 32-bit constant value in single instruction, which are out of range of the MOV and MVN instructions.
- The LDR pseudo-instruction is the most efficient single instruction to generate an arbitrary 32-bit immediate value.

Generating 32-Bit Constants with LDR Instruction

LDR WITH PC-RELATIVE ADDRESSING MODE

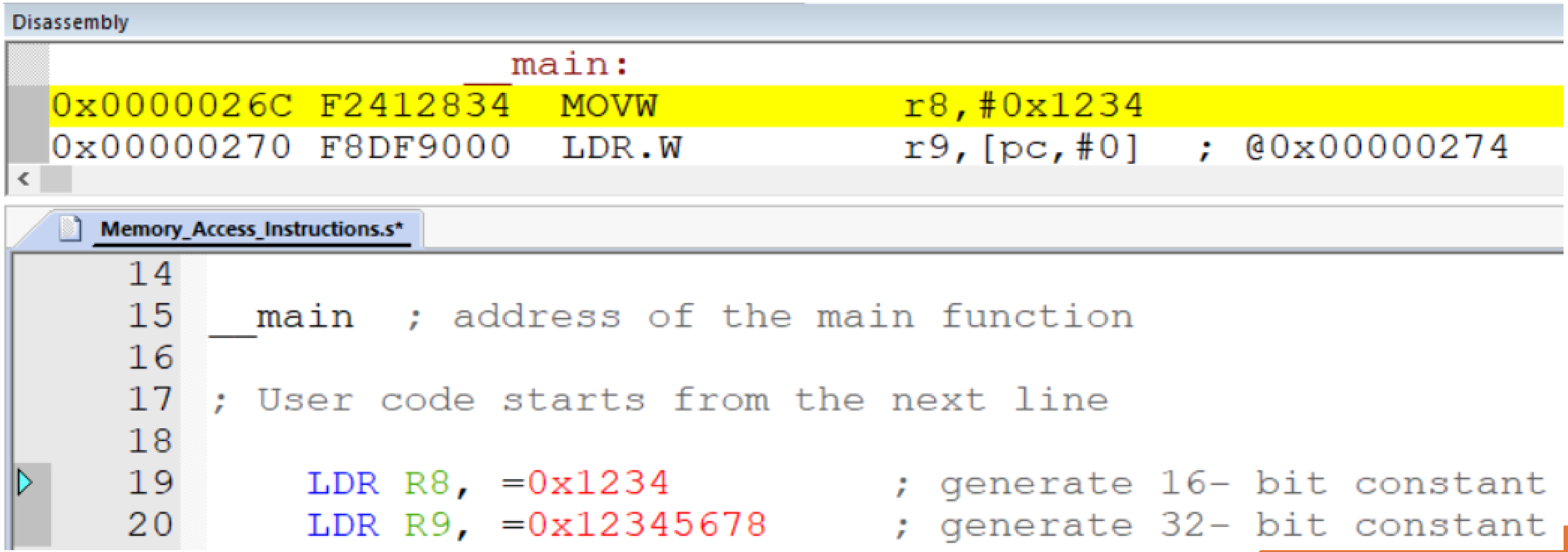
The LDR pseudo instruction can be converted to two different assembly instructions depending on the value of the immediate constant

1. If the immediate value can be constructed using a single MOV or MVN instruction, the assembler is responsible for generating the appropriate instruction.
2. If the immediate value cannot be constructed using single MOV or MVN instruction, the assembler:
   - Places the value in a literal pool (which is a special region in the code memory and is responsible to hold constant values).
   - Generates an LDR instruction with a PC-relative address that reads the constant from the literal pool.

Generating 32-Bit Constants with LDR Instruction

LDR WITH PC-RELATIVE ADDRESSING MODE

- It must be ensured that there is a literal pool within range of the LDR instruction generated by the assembler.
- **LTORG** assembler directive can be used to create a literal pool within range when the code section becomes too large.



Disassembly

```
                    __main:
0x0000026C F2412834  MOVW          r8,#0x1234
0x00000270 F8DF9000  LDR.W         r9,[pc,#0]   ; @0x00000274
```

Memory_Access_Instructions.s*

```

__main  ; address of the main function

; User code starts from the next line

    LDR R8, =0x1234              ; generate 16- bit constant
    LDR R9, =0x12345678          ; generate 32- bit constant
```

MULTIPLE WORD MEMORY ACCESSES

ARM Cortex-M4 Assembly Language

Accessing More than One Word

MULTIPLE WORD MEMORY ACCESSES

- A byte, halfword, or word size of data can be accessed on memory of a Cortex-M by using simple load and store instructions
- In case memory data access becomes larger than one word, two solutions can arise.
  1. **Single Load/Store Instruction**: Use the existing word size memory access instruction repeatedly
  2. **Double or Multiple Load/Store Instruction**: But a more efficient solution allows either double or multiple load and store operations to be performed in single memory access instruction

Double Word Load and Store Instructions

MULTIPLE WORD MEMORY ACCESSES

- These instructions allow memory accesses for two data words.
- The general syntax for load double word instructions with immediate offset, pre-index offset, post-index offset, and pc-relative offset, respectively, is given

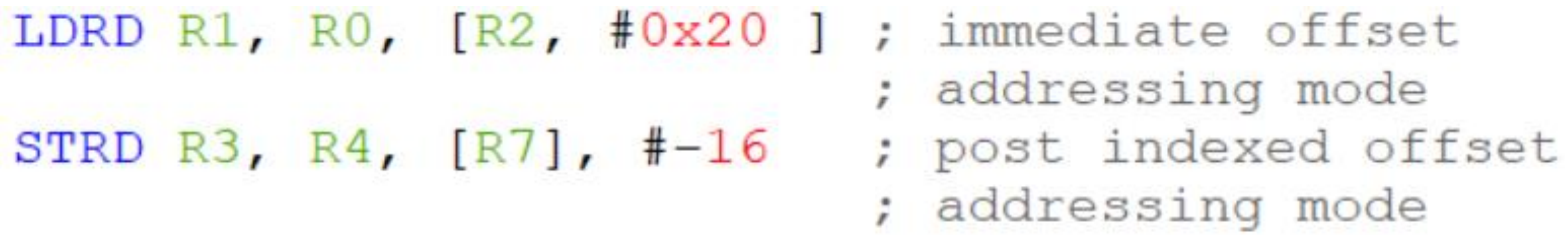
$$\begin{array}{llll}
\mathrm{LDRD}\{cond\} & Rt1, & Rt2, & [Rn\ \{,\ \#offset\}] \\
\mathrm{LDRD}\{cond\} & Rt1, & Rt2, & [Rn,\ \#offset]! \\
\mathrm{LDRD}\{cond\} & Rt1, & Rt2, & [Rn],\ \#offset \\
\mathrm{LDRD}\{cond\} & Rt1, & Rt2, & label
\end{array}$$

- The above syntax is equally applicable for store double word instructions by replacing LDRD with STRD.
- Rt1 and Rt2 are two destination registers for load instruction and these registers will become the source registers for store instruction

Double Word Load and Store Instructions

MULTIPLE WORD MEMORY ACCESSES

- The first instruction loads register R1 with a word at an address obtained by adding an offset of 0x20 to the value in register R2 and loads a word to R0 from an offset of 0x24
- The second instruction stores R3 to an address in R7, and the register R4 to an address equal to R7+4 and then decrements R7 by 16.



```
LDRD R1, R0, [R2, #0x20 ] ; immediate offset
                          ; addressing mode
STRD R3, R4, [R7], #-16   ; post indexed offset
                          ; addressing mode
```

General Syntax of LDM/STM

MULTIPLE WORD MEMORY ACCESSES

- Multiple registers can be loaded or stored using assembly instructions, load multiple registers (LDM) and store multiple registers (STM), respectively

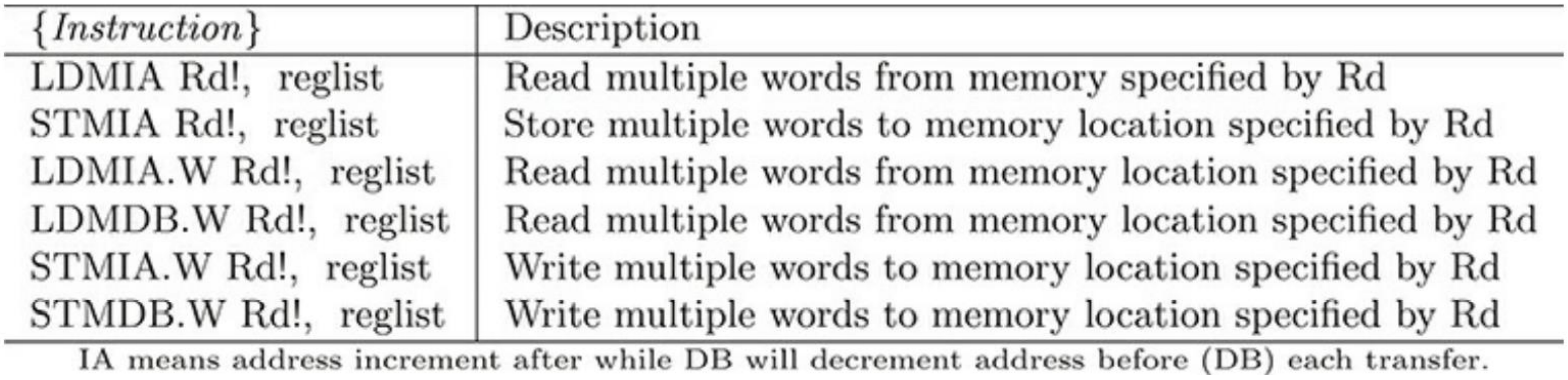
$$\begin{array}{lll} \text{LDM}\{addrmode\}\{cond\} & Rd\{!\}, & reglist \\ \text{STM}\{addrmode\}\{cond\} & Rd\{!\}, & reglist \end{array}$$

- The optional suffix **{addrmode}** can be either address increment after (IA) each access or address decrement before (DB) each access. The former is the default.
- The **{cond}** is an optional condition code suffixes
- The register **Rd** holds the memory base address, from where multiple registers are either loaded from or stored to.
- The **reglist** is the non-empty list of one or more registers to be loaded or stored. Register ranges can also be described for this operand. In case there are more than one register or register range, the registers should be separated by commas.
- Exclamation mark **{!}** field is optional and specifies whether the address register **Rd** should be updated after the instruction has been executed

Possible Combinations

MULTIPLE WORD MEMORY ACCESSES

- List of different combinations that can be obtained from the LDM/STM



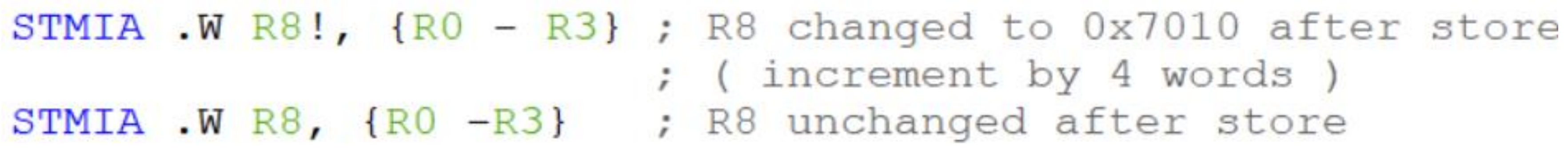
{*Instruction*}	Description
LDMIA Rd!, reglist	Read multiple words from memory specified by Rd
STMIA Rd!, reglist	Store multiple words to memory location specified by Rd
LDMIA.W Rd!, reglist	Read multiple words from memory location specified by Rd
LDMDB.W Rd!, reglist	Read multiple words from memory location specified by Rd
STMIA.W Rd!, reglist	Write multiple words to memory location specified by Rd
STMDB.W Rd!, reglist	Write multiple words to memory location specified by Rd

IA means address increment after while DB will decrement address before (DB) each transfer.

Working of ! sign

MULTIPLE WORD MEMORY ACCESSES

- The following example shows the working of STMIA with and without {!}. Let’s assume that the register **R8** equals 0x7000 as the base address.



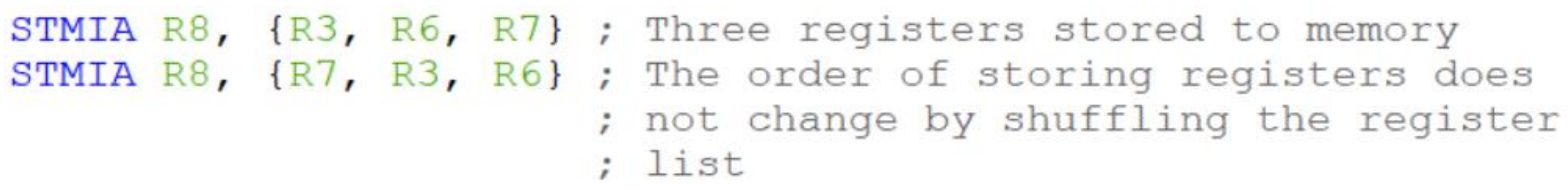
```
STMIA .W R8!, {R0 - R3} ; R8 changed to 0x7010 after store
                        ; ( increment by 4 words )
STMIA .W R8, {R0 -R3}   ; R8 unchanged after store
```

- In above example, **R0** is the first register to be copied to the memory, then **R1** and in the end **R3** is stored in the memory.
- The LDM instructions work in the opposite manner as compared to their STM counterparts.

Order of Registers

MULTIPLE WORD MEMORY ACCESSES

- For LDM, LDMIA, STM, and STMIA, the accesses occur in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.
- For LDMDB and STMDB the accesses occur in order of decreasing register numbers, with the highest numbered register using the highest memory address and the lowest number register using the lowest memory address.
- Therefore, the order in which we write the registers does not matter



```
STMIA R8, {R3, R6, R7} ; Three registers stored to memory
STMIA R8, {R7, R3, R6} ; The order of storing registers does
                       ; not change by shuffling the register
                       ; list
```

STACK MEMORY ACCESS

ARM Cortex-M4 Assembly Language

What is Stack Memory?

STACK MEMORY ACCESS

- The stack is part of main memory and is used to store data objects in last-in-first-out (LIFO) buffering format.
- In ARM Cortex-M processor, the stack always operates on 32-bit data.
- The stack pointer (R14, a CPU register) contains an address that points to a 32-bit data at the top of the stack.
- As we push data objects onto the stack, the addresses are decremented. The most recent item known as the “top of the stack” is the data object stored at the lowest address.
- Hence, the stack grows in the downward direction, and we say that it employs a full-descending stack management.
- The assembly instructions to access the stack memory region are **PUSH** and **POP**.

PUSH/POP Instructions

STACK MEMORY ACCESS

- A PUSH instruction copies one or more data objects from a register or a list of registers onto the stack, whose starting address is determined by SP. The POP instruction works in the opposite manner.

$$\begin{array}{ll} \text{PUSH}\{cond\} & reglist \\ \text{POP}\{cond\} & reglist \end{array}$$

- In the case of PUSH, the register or set of registers is source operand while the memory region reserved for the stack in the RAM is the destination.
- For POP instruction a register or set of registers is the destination and memory is the source.
- Contents of the source operands are not changed, while those of the destination operands get updated.

PUSH/POP Instructions

STACK MEMORY ACCESS

$$\begin{array}{ll} \mathrm{PUSH}\{cond\} & reglist \\ \mathrm{POP}\{cond\} & reglist \end{array}$$

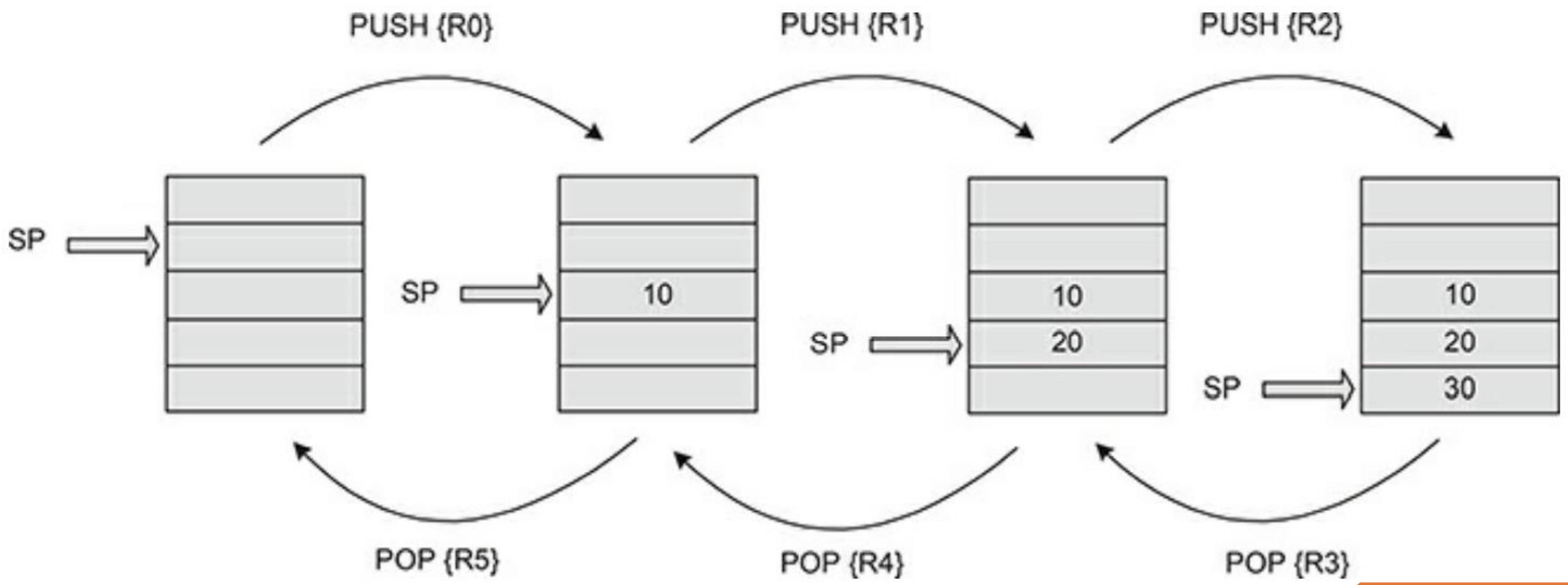
- To push a data object on the stack, the stack pointer is first decremented by 4, and then the 32-bit information is stored at the address specified by SP.
- To pop a data object from the stack, the 32-bit information pointed to by SP is first retrieved, and then the stack pointer is incremented by 4.
- It is important to note that with PUSH and POP instructions SP gets adjusted automatically.

PUSH/POP Instructions

STACK MEMORY ACCESS

```
PUSH {R0}    ; push the 32- bit value of R0 onto the stack
PUSH {R1}    ; push the 32- bit value of R1 onto the stack
PUSH {R2}    ; push the 32- bit value of R2 onto the stack

POP {R3}     ; retrieve a 32- bit value from the stack and store in R3
POP {R4}     ; retrieve a 32- bit value from the stack and store in R4
POP {R5}     ; retrieve a 32- bit value from the stack and store in R5
```



PUSH/POP Instructions

STACK MEMORY ACCESS

- The PUSH instruction with multiple registers, the registers are loaded in **descending order** of register numbers
  - For instance, when registers R1 and R2 are pushed together, R2 will be pushed first followed by R1.

- The POP instruction with multiple registers, the registers are loaded in **ascending order** of register numbers.
  - For the above-mentioned scenario, register R1 will be popped first followed by register R2.

PUSH/POP Instructions

STACK MEMORY ACCESS

```
PUSH {R7 , R8}                ; push R7 and R8 onto stack and
                              ; SP = SP -8
PUSH {R0 -R7 , R12 , R14 }    ; push R0 - R7 , R12 , R14 onto stack
                              ; and SP = SP -40
POP {R0 -R7 , R12 , R14 }     ; pop R0 - R7 , R12 , R14 from stack
                              ; and SP = SP +40
POP {R7 , R8}                 ; pop R7 and R8 from the stack
                              ; and SP = SP +8
```

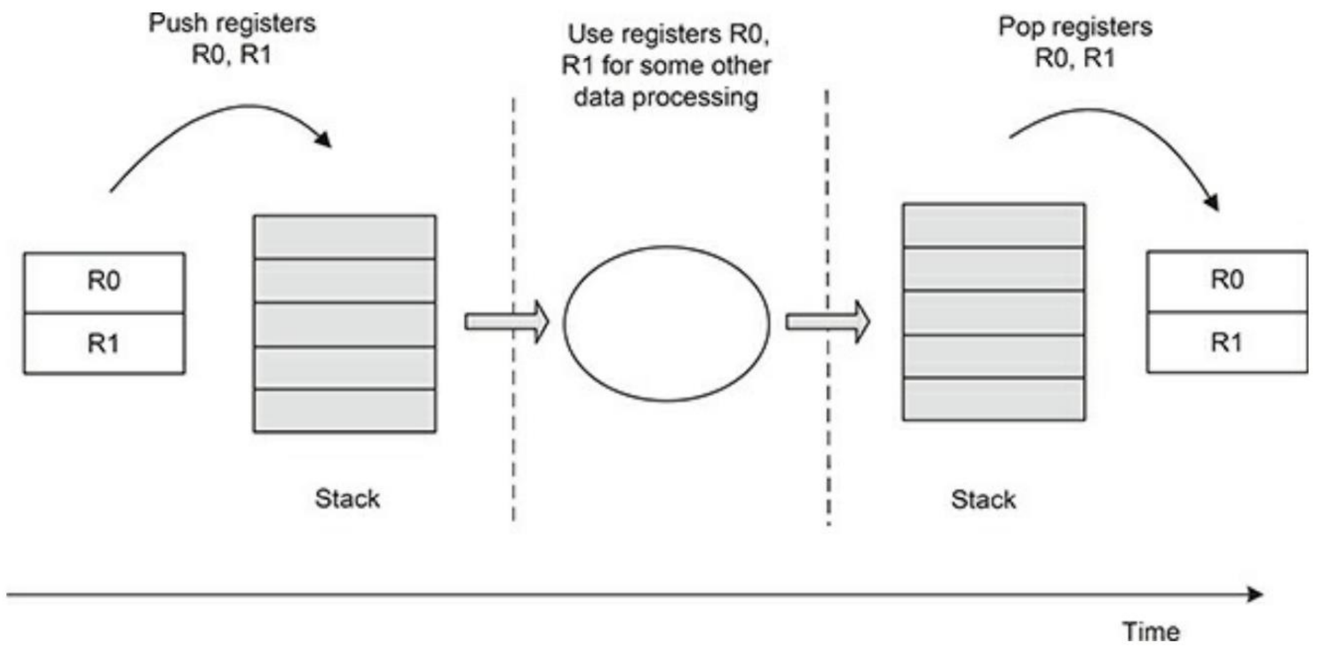
- When first instruction is executed, register R8 is pushed first followed by register R7.
- The register ordering in the instruction does not affect the ordering in which the registers are pushed onto the stack, i.e., writing PUSH {R8, R7} does not affect the order in which the registers are pushed onto the stack.
- The same is true for the POP operation performed with multiple registers.
- After executing the two PUSH instructions followed by two POP instructions, the register contents of all the registers used by the four instructions remain intact.

Why Do We Need a Stack?

STACK MEMORY ACCESS

Preserving Registers Values

- Consider a scenario where during the execution of a task it is required to use some registers, and we do not want to lose the current contents of these registers.
- What we normally do is to push the current contents of registers onto the stack, use them for processing the task and after completing the task, we can pop them from the stack.
- For proper stack usage, each PUSH instruction must have a corresponding POP instruction.



Why Do We Need a Stack?

STACK MEMORY ACCESS

Parameters Passing

- Stack can also be used for parameter passing purposes.
- Conventionally, few general purpose registers are used for passing the parameters while making a function call.
- If the available registers for parameter passing are not enough, then stack is used to pass parameters during function calls.
- The parameters are pushed onto the stack before making a function call and are popped inside the function.

Local Variables Declaration

- Stack memory is also used for declaring local variables.
- When a function is being executed, stack memory is allocated for its local variables, which is deallocated while returning from the function.

Why Do We Need a Stack?

STACK MEMORY ACCESS

Store Return Addresses

- In addition to parameter passing, stack can also be used for storing the return address.
- In Cortex-M processor, the LR (R14) register is meant to hold the return address.
- Once a function is called, it is good practice to store the LR register to the stack before starting the function execution.
- But what about the nested function calls?

Why Do We Need a Stack?

STACK MEMORY ACCESS

Store Return Addresses

- Let us illustrate the reason for doing so.
    - Assume that during the execution of the main function, another function named ‘func1’ is called.
    - The control will break from this point and starts executing the instructions of ‘func1’.
    - But once we are done with ‘func1’, we have to return to the main function and execute any of the remaining instructions.
    - Now consider the possibility that while executing ‘func1’, another function named ‘func2’ needs to be called.
    - At this point, the address in LR register, to return from ‘func1’, will be overwritten by the new return address corresponding to ‘func2’.
    - If the LR register is saved to the stack before calling ‘func2’ then returning from ‘func1’ will not be a problem.
    - This scenario is termed **nested function calls**.

Why Do We Need a Stack?

STACK MEMORY ACCESS

Store Return Addresses

- It is important to realize that implementing nested functions has a direct impact on the size of the stack.
- If the number of nested levels is **x** and we are only pushing LR register in each nested function call, then the stack space required will equal **4x**.
- However, if a certain subset of registers is pushed onto the stack, say **y** number of registers are pushed onto the stack each time a nested function call is made, then stack size requirement for implementing the nested function calls only is given by **4xy**.
- Based on this fact it is important that the stack size is selected carefully for the worst possible scenario to avoid any stack overflow condition

Why Do We Need a Stack?

STACK MEMORY ACCESS

Interrupt Handling

- Another usage of the stack is in case of interrupts.
- A number of registers are pushed automatically on the stack when an interrupt occurs.
- Similarly, the pushed registers will be popped automatically when exiting an interrupt handler and the SP is adjusted accordingly.
- The nested function call scenario also applies to the nested interrupts in the context of stack memory usage.
- Nested interrupts can occur, due to the possibility of assigning different priority levels to the interrupts from different sources.

Rules for Stack Usage

STACK MEMORY ACCESS

When we use PUSH/POP instructions, the SP is incremented or decremented automatically. However, the user is still required to keep track of following basic rules when using a stack:

Rule 1: **The PUSH and POP operations should always be performed inside the memory region specified for stack.** Violation of this rule can cause one of the following two issues:

- **Stack Overflow**: occurs when a push operation is performed, while the stack is already completely filled and no more room is available
- **Stack Underflow**: occurs when the pop operation is performed while the stack is empty already

Rules for Stack Usage

STACK MEMORY ACCESS

<u>Rule 2</u>: **The number of push operations should always have a matching number of pop operations following a specified order**. An overflow can occur due to

- a mismatch in the number of push and pop operations
- large number of push operations (performed without any pop operation in between). This can happen due to nested function calls as well as recursion.

- On the other hand, a stack underflow usually occurs due to corruption of the stack pointer.
- The stack overflow as well as underflow may result in data corruption or a bus fault when a read from or write to an invalid address is performed.

THANK YOU

Any Questions???