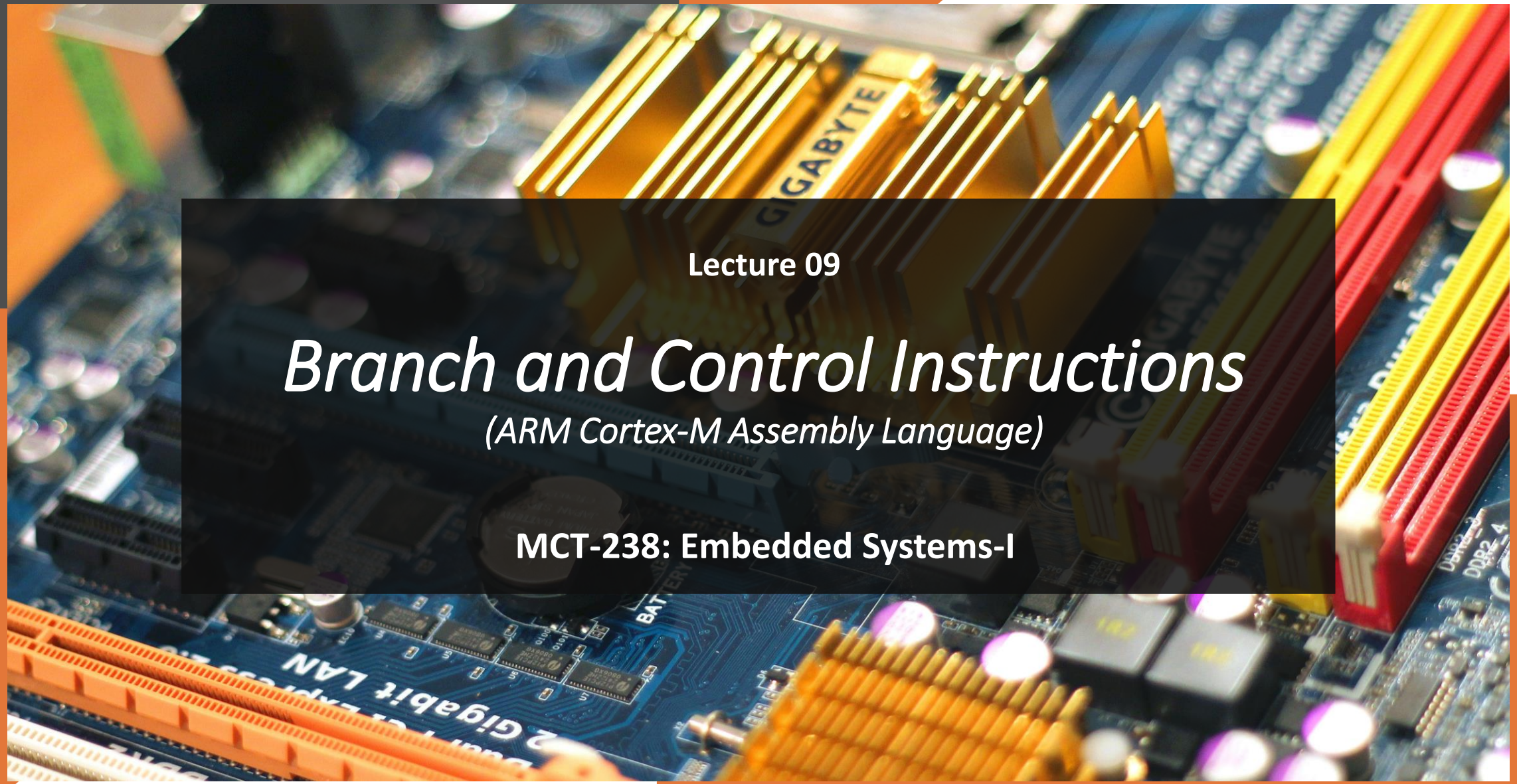
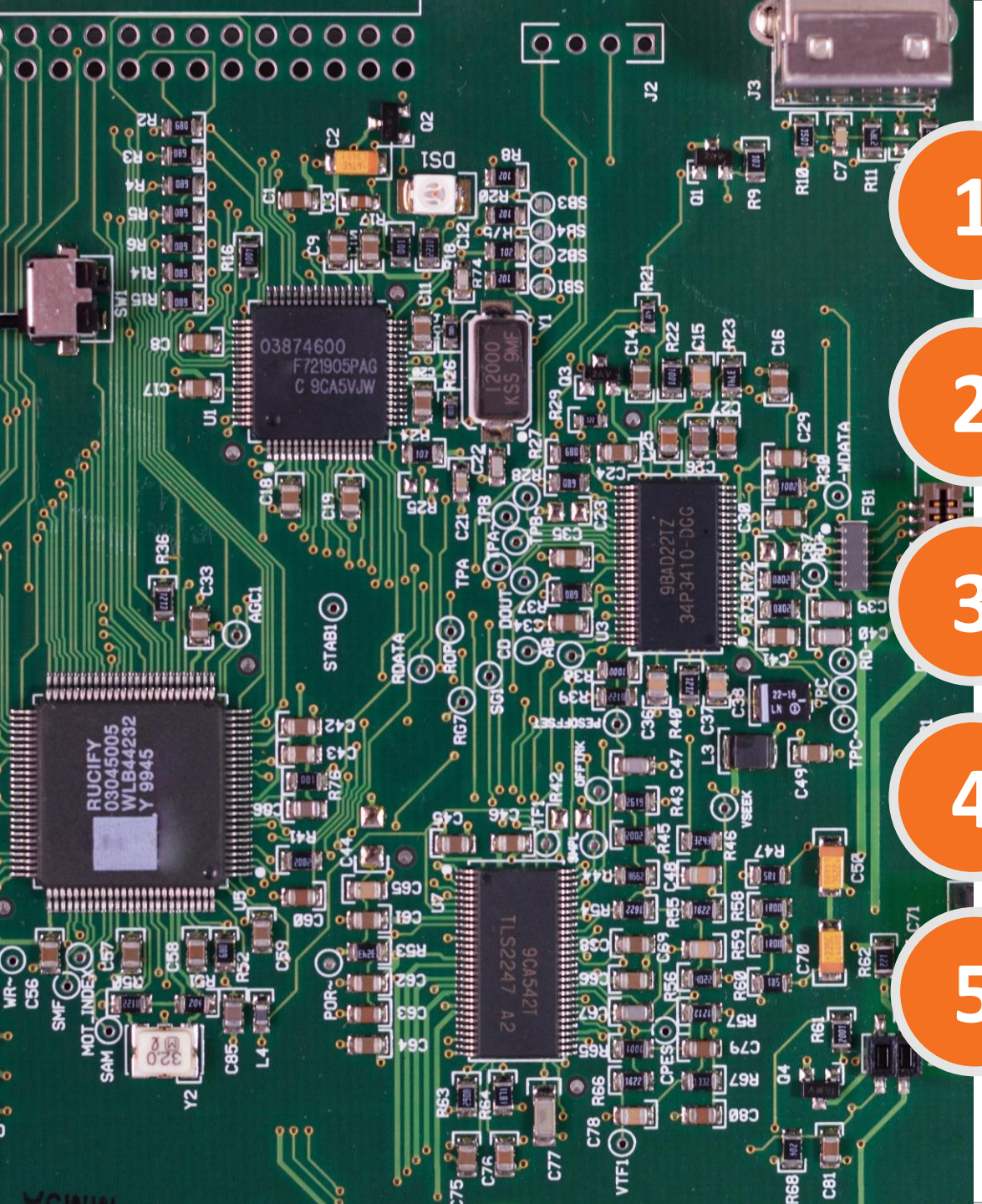


Lecture 09

Branch and Control Instructions

(ARM Cortex-M Assembly Language)

MCT-238: Embedded Systems-I



1. INTRODUCTION TO BRANCH INSTRUCTIONS
2. CONDITIONAL BRANCH EXECUTION
3. IMPLEMENTING BRANCHING STRUCTURES/FUNCTIONS
4. COMBINED COMPARE AND CONDITIONAL BRANCH
5. IF-THEN CONDITIONAL INSTRUCTION BLOCK

INTRODUCTION TO BRANCH INSTRUCTIONS

ARM Cortex-M4 Assembly Language

Overview

INTRODUCTION TO BRANCH INSTRUCTIONS

- A user program for real-time application involve decision making, conditional execution, calling subtasks or functions
- To handle these complexities, the processor architecture supports branch and control instructions to allow the software developer to control the flow of program execution
- The Cortex-M processor supports different types of branch and control instructions with varying complexity
    1. **Branch instructions** (conditional and unconditional) – simple branch instructions
    2. **Function calls** (conditional and unconditional) – requires a return address before calling the function
    3. **Combined compare and conditional branch** – implements two operations in one instruction to improve the branch operation efficiency
    4. **Conditional execution of instructions** (IF-THEN instruction) – allow to map if-else and switch-case types of higher-level conditional constructs directly to these assembly instructions
    5. **Table branch** –

BRANCH INSTRUCTIONS

ARM Cortex-M4 Assembly Language

B, BX, BL, & BLX

BRANCH INSTRUCTIONS

The most basic branch instructions, supported by Cortex-M processor, are:

Mnemonic	Brief description	Encoding	Flags
B	Branch	16 or 32 bit	-
BX	Branch indirect	16 bit	-
BL	Branch with link	32 bit	-
BLX	Branch indirect with link	16 bit	-

- The instruction **B** is also called branch immediate, and **BX** is also termed branch register
- For **BL** and **BLX** instructions, the address of the next instruction, following the branch instruction, is saved to register **LR** (link register or R14) at the time of execution of the branch instruction, which is used subsequently at the time of returning.

B, BX, BL, & BLX

BRANCH INSTRUCTIONS

- The general syntax for branch instructions is given:

$$\begin{array}{ll} \mathrm{B}\{cond\} & label \\ \mathrm{BX}\{cond\} & Rm \\ \mathrm{BL}\{cond\} & label \\ \mathrm{BLX}\{cond\} & Rm \end{array}$$

- **{cond}** is one of the condition codes.
  - When the optional condition code is omitted, these branch instructions become **unconditional branches**.
  - On the other hand, use of any of the conditional codes makes these branch instructions become **conditional branches**.
- **label** specifies the address of the instruction to which the branch must be performed.
  - The compilation tools (assembler or compiler) are responsible to evaluate the required value of the offset from this **label** to the current value of the program counter (R15 or **PC**) (i.e., the address of the current **B** instruction being executed).
- **Rm** is a register containing the address to which the branch must be performed
  - Bit 0 of the address contained in register Rm should be set to 1, which indicates that the code execution is in Thumb mode. However, the address to which the branch is performed to is obtained by resetting the bit 0 to 0.

Conditional Code Suffixes

BRANCH INSTRUCTIONS

Code Suffix	Flags	Description
EQ	Z = 1	Equal (==)
NE	Z = 0	Not equal (! =)
CS or HS	C = 1	Unsigned higher or same ($\geq$)
CC or LO	C = 0	Unsigned lower (<)
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow occurred
VC	V = 0	Overflow did not occur
HI	C = 1 and Z = 0	Unsigned higher (>)
LS	C = 0 or Z = 1	Unsigned lower or same ($\leq$)
GE	N = V	Signed greater than or equal ($\leq$)
LT	N != V	Signed less than
GT	Z = 0 and N = V	Signed greater than
LE	Z = 1 and N != V	Signed less than or equal
AL	any value	Always. This is the default setting

Branch Ranges

BRANCH INSTRUCTIONS

Instruction	Branch instruction offset range
B	-256 to 254 bytes for 16-bit encoding (outside IT block) -2 KB to +2 KB bytes for 16-bit encoding (inside IT block) -1 MB to +1 MB bytes for 32-bit encoding (outside IT block) -16 MB to +16 MB bytes for 32-bit encoding (inside IT block)
BL	-16 MB to +16 MB bytes
BX	Any arbitrary value in the register
BLX	Any arbitrary value in the register

CONDITIONAL BRANCH EXECUTION

ARM Cortex-M4 Assembly Language

Introduction

CONDITIONAL BRANCH EXECUTION

- Conditional branch instructions are used to determine whether a branch should be carried out and are implemented with the help of Cortex-M processor application program status register (APSR)
- There are four bits, namely, overflow (V), carry (C), negative (N), and the zero (Z) flags, which indicate the status of the current operation performed by the microprocessor.
- Irrespective of their category, branch instructions work depending on the current values of the flags but do not affect the flags.

Introduction

CONDITIONAL BRANCH EXECUTION

- Conditional branch instructions can be broadly categorized in three groups:

1. **Single-flag Branch instructions**
   - e.g., in making the decision whether 0x7FFF FFFF and 0x8000 0000 are equal, the microprocessor needs to take only one flag (the Z flag in this case) into account
2. **Signed Branch instructions** – used when operands are treated as signed numbers
   - e.g., let's take the case of testing whether 0x7FFF FFFF is greater than 0x8000 0000 (both signed), then 0x7FFF FFFF is greater than 0x8000 0000
3. **Unsigned Branch instructions** – used when the operands are interpreted as unsigned numbers
   - e.g., let's take the case of testing whether 0x7FFF FFFF is greater than 0x8000 0000 (both unsigned), then 0x8000 0000 is greater than 0x7FFF FFFF

- For signed/unsigned branch, we may need to consider multiple flags

Single-Flag Branch Instructions

CONDITIONAL BRANCH EXECUTION

- **BEQ**, **BNE**, **BCS**, **BCC**, **BMI**, **BPL**, **BVS**, and **BVC** are examples of single-flag branch instructions
- These single-flag instructions can be used in different scenarios for making decisions

Code Suffix	Flags	Description
EQ	Z = 1	Equal (==)
NE	Z = 0	Not equal (! =)
CS or HS	C = 1	Unsigned higher or same ($\geq$)
CC or LO	C = 0	Unsigned lower ($<$)
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow occurred
VC	V = 0	Overflow did not occur
HI	C = 1 and Z = 0	Unsigned higher ($>$)
LS	C = 0 or Z = 1	Unsigned lower or same ($\leq$)
GE	N = V	Signed greater than or equal ($\leq$)
LT	N != V	Signed less than
GT	Z = 0 and N = V	Signed greater than
LE	Z = 1 and N != V	Signed less than or equal
AL	any value	Always. This is the default setting

Single-Flag Branch Instructions

CONDITIONAL BRANCH EXECUTION

The Z-Flag Usage

- The Thumb2 instruction set uses condition codes **EQ** (equal) and **NE** (not equal) with branch instructions to check equality (==) and inequality (!=) of data objects.
- The **EQ** code results in true condition when **Z** flag is 1 while **NE** code gives true condition for **Z** = 0

```
        MOV R3, #10
loop
        SUBS R3, R3, #2
        BNE loop      ; Branch to Loop if result of previous
                      ; instruction is not equal to zero
```

Single-Flag Branch Instructions

CONDITIONAL BRANCH EXECUTION

The C-Flag Usage

- Using the BCC or BCS single-flag branch instructions, we can count the number of 1s in each number.

```
    LDR R0, = var    ; variable
    LDR R1, [R0]
    MOV R5, #32      ; Initialize the counter to 32
loop                 ; assuming a word size variable
    RORS R1, #1
    BCC skip
    ADD R4, #1       ; Increment if bit is 1
    SUBS R5, #1
    BNE loop
    B store
skip
    SUBS R5, #1
    BNE loop
store
    LDR R2, = parity     ; variable to store 1 count
    STRB R4, [R2]
stop
    B stop
```

Unsigned Conditional Branch Instructions

CONDITIONAL BRANCH EXECUTION

- The carry and zero flags are used In order to implement unsigned conditional branch
- For some comparisons, either the zero or carry flag is used, while for others their combination is used

{Instruction }	Description
BLO label	Branch to label if unsigned less than (same as BCC)
BHS label	Branch to label if unsigned greater than or equal to (same as BCS)
BLS label	Branch to label if unsigned less than or equal to
BHI label	Branch to label if unsigned greater than

- One example illustrating the use of unsigned conditional branch is when we need to differentiate characters from the standard and extended ASCII character sets

Signed Conditional Branch Instructions

CONDITIONAL BRANCH EXECUTION

- To implement the signed conditional branch, the negative, overflow, and zero flags are used

{Instruction}	Description
BLT label	Branch to label if signed less than
BGE label	Branch to label if signed greater than or equal to
BGT label	Branch to label if signed greater than
BLE label	Branch to label if signed less than or equal to

- An example illustrating the use of signed conditional branch instructions to count the negative numbers in an array of numbers

IMPLEMENTING BRANCHING STRUCTURES

ARM Cortex-M4 Assembly Language

Introduction

IMPLEMENTING BRANCHING STRUCTURES

- The execution flow of a program can be altered using branching structures.
- Let's discuss the following three basic branching structures.
  1. If-else branching
  2. Loop based branching
  3. Switch-case based branching
- In the following we discuss these three basic branching structures along with their different variants and examples.

If-else Branching

IMPLEMENTING BRANCHING STRUCTURES

Implementing *if*

- The *if* branching structure tests a condition and based on the result of condition testing, a set of instructions is executed.
- If the condition is true, the instructions inside the *if* block are executed

```
// verify that the variable is in range
if(('a' <= x) && (x <= 'z'))
  {
    x = x - 32;
  }
```

```
    CMP R0, #'a' ; Less than 'a' condition is tested
    BLT stop
    CMP R0, #'z' ; Greater than 'z' condition is tested
    BGT stop
    SUB R0, #0x20
stop
    B stop
```

If-else Branching

IMPLEMENTING BRANCHING STRUCTURES

Implementing *if-else*

- In case of if-else structure, there is a condition to test, while there are two different instruction blocks.
- When the condition is true, the set of instructions corresponding to if block is executed, otherwise the instructions inside the else block are executed.
- Omitting the else block reduces the if-else structure to if block.

```
if((x%2) ==0) // if remainder is zero, then even
  y = 'E';
else
  y = 'O';    // otherwise it is odd
```

```
    TST R0, #0x1 ; Test if LSB of R0 is 1
    BEQ even
    MOV R1, #'O'
    B stop
even
    MOV R1, #'E'
stop
    B stop
```

Loop Based Branching

IMPLEMENTING BRANCHING STRUCTURES

Implementing *for* Loop

- Looping structures allow a program to execute a set of instructions for a number of times
- A *for* loop is used whenever we know in advance the number of times a set of instructions should be executed

```
for(i = 0; i <50; i ++)
  Array [i] = '*';
```

```
        MOV R0, #'*'
        LDR R2, = Array
        MOV R1, #0
loop
        CMP R1, #50          ; Check for loop termination
        BGE stop             ; Terminate the loop if count is 50
        STRB R0, [R2], #1 ; Store '*' to the current address
        ADD R1, #1
        B loop
stop
        B stop
```

Loop Based Branching

IMPLEMENTING BRANCHING STRUCTURES

Implementing *while* Loop

- When the number of loop iterations is not known in advance, we use *while* loop
- *while* loop is iterated until a certain condition is fulfilled

```
int count = 0;
int i = 0;
// Test the condition for loop
while( ARRAY [i] != '\n')
  count ++;
```

```
    LDR R2, = ARRAY
    MOV R1, #0        ; initialize the count to 0
    LDRB R0, [R2], #1
loop
    CMP R0, #0x0A     ; check loop termination condition
    BEQ stop
    ADD R1, #1
    LDRB R0, [R2], #1
    B loop
stop
    B stop
```

Switch-case Based Branching

IMPLEMENTING BRANCHING STRUCTURES

- The condition testing can be performed for multiple possible values, resulting in multiway branching.
- For instance, a variable can be tested for three different possible values and correspondingly one of the three different code segments is executed.
- This situation can be implemented using switch-case conditional structure
- When the switch-case construct has multiple cases, it requires multiple test or compare operations followed by conditional branch instructions but this type of implementation will execute slowly if the number of cases is large

```
switch(x){ // Switch construct with three cases
  case ' <0':
    y = -1;
  case '==0':
    y = 0;
  case ' >0': // Note there is no default case
    y = 1;
}
```

```
        CMP R0, #0
        BLT negative
        BEQ zero
        BGT positive
negative
        MOV R1, -1
        B stop
zero
        MOV R1, 0
        B stop
positive
        MOV R1, 1
stop
        B stop
```

IMPLEMENTING FUNCTIONS

ARM Cortex-M4 Assembly Language

Simple Function Call

IMPLEMENTING FUNCTIONS

- When **BL** and **BLX** instructions are used, the address of the next instruction following the branch instruction is saved to LR register (the link register or R14)
- This address is used at the time of returning from the called function.
- The return from the function can be implemented using instruction **BX LR**, which causes program control to return to the calling process.
- Note that **BL** or **BLX** instructions are not used when returning from a function or subroutine

```
    AREA MYCODE, CODE, READONLY
    EXPORT __main
    ENTRY
__main
    MOV R0, #0x18
    MOV R1, R0
    BL addition ; Branch with link to 'addition'
    LDR R3, = addition
    BLX R3              ; Branch indirect with link
    SUB R2, R1, R2
    B stop              ; Branch to 'stop'
addition
    ADD R2, R1, R0 ; R2 = R1 + R0
    BX LR               ; Branch with link
stop
    B stop
    END
```

Nested Function Calls

IMPLEMENTING FUNCTIONS

```
        AREA MYCODE , CODE , READONLY
        EXPORT __main ; the first instruction to call
        ENTRY

__main
        MOV R0, #0x18
        MOV R1, R0
        BL multiply_add ; Branch with link to 'multiply_add '
        SUB R2, R1, R2
        B stop

multiply_add
        PUSH {LR}    ; First preserve the return address
        BL addition ; Call another function
        MUL R2, R2, R0
        POP {LR}     ; Retrieve return address from stack
        BX LR        ; Branch indirect with link

addition
        PUSH {LR}
        ADD R2, R1, R0 ; R2 = R1 + R0
        POP {PC}

stop
        B stop
        END
```

Implementing Branch Operations Indirectly

IMPLEMENTING FUNCTIONS

- Other assembly programming instructions can be used to implement the branch operation indirectly
- Example below illustrate indirect branch operation using MOV, LDR instructions

```
MOV R15, R0      ; Branch to an address in R0
LDR R15, [R0]    ; Branch to an address in memory
                 ; specified by R0
POP {R15}        ; Do a stack pop operation to change
                 ; the program counter .
```

- The **POP PC** instruction is used to implement the return operation from a function call, which conventionally would have been implemented using a BX LR instruction
- Some other assembly programming instructions, e.g., ADD and SUB, can also update the PC register. However, it is recommended that these instructions should not be used to create a branch operation

Recursive Function Call

IMPLEMENTING FUNCTIONS

```
unsigned long Fact(unsigned long n)
{
  if(n <=1)
  {
    return 1;
  }
  return n*Fact(n -1);
}
```

```
    AREA Mydata, DATA, READWRITE
Result DCD 0

    ; Separate memory segment for function call
    AREA MyFunction, CODE , READONLY
Func1
    PUSH {LR}          ; Assume sufficient stack size
    CMP R0, #1         ; is available
    BEQ Done
    PUSH {R0}
    SUB R0, R0, #1
    BL Func1
    POP {R1}
    MUL R0, R0, R1
Done
    POP {LR}
    BX LR

    ; New memory segment for main function
    AREA MyMain, CODE, READONLY
    EXPORT __main      ; Entry point of user program
                       ; and is called from startup
__main                 ; file
    MOV R0, #4
    BL Func1
    LDR R1, = Result
    STR R0, [R1]
Stop
    B Stop
    END
```

Passing Parameters to Functions

IMPLEMENTING FUNCTIONS

- In high level languages, there are two well-known mechanisms for parameter passing to functions:
  1. **Call by Value**: the actual data values are passed to the called function
  2. **Call by Reference**: the address of the data is passed to the function. Particularly useful when dealing with data arrays.
- Unlike high-level languages, assembly language functions do not have associated parameter lists due to which it is up to the programmer to devise strategies for passing parameters to functions

Passing Parameters to Functions

IMPLEMENTING FUNCTIONS

Call by Value

- For R4 = M + N - R3, where M and N are 32-bit data.
- Addition function implemented and the operands are passed through the call by value method

```
    AREA DATA1, READONLY
M DCD 5
N DCD 3
    AREA MYCODE, CODE, READONLY
    EXPORT __main ; the first instruction to call
    ENTRY
__main
    LDR R6, =M
    LDR R7, =N
    LDR R0, [R6]
    LDR R1, [R7]
    BL addition ; Branch to function 'addition ' and
                ; pass R0 and R1 as parameters
    MOV R3, #0x3
    SUB R4, R1, R3
    B stop
addition
    ADD R1, R1, R0 ; R1 = R1 + R0
    BX LR
stop
    B stop
    END
```

Passing Parameters to Functions

IMPLEMENTING FUNCTIONS

Call by Reference

- For R4 = M + N - R3, where M and N are 32-bit data.
- Addition function implemented and the operands are passed through the call by reference method

```
     AREA DATA1, READONLY
M DCD 5
N DCD 3
     AREA MYCODE, CODE, READONLY
     EXPORT __main
     ENTRY
__main
     LDR R6, =M
     LDR R7, =N
     BL addition ; Branch to function 'addition'
     MOV R3, #0x3
     SUB R4, R1, R3
     B stop
addition
     LDR R0, [R6]
     LDR R1, [R7]
     ADD R1, R1 , R0 ; R1 = R1 + R0
     BX LR
stop
     B stop
     END
```

COMBINED COMPARE AND CONDITIONAL BRANCH

ARM Cortex-M4 Assembly Language

CBZ & CBNZ Instructions

COMBINED COMPARE AND CONDITIONAL BRANCH

```
CBZ    Rn, label
CBNZ   Rn, label
```

- **CBZ** and **CBNZ** performs comparison with zero and branch conditionally

Mnemonic	Brief description	Encoding	Flags
CBZ	Compare and branch if zero	16 bit	-
CBNZ	Compare and branch if not zero	16 bit	-

- **CBZ** instruction is equivalent to two consecutive instructions, **CMP** (compare with zero) followed by conditional branch **BEQ**
- Similarly, CBNZ is equivalent to **CMP** with zero followed by **BNE**
- **APSR** condition flags values are not affected by the **CBZ** and **CBNZ** instructions
- The compare and branch instructions only support forward branches

CBZ & CBNZ Instructions

COMBINED COMPARE AND CONDITIONAL BRANCH

CBZ Example

```
// Iterative function calling
i=6;
while (i != 0 ){
   funcA ();      // Call a function
   i--;
}
```

```
     MOV R0, #6            ; Set loop counter
loop1
     CBZ R0, loop1exit     ; If loop counter = 0 then exit loop
     BL funcA              ; Call a function
     SUB R0, #1            ; Loop counter decrement
     B loop1               ; Next loop iteration
loop1exit
```

CBZ & CBNZ Instructions

COMBINED COMPARE AND CONDITIONAL BRANCH

CBNZ Example

```
status = strchr ( emailid , '@ ');
// if emailid does not contain @, then status is 0.
if ( status == 0){
  generate_error_message ();
}

...
    CBNZ R0 , email_id_ok    ; R0 contains a character from
                             ; the emailid string .
    BNE generate_error_message
email_id_ok
...
```

IF-THEN CONDITIONAL INSTRUCTION BLOCK

ARM Cortex-M4 Assembly Language

What is If-Then Block?

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- Apart from the conditional branch instructions ARM Cortex-M architecture extends the use of condition codes to other instructions as well
  - e.g., ADD and SUB instructions can be executed conditionally by using the optional condition code
- Conditional execution of an instruction is implemented by Thumb2 ISA using **If-Then** (IT) block.
  - The IT instructions allow up to four succeeding instructions to be conditionally executed and they collectively form an **IT block**.
  - The IT block instruction is very useful for handling small conditional codes.
  - It avoids branch penalties because there is no change to program flow.
  - Conditional instructions, except for conditional branches, must be inside an IT instruction block.

Syntax

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- The syntax of IT instruction is

$$\mathrm{IT}\{x\{y\{z\}\}\} \quad cond$$

- ***x***, ***y***, and ***z*** are the optional conditional execution switches for second, third, and fourth instructions in the IT block
- ***cond*** is the base condition for the IT instruction block. The first instruction following IT instruction is executed if the **cond** is true
- Each of the optional condition switches ***x***, ***y***, and ***z*** can be either **T** (THEN) or **E** (ELSE)
  - When condition switch **T** is used, then ***cond*** is applied to the corresponding instruction
  - The use of condition switch **E** applies the inverse ***cond*** to the corresponding instruction

Mnemonic	Brief description	Encoding	Flags
IT	If-then conditional instruction	16 bit	-

Syntax

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- In IT instruction blocks, the first instruction must be the IT instruction itself, detailing the choice of condition switches along with the condition it checks.
- The first conditional instruction after the IT instruction must be TRUE.
- The condition codes for second through fourth instructions following the IT instruction can be either TRUE or FALSE.
- The use of condition code with an instruction is specified with a two-letter suffix, such as EQ or CC, appended to the instruction mnemonic.
- The conditional instructions following the IT instruction in the IT block must specify either the condition code ***cond*** or its logical inverse as part of their instruction syntax

Conditional Code Suffixes

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- The ***cond*** operand in IT instruction uses the same condition codes as conditional branch
- It is possible to use AL (the always condition) for ***cond*** in an IT instruction.
- If this is done, all of the instructions in the IT block must be unconditional, and each of ***x***, ***y***, and ***z*** must be T or omitted but not E.

Code Suffix	Flags	Description
EQ	Z = 1	Equal (==)
NE	Z = 0	Not equal (! =)
CS or HS	C = 1	Unsigned higher or same ($\geq$)
CC or LO	C = 0	Unsigned lower (<)
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow occurred
VC	V = 0	Overflow did not occur
HI	C = 1 and Z = 0	Unsigned higher (>)
LS	C = 0 or Z = 1	Unsigned lower or same ($\leq$)
GE	N = V	Signed greater than or equal ($\leq$)
LT	N != V	Signed less than
GT	Z = 0 and N = V	Signed greater than
LE	Z = 1 and N != V	Signed less than or equal
AL	any value	Always. This is the default setting

How Condition Code works?

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- The condition code suffix appended to an instruction inside an IT block requires the processor to test the condition code based on the status of the flags.
- If the condition test of a conditional instruction in IT block fails, the instruction
  - Does not execute
  - Does not write any value to its destination register
  - Does not affect any of the flags
  - Does not generate any exception
- This feature often eliminates the need to branch, avoiding pipeline stalls and results in an improved execution performance and can also increase code density

How Condition Code works?

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- By default the data processing instructions do not affect the condition code flags but can be made to by suffixing S.
- On the other hand, the comparison instructions, e.g., CMP, TST, do affect the flags implicitly.
- The execution of an instruction in IT block is conditionally dependent on the status of the condition flags, which are updated by a priorly executed instruction.
- The prior instruction that has updated the flags can either be the immediately preceding instruction that updated the flags, or there may be an arbitrary number of intermediate instructions that did not update the flags

Examples

IF-THEN CONDITIONAL INSTRUCTION BLOCK

IF Only Block

- 3 Instructions

```
CMP R0, R1              ; Compare parameters loaded to R0
                        ; with R1
ITTT LT                 ; If R0 < R1
ADDLT R2, R0, R1        ; add the two operands
MULLT R3, R2, R0        ; multiply the sum with first
                        ; operand in R0
ASRLT R3, R3, #1        ; divide the result by 2
```

IF-ELSE Block

- 4 instructions

```
CMP R0, R1              ; Compare parameters
                        ; loaded to R0 with R1
ITTEE LT                ; If R0 < R1
ADDLT R2, R0, R1        ; part of IF
MULLT R3, R2, R0        ; part of IF
SUBGE R2, R0, R1        ; part of ELSE
ASRGE R3, R3, #1        ; part of ELSE
```

Examples

IF-THEN CONDITIONAL INSTRUCTION BLOCK

Implementing if-else using IT block

```
if (x < 0){
   y = -x;
}
else{
   y = x;
}
```

```
CMP R0, #0          ; Compare R0 with 0
ITE LT              ; Condition to check if R0 is negative
RSBLT R3, R0, #0    ; take absolute value of negative number
MOVGE R3, R0        ; do nothing and return original value
```

Examples

IF-THEN CONDITIONAL INSTRUCTION BLOCK

Implementing nested if-else using IT block

```
if(R0 > R1){
  if(R1 > R2){
    if(R2 > R3){
      R4 = 0x123;
    }
  }
}
```

```
CMP R0, R1          ; Compare R0 with R1
ITTT GT             ; Condition to check if R0 > R1
SUBSGT R5, R1, R2   ; check if R1 > R2
SUBSGT R5, R2, R3   ; check if R2 > R3
MOVGT R4, #0x123
```

Instruction Execution

IF-THEN CONDITIONAL INSTRUCTION BLOCK

Data Processing Instructions inside IT Block

- It is important to note that data processing instructions, encoded using 16-bit encoding, do not update APSR when they are used inside an IT instruction block.
- If the suffix S is added to the conditionally executed instructions inside the IT block, then 32-bit encoding of the instruction would be used by the assembler.

Interrupt Handling while inside IT Block

- In case an exception or interrupt has happened, while the processor was executing an IT instruction block, the execution status for that IT block is stored to the stacked **xPSR** register (specifically in the IT/Interrupt-Continuable Instruction [ICI] bit field).
- Once the execution of the interrupt service routine has been completed, the IT block execution is resumed and the remaining instructions of the block can continue the execution.
- In case of multi-cycle instructions (for example, multiple load and store) inside an IT block, if an exception occurs during its execution, the whole instruction is abandoned and restarted after the interrupt process is completed.

Rules for IT Block Construction

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- A branch or any instruction that modifies the PC must either be outside an IT block or must be the last instruction inside the IT block.
- Do not branch to any instruction inside an IT block, except when returning from an exception handler
- All conditional instructions except conditional branches must be inside an IT block. Conditional branch instructions can be the last instructions of an IT block.
- The conditional branch instructions have a larger branch range when they are used inside an IT block.
- Each instruction inside the IT block must specify a condition code suffix that is either the same or logical inverse of the condition code used by the IT instruction.
- The first conditional instruction inside the IT block should always use the condition code the same as the one used by the IT instruction itself.

IT Instruction Advantages Illustration

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- Conditional execution based on IT block can provide both the **code density** as well as **execution speed improvement**
- To validate these advantages, Greatest Common Divisor (GCD) Algorithm, proposed by Euclid, is implemented for both with and without IT Instructions

```
// GCD implementation based
// on Euclid algorithm
int gcd( int a, int b){
  while (a != b){
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

```
; Euclid GCD algorithm
; without IT Instructions
gcd
    CMP R0, R1
    BEQ stop
    BLT less
    SUBS R0, R0, R1
    B gcd
less
    SUBS R1, R1, R0
    B gcd
stop
```

```
; Euclid GCD algorithm
; with IT Instructions
gcd
    CMP R0, R1
    ITE GT
    SUBGT R0, R0, R1
    SUBLE R1, R1, R0
    BNE gcd
```

IT Instruction Advantages Illustration

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- It is important to note that branch instructions for Cortex-M3/M4 based processors can take from 2 to 4 cycles for their execution, depending on the alignment and width of the target instruction.
- For the micro-controller used in this class, it takes **3 cycles** for branch execution.
- On the other hand, many of the data processing instructions require **one cycle** for their execution.
- The extra cycles required for the branch operation are attributed to the fact that branch instructions require refilling of the pipeline.
- In case of conditional branch, if the condition is false then branch does not occur and in this case the instruction only takes **one cycle** for its execution

IT Instruction Advantages Illustration

IF-THEN CONDITIONAL INSTRUCTION BLOCK

- The execution performance for the GCD algorithm is superior for the IT block-based implementation compared to the conditional branch-based implementation
- **Euclid GCD Algorithm without IT Instructions**
  - Seven 16-bits instructions (14 bytes of ROM used)
- **Euclid GCD Algorithm with IT Instructions**
  - Five 16-bits instructions (10 bytes of ROM used)

```
; Euclid GCD algorithm
; without IT Instructions
gcd
     CMP R0, R1
     BEQ stop
     BLT less
     SUBS R0, R0, R1
     B gcd
less
     SUBS R1, R1, R0
     B gcd
stop
```

```
; Euclid GCD algorithm
; with IT Instructions
gcd
     CMP R0, R1
     ITE GT
     SUBGT R0, R0, R1
     SUBLE R1, R1, R0
     BNE gcd
```

	Conditional Branch	IT block
R0 = 1, R1 = 2	13 cycles	12 cycles
R0 = 10, R1 = 24	45 cycles	40 cycles

THANK YOU

Any Questions???