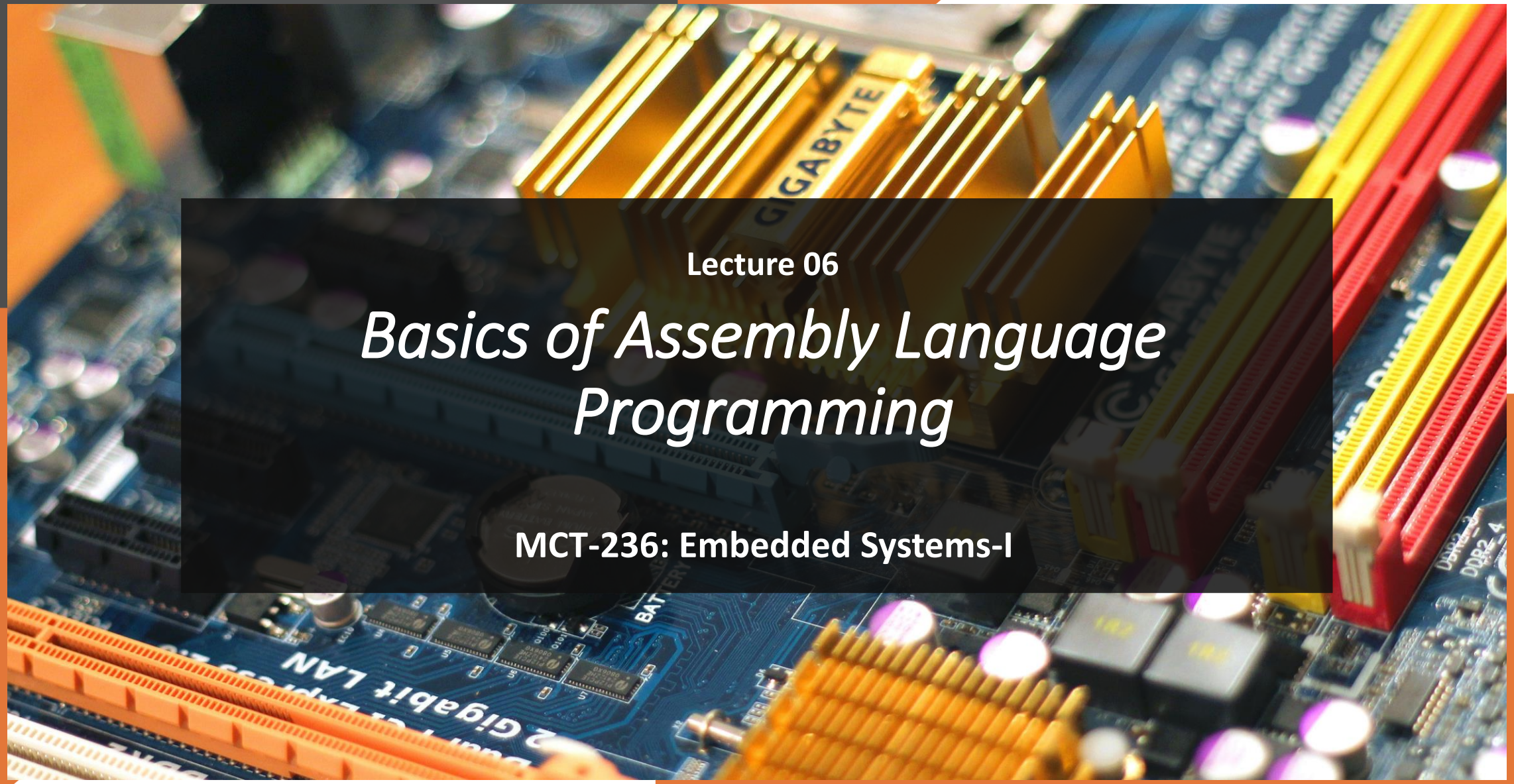
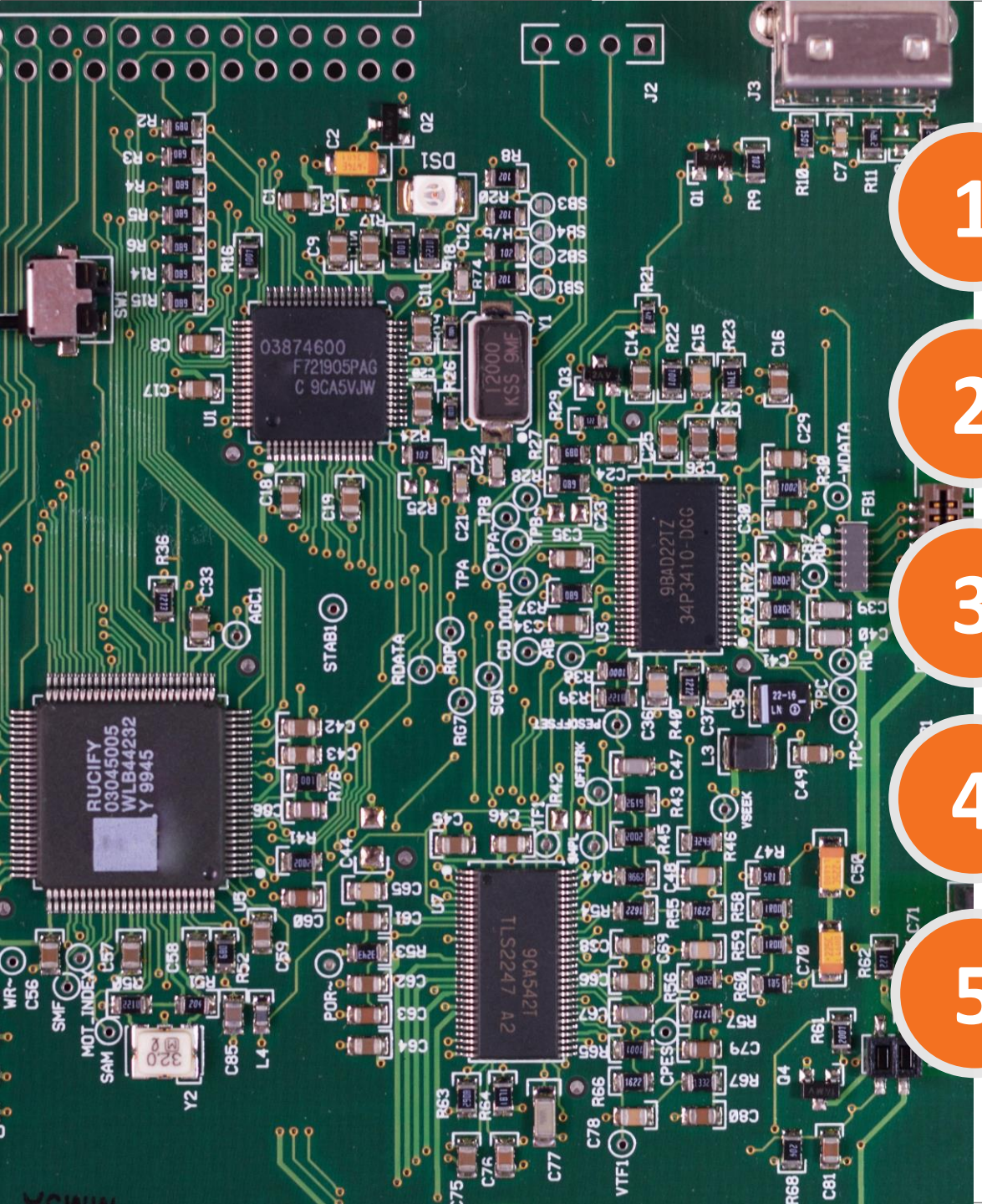


Lecture 06

Basics of Assembly Language Programming

MCT-236: Embedded Systems-I



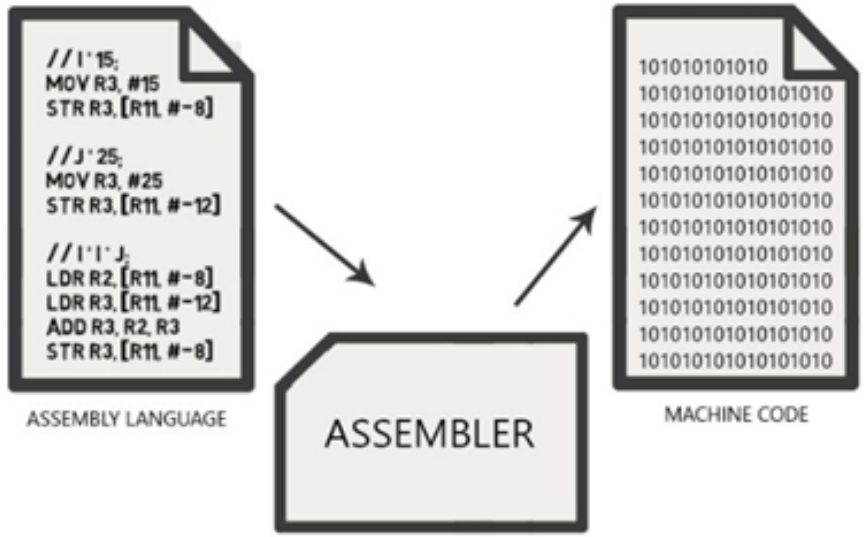
1. CORTEX-M ASSEMBLER
2. INTRODUCTION TO ARM INSTRUCTION SETS
3. ARM ASSEMBLY LANGUAGE INSTRUCTIONS
4. FIRST ASSEMBLY PROGRAM
5. INSTRUCTION ENCODING

CORTEX-M ASSEMBLER

Basics of Assembly Language, ARM Cortex-M4 Architecture, TIVA Cortex-M4 Assembler

Basics of Assembly Language

CORTEX-M ASSEMBLER

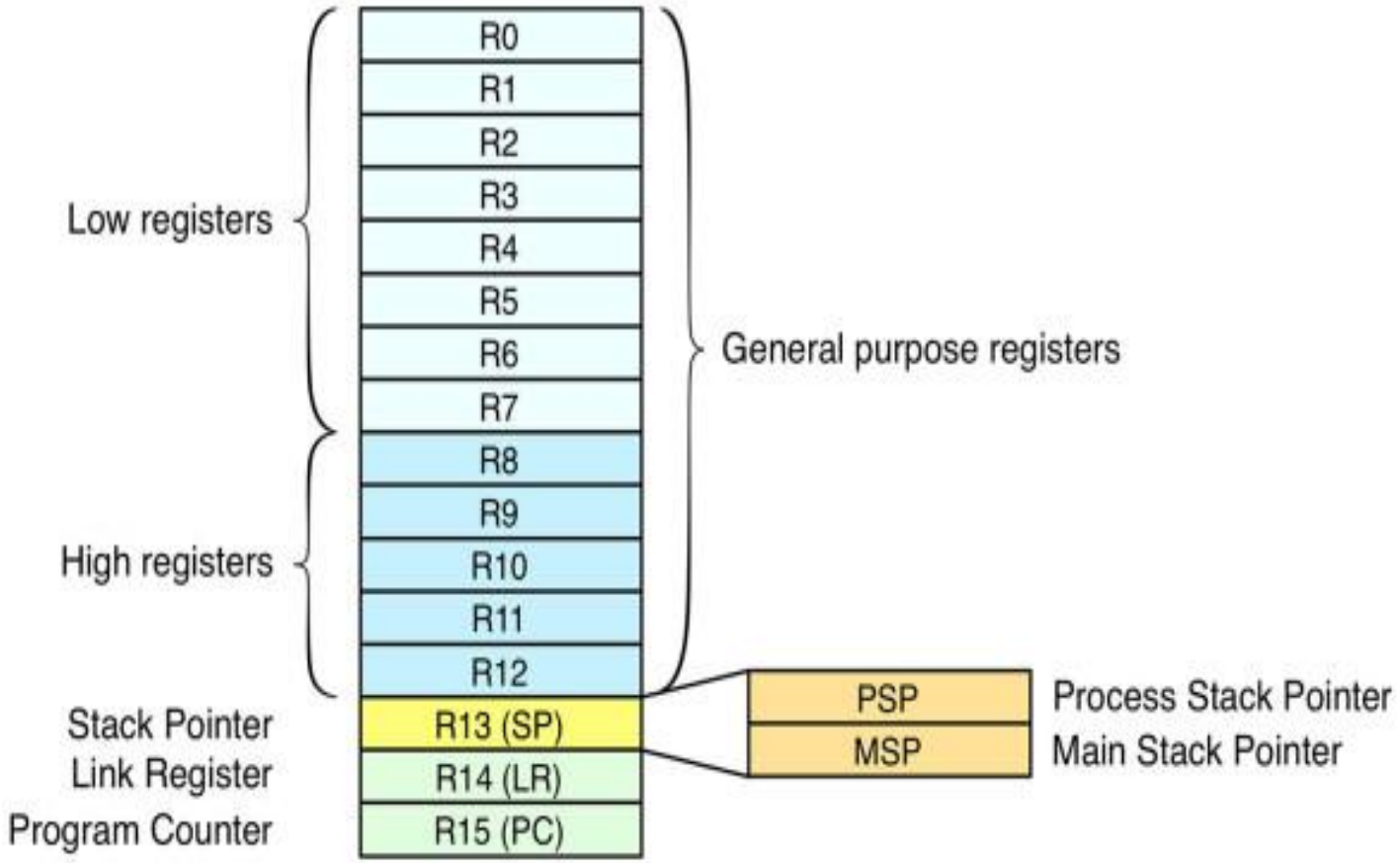


- **Machine Language**: Computer program collection of numbers in the form of binary strings
- **Assembly Language**: Computer program collection of words called mnemonics
  - a low-level programming language, which is specific for a processor architecture
- **Assembler**: Computer program that translate the assembly language instruction into executable machine language instructions
- Advantages of learning assembly language:
  - It provides complete as well as precise control of the available hardware resources
  - It gives the user complete understanding of the system architecture
- In case of high-level language, we are unaware of the underlying hardware capabilities, which can lead to highly inefficient implementation

ARM Cortex-M4 Architecture

CORTEX-M ASSEMBLER

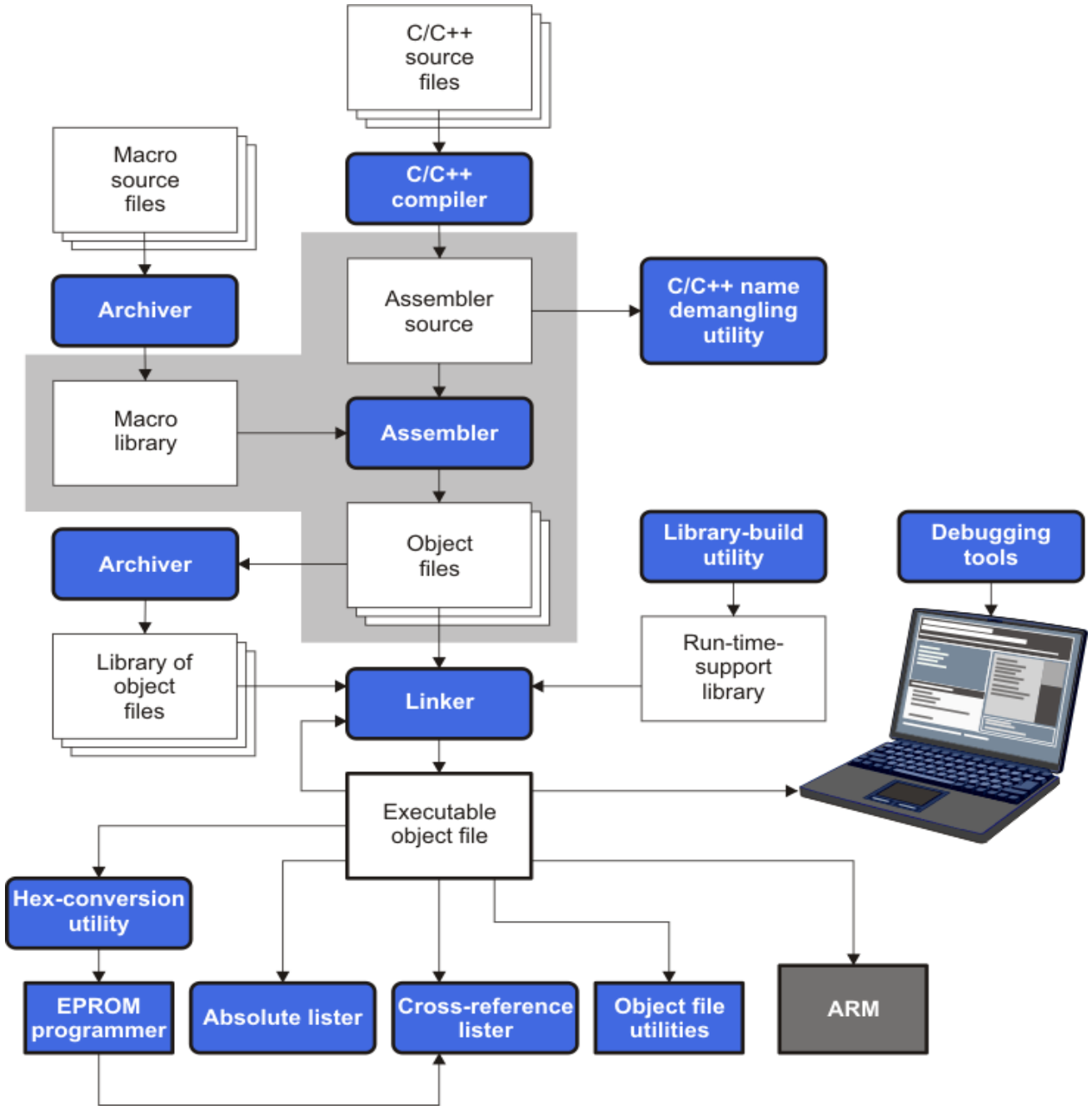
- RISC architecture with simple implementation of load/store model (R2R)
- 16 read/write, 32-bit registers
- **Stack Pointer (SP)**: holds the address of top element of program stack
- **Link Register (LR)**: holds the return address from PC when BL/BLX statement is used
- **Program Counter (PC)**: holds the current program address



TIVA Cortex-M4 Assembler

CORTEX-M ASSEMBLER

- Translates assembly language **source files** into machine language **object files**
- The executable object file can be executed by ARM device
- Source Files can contain the following assembly language elements
  - Assembler Directives
  - Assembly Language Instructions
  - Macro Directives



INTRODUCTION TO ARM INSTRUCTION SETS

ARM vs Thumb Instructions, Thumb2 Instructions

ARM vs Thumb Instruction Sets

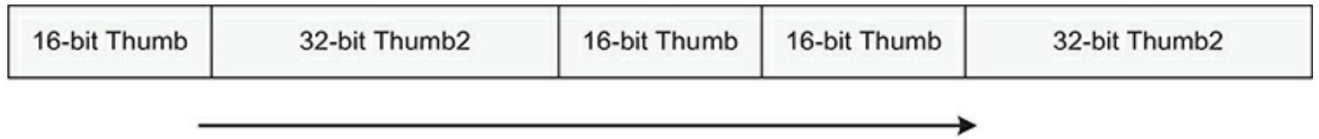
INTRODUCTION TO ARM INSTRUCTION SETS

- The ARM processors used to operate mainly in two states.
- **ARM State** allowed only the execution of 32-bit word aligned instructions providing higher performance
- **Thumb State** allowed only 16-bit half-word aligned instruction execution providing higher instruction code density. In Thumb state
  - all the functionality provided by ARM instructions was not possible
  - it required multiple Thumb instructions to complete certain single cycle operations of ARM state
- The processors that supported both 32-bit as well as 16-bit instructions required to switch between ARM state and Thumb state incurring an additional overhead of state switching

Thumb2 Instruction Set

INTRODUCTION TO ARM INSTRUCTION SETS

- **Thumb2** architecture introduced many 32-bit instructions in addition to the existing 16-bit Thumb instruction set
- These 32-bit and 16-bit instructions can be intermixed freely for high code density as well as better execution efficiency
- The Cortex-M processor is based on ARMv7-M architecture and as a result uses Thumb2 instruction set
- One limitation of Thumb2 instruction set architecture is that the older ARM code is no longer reusable on Cortex-M processor, which only supports Thumb2 architecture
- Some key advantages over the conventional ARM processors
  - Since no state switching is required, this results in an improved execution performance and saves instruction memory space as well.
  - In conventional ARM processors separate ARM and Thumb code source files were maintained. However, this is not required for Cortex-M processors, providing simpler software development as well as maintenance.



Instruction Execution in Thumb2 Architecture

ARM ASSEMBLY LANGUAGE INSTRUCTIONS

ARM Cortex-M4 Microcontroller & Keil µVision Toolchain

Instruction Format

ARM ASSEMBLY LANGUAGE INSTRUCTIONS

$$[label]\ mnemonics\ [operand\ list]\ [;comment]$$

Field 4 (Optional):
- Starts with semi-colon (;)
- Ignored by the assembler
- Make program more comprehensible for human
- Don't affect the program operation

Field 2: (Mandatory)
- Each mnemonics (also called opcode) corresponds to machine instruction
- Assembler must translate each mnemonics into binary equivalent
- Contains one of following items:
  - Machine-instruction mnemonics (STR etc.)
  - Assembler Directive (EQU, END, etc.)
  - Macro Directive

Field 3 (Optional):
- Number of operands depends on instruction
- Normally, first operand is the destination of the operation
- An operand can be one of the following:
  - an immediate operand (usually a constant or symbol)
  - a register operand
  - a memory reference operand
  - an expression that evaluates to one of the above

Field 1 (Optional):
- Alphanumeric unique names used to define starting location of a block of instruction
- May be used as operand for some instructions
- First character of a line reserved for the label field, and it should be left blank for the instructions with no labels

Instruction Format

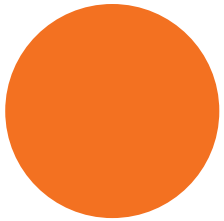
ARM ASSEMBLY LANGUAGE INSTRUCTIONS

$$[\boldsymbol{label}]\ \boldsymbol{mnemonics}\ [\boldsymbol{operand\ list}]\ [;\boldsymbol{comment}]$$

Examples of different assembly instructions and indicate different fields

Types of Assembly Language Instructions

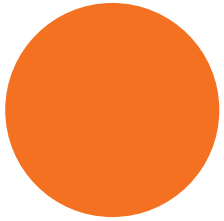
ARM ASSEMBLY LANGUAGE INSTRUCTIONS



Data Processing Instructions

Manipulate the data within the registers

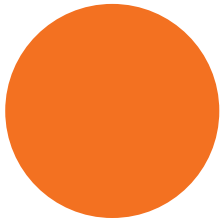
Move, arithmetic, logical, and relational instructions



Memory Access Instructions

Move data to and from the main memory

Load/Store instructions



Branch Instructions

Change the control flow of the program

Change register R15, PC to implement conditional statements, loops etc.

ADD Data Processing Instruction

ARM ASSEMBLY LANGUAGE INSTRUCTIONS

- In below listing ADD and ADDS are called instruction mnemonics and represent the opcode field
- The registers R1 - R6 and immediate value #0x123 are the operands
- The # symbol is used to specify an immediate value or number.
- The comments provide useful description of the instructions.
- The ADD instruction only performs addition of operands and the result is stored in the destination operand

```
ADD    R1, #0x6          ; R1 = R1 + 0x6
ADD    R4, R2            ; R4 = R4 + R2
ADD    R6, R5, R3        ; R6 = R5 + R3
ADDS   R6, R5, R3        ; R6 = R5 + R3, flags are updated
```

ADD Data Processing Instruction

ARM ASSEMBLY LANGUAGE INSTRUCTIONS

- However, the optional suffix S in ADDS instruction is responsible for updating the condition code flags in the application program status register in addition to performing addition
- Th first two instructions use two operands, while the third instruction uses three operands
- In the first two instructions the destination register is the same as one of the source operands
- However, the third instruction uses a destination operand that is different from the source operands
- Based on this fact, it can be concluded that the number of operands for an instruction may vary, depending on the instruction format used

```
ADD    R1, #0x6          ; R1 = R1 + 0x6
ADD    R4, R2            ; R4 = R4 + R2
ADD    R6, R5, R3        ; R6 = R5 + R3
ADDS   R6, R5, R3        ; R6 = R5 + R3, flags are updated
```

MOV Data Processing Instruction

ARM ASSEMBLY LANGUAGE INSTRUCTIONS

- MOV instruction is used for simple data transfers within the processor
- Another data movement operation may require to transfer the contents of one register to another register
- The use of MOV assembly instruction to perform these data exchanges, involving an immediate data transfer to a register and data transfer between two registers
- The second instruction illustrates how the ASCII value can be used as an immediate data operand
- The MOV instruction cannot be used for data transfers between processor registers and memory

```
MOV   R2,   #0x123          ; move immediate value of 0 x123 to R2
MOV   R4,   #'A'            ; move ASCII value of A (i.e., 0x41) to
MOV   R7,   R3              ; move the contents of R3 to R7
```

LDR and STR Memory Access Instructions

ARM ASSEMBLY LANGUAGE INSTRUCTIONS

- LDR and STR instructions allow data to be transferred between processor and memory
- The load register, LDR, instruction is used to transfer data from memory to the processor register, while store register, STR, instruction is used to transfer data from processor register to memory
- The first LDR instruction retrieves data from memory address specified by the value of register R1 and transfers it to register R2
- The second LDR instruction loads the immediate value to the destination register and, in that context, it is functionally equivalent to the MOV instruction used for loading immediate value
- The last instruction stores the contents of R4 to the memory location addressed by the value in R3

```
LDR R2, [R1]      ; load R2 with data from memory pointed to by
LDR R0, = NUM1    ; load R0 with the value of constant NUM1 (thi
                  ; constant might represent the memory address)
STR R4, [R3]      ; store R4 to a memory location addressed by R
```

Unconditional- and Combined Compare-Branch Instructions

ARM ASSEMBLY LANGUAGE INSTRUCTIONS

- Unconditional/conditional branch instructions perform comparison in addition to branching
  - The branch instruction, mnemonic **B**, represents unconditional branch
  - The compare and branch on zero instruction, **CBZ**, performs conditional branch if the result of comparison is zero
  - If the result of comparison is not zero, then **CBZ** behaves as failed conditional branch instruction
- The label '**loop1**' effectively is the address of the instruction **LDR R2, [R1]**.
- The same program also uses **CBZ**, which compares the value in register **R5** with zero and if the result of comparison is true a branch to address '**label1**' occurs.
- The use of instruction **CBZ R5**, **label1** also implies that there is an instruction inside '**loop1**' block that decrements the value of register R5 and eventually a branch to '**label1**' occurs.

```
loop1 LDR R2 , [R1]          ; load R2 with data from memory poin
                             ; to by register R1
    ...
      CBZ R5 , label1        ; Compare R5 with zero . If comparis
                             ; result is true then branch to labe
    ...
      B loop1                ; jump to the memory location labele
                             ; loop
label1
      MOV R3 , #0 x034
```

FIRST ASSEMBLY LANGUAGE PROGRAM

Assembly Language Program

FIRST ASSEMBLY LANGUAGE PROGRAM

- Can you guess, what this assembly language program is performing?
  Sum of 1st 5 even numbers {2, 4, 6, 8, 10} = 30

```
        MOV R0 , #0          ; R0 will accumulate the sum
        MOV R1 , #2          ; R1 will have the updated even number
        MOV R2 , #5          ; the counter for the loop
lbegin
        CBZ R2 , lend        ; If R2 != 0 continue with the next
                             ; instruction
        ADD R0 , R1
        ADD R1 , #2
        SUB R2 , #1
        B lbegin             ; branch unconditionally to lbegin
lend
```

Steps to Run the Program on Cortex-M Processor

FIRST ASSEMBLY LANGUAGE PROGRAM

- The previous program cannot run straightaway as we need to address following issues:
- **Need a Software Package**: to write the code, compile (assemble) it to generate the machine code and link it to generate the executable
- **Aware of Reset Sequence & Memory Map**: to know how the Cortex-M processor is initialized, where our code and data will reside, and where the interrupts/exceptions handlers are loaded in the memory
- **Select the Target**: to download and run the program on an actual hardware using a microcontroller or are we interested in executing it using a simulator

Writing Program to Build the Executable

FIRST ASSEMBLY LANGUAGE PROGRAM

- The first instruction of the user program is not the first instruction that is executed when the program is run
- There are certain system level initializations that need to be performed before the user program execution is started
- The minimum system initialization that will allow to run the user program includes the following two key components as the basic building blocks
  - **Assembler directives:** commands to the assembler that are followed before the first user instruction executes
  - **Reset sequence:** The understanding of the reset sequence and the memory map loaded with the object file will help us in writing assembly programs

Assembler Directives

FIRST ASSEMBLY LANGUAGE PROGRAM

- An assembly source file is a collection of assembly language instructions, **assembler directives** and **macro directives**.
- An assembler translates assembly language source files to machine understandable object file.
- It should be clear that the assembler and linker syntax depends on which software tools are being used
- The assembler syntax is based on the use of assembler directives, which are pseudo-opcodes or pseudo-operations performed by an assembler and are not part of the assembly instruction set.
- These assembler directives are executed during the assembling process at compilation time, in contrast to assembly instructions that are executed at runtime

Assembler Directives

FIRST ASSEMBLY LANGUAGE PROGRAM

- One batch of these directives is used to control the placement of data and code in the memory.
- For example, the **AREA** directive is used to instruct the assembler for a new code or data section.
- **Sections** are independent, named individually, and comprise indivisible chunks of code or data that are manipulated by the linker
- The general syntax for the AREA directive is **AREA *sectionname*** {,***attr***}{,***attr***}…
  - ***sectionname*** is the name assigned by the user to a particular data or code section
  - Some of the section names are conventional. For instance, | .text | is used to mention code sections generated by the C compiler, or it can also be used for code sections associated with the C library.
  - The ***attr*** is one or more comma-delimited optional section attributes.

Assembler Directives

FIRST ASSEMBLY LANGUAGE PROGRAM

- Some useful attributes for AREA directive

Attribute	Description
READONLY	Indicates that this section must not be written to. This is the default for code areas.
NOINIT	Use of this attribute indicates that the data section is uninitialized, or initialized to zero. A section with this attribute contains only space reservation directives SPACE or DCB, DCD, DCDU, DCQ, DCQU, DCW, or DCWU with initialized values of zero. You can decide at link time whether an area is uninitialized or zero initialized.
READWRITE	Indicates that this section can be read from and written to. This is the default for Data areas.
DATA	Contains data, not instructions. READWRITE is the default.
CODE	Contains machine instructions. READONLY is the default.
ALIGN=*number*	By default, sections are aligned on a four-byte boundary. The *number* can have any integer value from 0 to 31. The section is aligned on a 2^{number} byte boundary. For example, if expression is 10, the section is aligned on a 1 KB boundary.

Assembler Directives

FIRST ASSEMBLY LANGUAGE PROGRAM

- The second batch of directives is used for defining variables or constants of different sizes.
- For example, **SPACE** is used to allocate a zero initialized block of memory for one or more variables.
- Similarly, **DCB**, **DCW**, and **DCD**, respectively, are used to allocate byte, half-word, or word size of memory and specify the initial content.
- The **DCB**, **DCW**, and **DCD** assembler directives can be used in both **CODE** as well as **DATA** memory sections.
- The **EQU** directive gives a symbolic name to a numeric constant. In addition, EQU can also be used to generate register-relative or PC-relative addresses.
- One can define constants using EQU and then use them within the user program code.

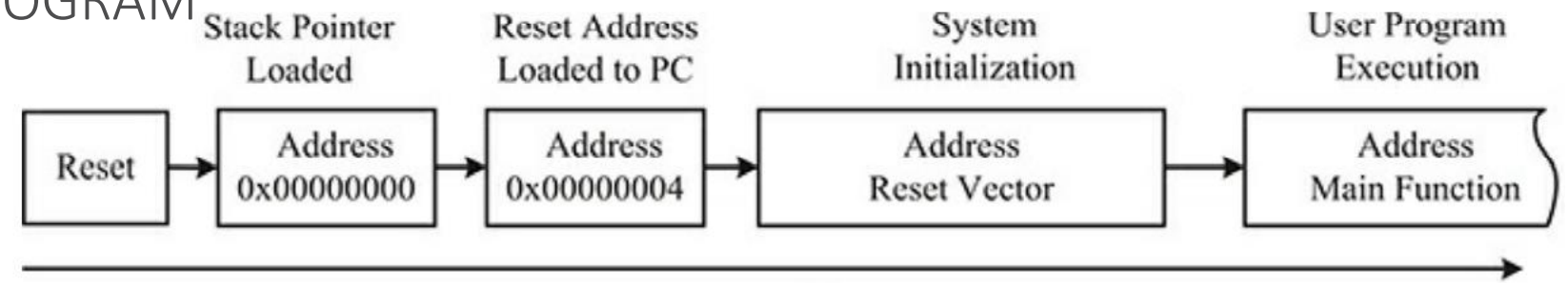
Assembler Directives

FIRST ASSEMBLY LANGUAGE PROGRAM

- Another set of useful directives include **ENTRY**, **END**, **IMPORT**, and **EXPORT**.
- The **ENTRY** directive declares an entry point to a program.
- The **END** directive informs the assembler that it has reached the end of a source file.
- The **EXPORT** directive declares a symbol that can be used by the linker to resolve symbol references in two different object files.
- **GLOBAL** is a synonym for **EXPORT**. GLOBAL and EXPORT directives tell the assembler that the associated labels are not defined in the current assembly program file and need to be investigated some other object files (generated from other source files) or libraries.
- The **THUMB** directive instructs the assembler to interpret subsequent instructions as Thumb or Thumb2 instructions.
- The **PRESERVE8** directive specifies that the current file preserves 8-byte alignment of the stack.

Reset Sequence

FIRST ASSEMBLY LANGUAGE PROGRAM



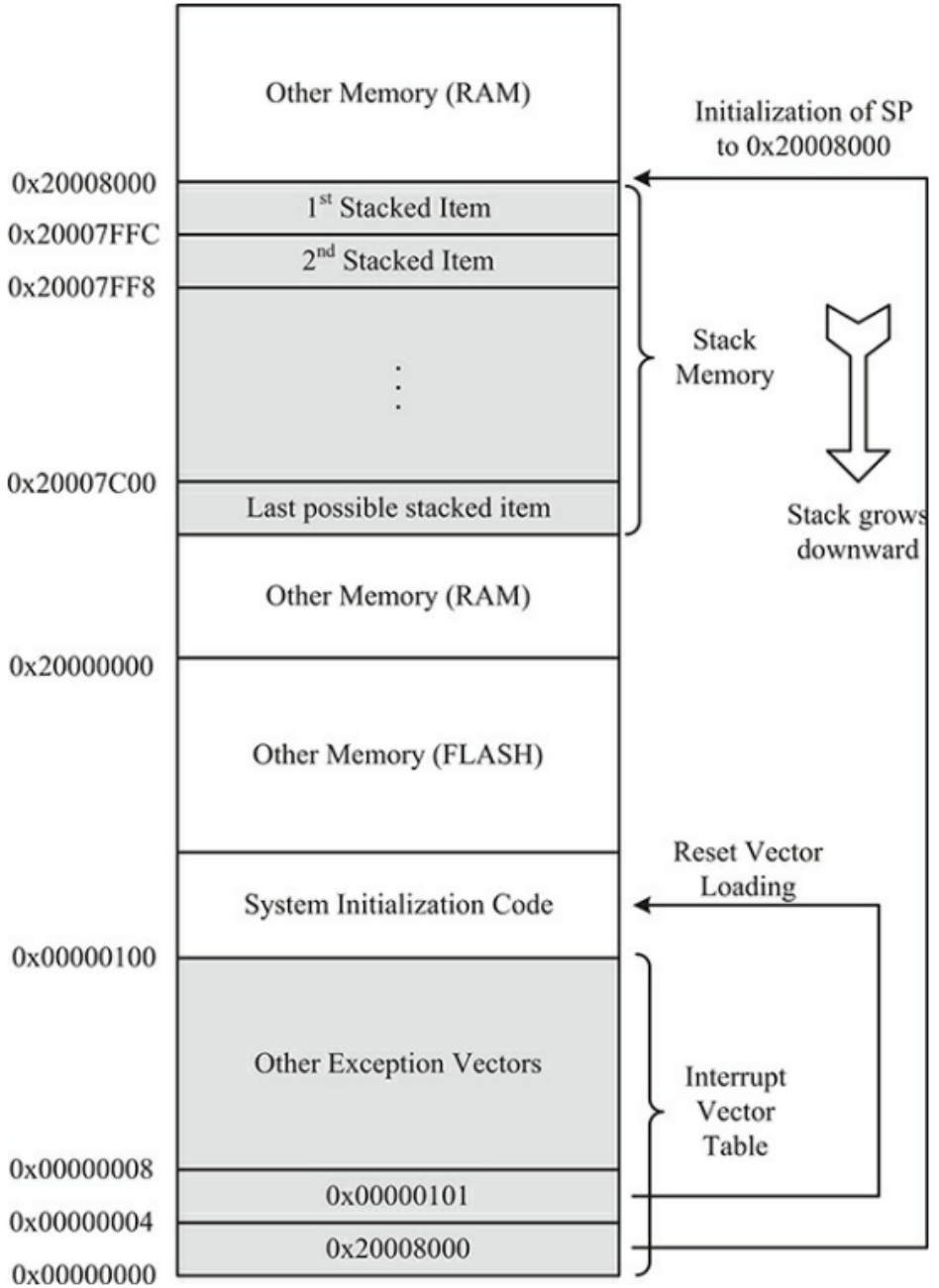
After a reset Cortex- M processor reads two words from code memory:

- First, it fetches a 32-bit value from address 0x00000000 that contains the address of the stack area in memory (RAM) and initializes the main stack pointer (MSP) register with that value.
  - It is important to remember that only MSP is initialized on the reset and the value of PSP is undefined on reset
- Second, it fetches another 32-bit value from address 0x00000004 that contains the address where the boot code or reset handler resides and initializes its program counter (PC) register with this value.
- Also, the bit field T in the execution program status register (EPSR) is set to '1' to mark the Thumb state.
- The link register LR is reset to 0xFFFFFFFF.
- Once the PC is initialized, Cortex-M processor will start executing the reset handler instructions and from this point onward, it is ready to fetch and execute the user application program.

Memory Map & Reset Sequence

FIRST ASSEMBLY LANGUAGE PROGRAM

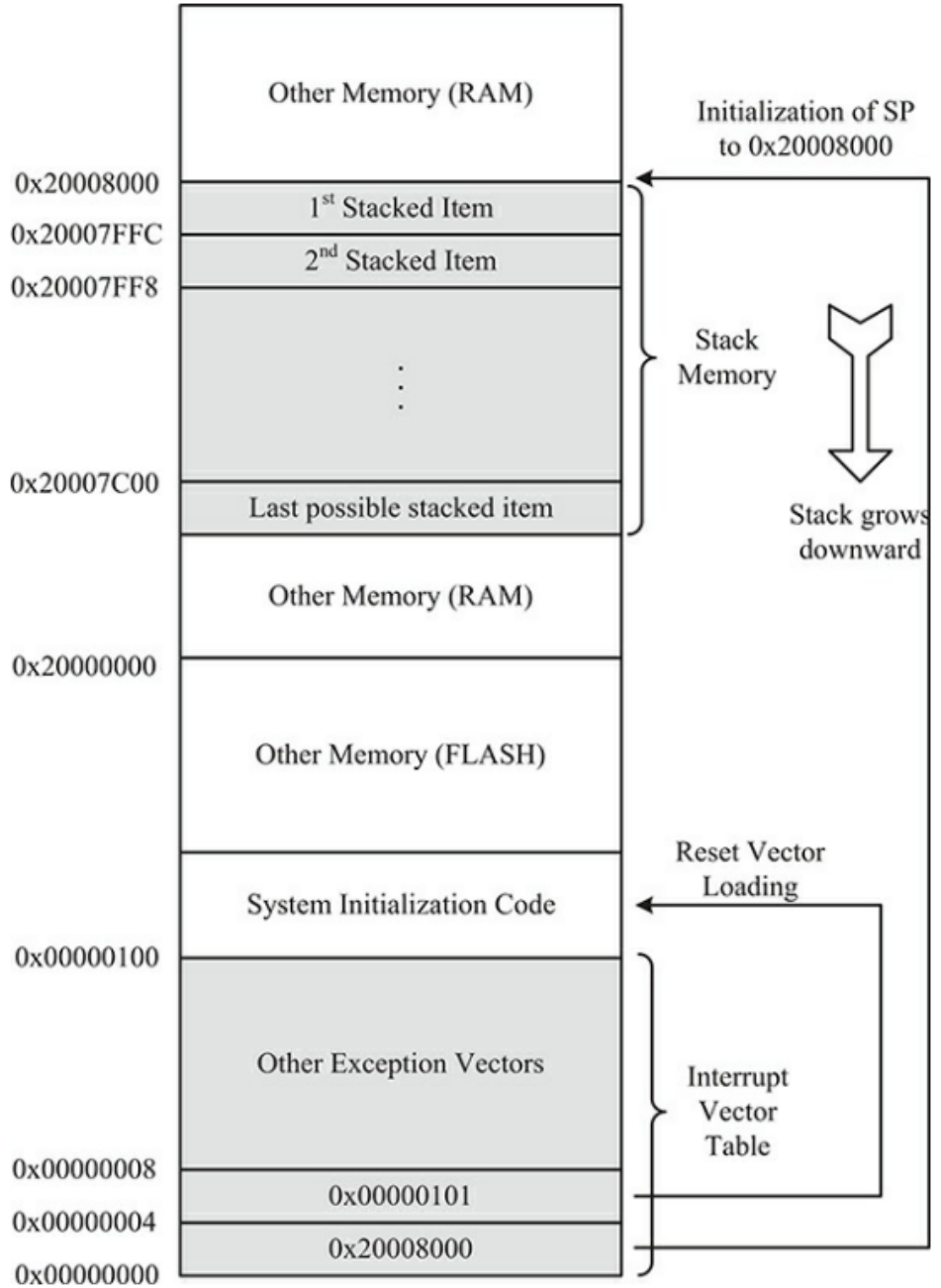
- Figure provides an illustration of the memory map, when the memory is loaded with a program
- Address 0x00000000 is initialized with a 32-bit value (0x20008000 in this example). The linker is responsible for assigning absolute addresses and generates this value.
- Beside other factors, these tools will take the stack size defined by the programmer for this program into account
- For this typical example, the programmer is required to have a stack of size 1 KB. So, the linker reserved a memory region starting from 0x20007C00 to 0x20007FFF (0x20007FFF - 0x20007C00 = 0x400 = 1024 or 1 KB) for the stack.
- The stack grows in the downward direction.



Memory Map & Reset Sequence

FIRST ASSEMBLY LANGUAGE PROGRAM

- Following the contents of 0x00000000, we find that the memory contains a table called **an interrupt/exception vector table**.
- It contains vectors or addresses of those functions which are executed whenever an interrupt/exception occurs
- Let us look at the first vector, which is located at address 0x00000004 in the memory. This vector is called the **reset vector**, which contains the address of the reset handler.
- The instructions written as part of the reset handler are also sometimes referred to as **boot code**.
- You can see that the code of the reset handler is located at address 0x00000100.
- The remaining interrupt/exception vectors, if any, are loaded to the memory after the reset vector in the exception table.
- It is important to realize that the vectors are placed in a pre-specified order in the interrupt vector table.



How an Assembly Program Run on a Cortex-M

FIRST ASSEMBLY LANGUAGE PROGRAM

- In summary, an assembly program needs to perform the following sequence of activities to ensure that the user application program is executed properly.
  1. Define the stack size and reserve appropriate memory space for stack.
  2. Define the reset vector.
  3. Write the reset handler code to perform any system related initializations and then make a jump to the user application program.

Complete First Assembly Program

FIRST ASSEMBLY LANGUAGE PROGRAM

- First assembly program that can be converted to a complete assembly program and complied to build the executable
- You can observe that a stack area of size 256 bytes is reserved in this example program
- The label **__Vectors** marks the beginning of the interrupt vector table and is initialized inside the code memory region.
- This memory segment is placed at the starting address of 0x00000000 in the code memory region.
- The first entry is defined as top of the stack region address. This is implemented by first reserving a memory region of size '**StackSize**' with starting address given by label '**MyStackMem**' in the RAM memory.
- The top of the stack address is then defined as '**MyStackMem** + **StackSize**' and being the first entry of the code memory region, it is stored at address 0x00000000, from where it is loaded to the SP (R13) after reset.

```
   THUMB                ; Marks the THUMB mode of operation

StackSize  EQU  0x00000100 ; Define stack size of 256 byes

   AREA STACK, NOINIT, READWRITE, ALIGN=3
MyStackMem  SPACE StackSize

   AREA    RESET, READONLY
   EXPORT  __Vectors
__Vectors
   DCD MyStackMem + StackSize ; stack pointer for empty stack
   DCD Reset_Handler          ; reset vector
   AREA    MYCODE, CODE, READONLY
   ENTRY
   EXPORT Reset_Handler
Reset_Handler
   MOV R0, #0            ; Initial value of sum
   MOV R1, #2            ; First even number
   MOV R2, #5            ; Counter for the loop iterations

lbegin
   CBZ R2, lend          ; Terminate loop if counter is zero
   ADD R0, R1            ; Build the sum
   ADD R1, #2            ; Generate next even number
   SUB R2, #1            ; Decrement the counter
   B lbegin
lend
   END
```

Complete First Assembly Program

FIRST ASSEMBLY LANGUAGE PROGRAM

- It is important to mention that SP initialization value is not an interrupt vector entry, but we have made it the first entry of the vector table to simplify the implementation.
- The second entry of the vector table is '**Reset_Handler**', which is the label of the reset interrupt service routine.
- The first instruction in the reset handler is indeed the first instruction that is fetched and executed by the processor after reset.
- The reset is the first interrupt entry in the interrupt vector table and is of highest priority as well

```
   THUMB                ; Marks the THUMB mode of operation

StackSize  EQU  0x00000100 ; Define stack size of 256 byes

   AREA STACK, NOINIT, READWRITE, ALIGN=3
MyStackMem  SPACE StackSize

   AREA    RESET, READONLY
   EXPORT  __Vectors
__Vectors
   DCD MyStackMem + StackSize ; stack pointer for empty stack
   DCD Reset_Handler          ; reset vector
   AREA    MYCODE, CODE, READONLY
   ENTRY
   EXPORT Reset_Handler
Reset_Handler
   MOV R0, #0             ; Initial value of sum
   MOV R1, #2             ; First even number
   MOV R2, #5             ; Counter for the loop iterations

lbegin
   CBZ R2, lend           ; Terminate loop if counter is zero
   ADD R0, R1             ; Build the sum
   ADD R1, #2             ; Generate next even number
   SUB R2, #1             ; Decrement the counter
   B lbegin
lend
   END
```

Complete First Assembly Program

FIRST ASSEMBLY LANGUAGE PROGRAM

- There are three different memory sections are defined
- The **STACK** section is defined in the RAM memory area using READWRITE attribute, while **RESET** and **MYCODE** sections are defined in the Flash memory area using READONLY attribute
- We observe that the RESET section defines two-word size entries, of which the first entry holds the stack top address while the second entry is the reset handler address
- In this example, the user program is implemented as part of the reset interrupt service routine and is made part of the MYCODE section.
- When the executable for this program is constructed, it is important that the RESET section is placed at address 0x00000000 in the code memory. This is achieved by instructing the linker using linker directives.
- Managing different memory areas and placement of code and data sections to appropriate memory areas is implemented using a linker script or linker scatter file.

```
   THUMB                ; Marks the THUMB mode of operation

StackSize  EQU  0x00000100 ; Define stack size of 256 byes

   AREA STACK, NOINIT, READWRITE, ALIGN=3
MyStackMem  SPACE StackSize

   AREA    RESET, READONLY
   EXPORT  __Vectors
__Vectors
   DCD MyStackMem + StackSize ; stack pointer for empty stac
   DCD Reset_Handler          ; reset vector
   AREA    MYCODE, CODE, READONLY
   ENTRY
   EXPORT Reset_Handler
Reset_Handler
   MOV R0, #0            ; Initial value of sum
   MOV R1, #2            ; First even number
   MOV R2, #5            ; Counter for the loop iterations

lbegin
   CBZ R2, lend          ; Terminate loop if counter is zero
   ADD R0, R1            ; Build the sum
   ADD R1, #2            ; Generate next even number
   SUB R2, #1            ; Decrement the counter
   B lbegin
lend
   END
```

INSTRUCTION ENCODING

16/32-Bit Instructions Encoding

ARM Instruction Encoding

INSTRUCTION ENCODING

- **Instruction encoding** is the process of assigning a unique binary codeword to each assembly instruction.
- One of the main jobs of an assembler is to translate each assembly instruction to an equivalent codeword.
- Some of the assembly instructions are encoded to 16-bit codewords, while some other instructions are encoded using 32bit codewords.
- However, there are many assembly instructions that can be encoded to either 16-bit or 32-bit codewords.

16-Bit Instruction Encoding

INSTRUCTION ENCODING

- Consider ADDS instruction for register and immediate data as well as register and register addition.
- The first two instructions illustrate two possibilities resulting in 16-bit encodings, primarily due to the appropriate choice of immediate values.
- The third instruction, illustrating register to register addition, is also encoded using 16-bit encoding

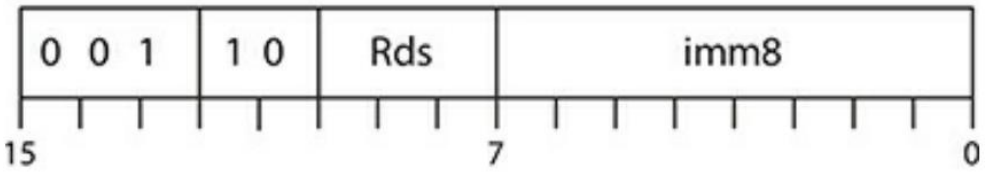
```
ADDS R2, #16          ; R2 = R2 + 16, Instruction is 16-
bit encoded
ADDS R4, R2, #6       ; R4 = R2 + 6, Instruction is 16-
bit encoded
ADDS R6, R5, R3       ; R6 = R5 + R3, Instruction is 16-
bit encoded
```

16-Bit Instruction Encoding

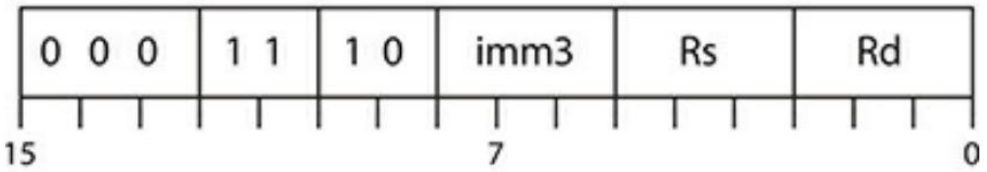
INSTRUCTION ENCODING

- Bit field of size 3 bits is allocated for each register operand that allow to select an arbitrary register from R0-R7, i.e., general-purpose low registers
- This means that, general purpose high registers (R8-R12) cannot be used as operands in 16- bit instruction encoding
- The immediate value operand field size depends on the choice of the destination register operand.
- If the same register is used as source and destination operand, the immediate value can have an 8-bit size.
- On the other hand, choosing different destination and source registers allows only a 3-bit immediate value operand.

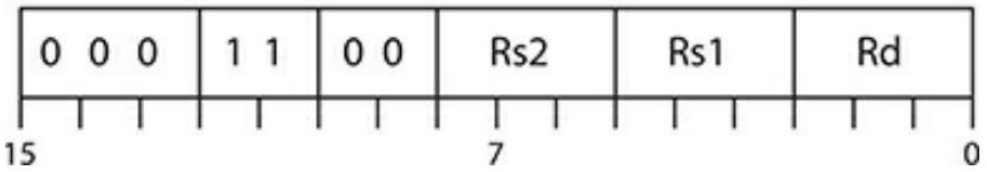
ADDS Rds, imm8



ADDS Rd, Rs, imm3



ADDS Rd, Rs1, Rs2

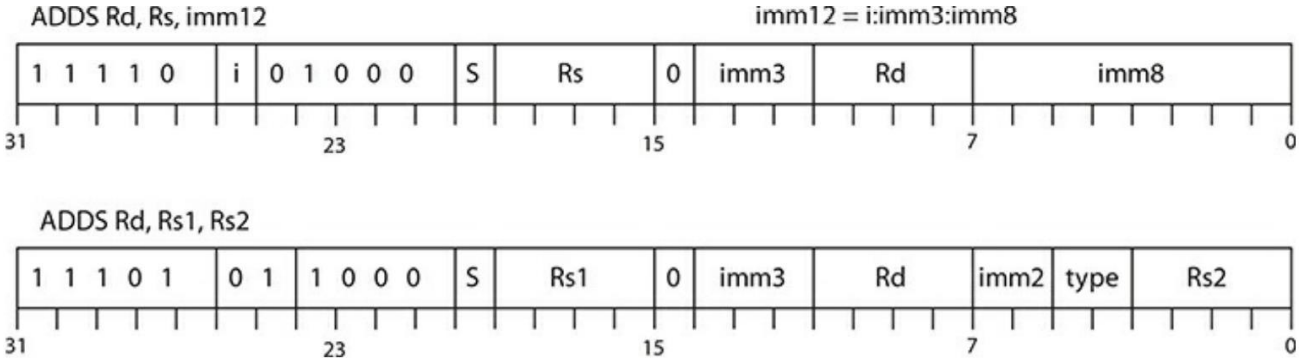


32-Bit Instruction Encoding

INSTRUCTION ENCODING

- What minimum changes to these instructions will make it mandatory to use 32-bit encoding?
- Now the immediate value operand is changed from 16 to 356 which cannot be encoded in 8-bit field
- Therefor this immediate value forces the assembler to rather use 32-bit instruction encoding
- Also, if any of the operand registers belongs to the general-purpose high register group, i.e., R8-R12, it will become mandatory to use 32-bit instruction encoding

```
ADDS R2, #356       ; R2 = R2 + 356, Instruction is 32-
bit encoded
ADDS R4, R2, #14    ; R4 = R2 + 14, Instruction is 32-
bit encoded
ADDS R6, R5, R9     ; R6 = R5 + R9, Instruction is 32-
bit encoded
```



16/32-Bit Instruction Encoding

INSTRUCTION ENCODING

32-bit encoding becomes mandatory

1. for different immediate values of Operand 2
   - When source and destination registers are different, an immediate value larger than 3-bit is sufficient for an instruction to be encoded as 32-bit
   - When source and destination registers are same, an immediate value greater than either 8-bit or 12-bit is required for mandating 32-bit encoding of an instruction
2. If any high general-purpose register is used as an operand

Operand 1	Operand 2	Encoding
Register R0 to R7	Register R0 to R7	16 bit
Register R0 to R7	Immediate value is limited to 3-bit (for different) or 8/12-bit (for same) source-destination registers	16 bit
Register R8 to R12	Register R0 to R7	32 bit
Register R0 to R7	Register R8 to R12	32 bit
Register R0 to R7	Immediate value exceeds 3-bit or 8/12-bit	32 bit

Visualizing the Instruction Encoding

INSTRUCTION ENCODING

ADDS R2, #16 instruction

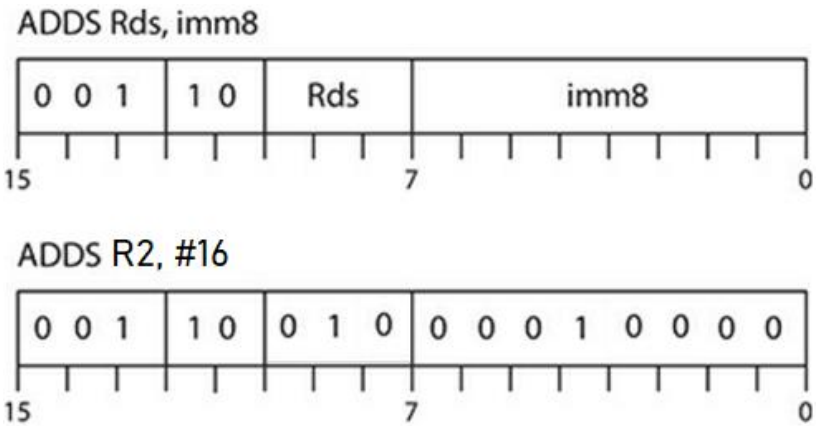
- Rds = R2 and imm8 = 16 (i.e., 0x10)
- 16-bit sequence becomes |0011 0010 0001 0000| i.e., 0x3210

Disassembly

```
0x0000026C 3210      ADDS        r2,r2,#0x10
0x0000026E 1D94      ADDS        r4,r2,#6
0x00000270 18EE      ADDS        r6,r5,r3
0x00000272 F51272B2  ADDS        r2,r2,#0x164
0x00000276 F112040E  ADDS        r4,r2,#0x0E
0x0000027A EB150609  ADDS        r6,r5,r9
```

Instructions_Encoding.s

```
17
18      ADDS R2, #16          ; R2 = R2 + 16,   instruction is 16-bit encoded
19      ADDS R4, R2, #6       ; R4 = R2 + 6,    instruction is 16-bit encoded
20      ADDS R6, R5, R3       ; R6 = R5 + R3,   instruction is 16-bit encoded
21
22      ADDS R2, #356         ; R2 = R2 + 356,  instruction is 32-bit encoded
23      ADDS R4, R2, #14      ; R4 = R2 + 14,   instruction is 32-bit encoded
24      ADDS R6, R5, R9       ; R6 = R5 + R9,   instruction is 32-bit encoded
```



THANK YOU

Any Questions???