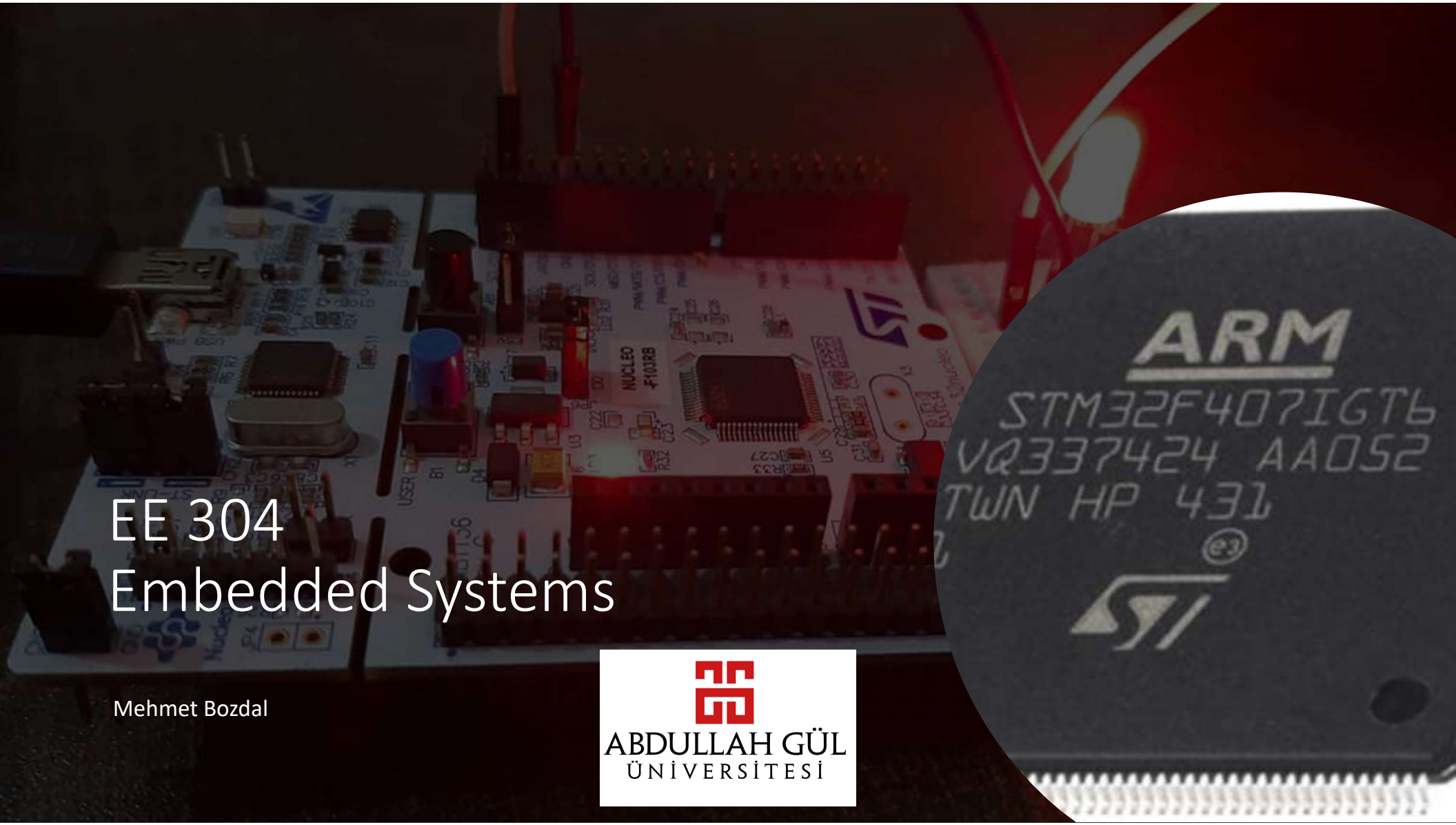

EE 304
Embedded Systems

Mehmet Bozdal

ABDULLAH GÜL
ÜNİVERSİTESİ

Foundational control structures

- Sequence
- Selection
- Iteration

It has been mathematically proven that any program can be written by using only these three control structures.

Control structures: Sequence

- Sequence is the basic control structure where instructions are executed one after the other in a linear, sequential order.

- Imagine a simple program that calculates the sum of two numbers. It would follow a sequence like this:
  - Accept input (e.g., user-entered numbers).
  - Perform the addition operation.
  - Display the result.

Control structures: Selection

- Selection allows programs to make decisions based on certain conditions, often implemented using "if" statements.
  - If-else statements
  - Switch-case statements

- Ex. Consider a program that checks whether a user is eligible for a discount. It may use an "if" statement to evaluate the user's age and apply a discount if they meet the condition.

Control structures: Iteration

- Iteration allows programs to repeat a set of instructions multiple times based on a specific condition.
  - "for" loop: Iterate over a range of values.
  - "while" loop: Continue until a condition is met.
  - "do-while" loop: Execute at least once, then check the condition.

- Think of a program that calculates the sum of numbers in a list. It may use a "for" loop to iterate over the list, adding up the values.

Combining Control Structures

- In real-world programming, solving complex problems often requires a combination of control structures. By combining sequence, selection, and iteration, we can address a wide range of scenarios effectively.

- BUT, assembly does not directly support selection and loop structures.

Control Flow Alteration

1. Branch Instructions: They enable the program to change the sequence of execution by jumping to different parts of the code, often based on conditions or specific addresses.

2. Conditional Execution: Conditional execution involves making decisions within the program, allowing different code paths to be executed based on specified conditions.

3. Calling a Subroutine: Subroutines are reusable blocks of code that can be called from various parts of the program to promote modularity and code reusability.

4. Interrupts: They provide a mechanism for external devices or events to temporarily pause the program's execution, enabling the handling of real-time events or management of hardware peripherals.

Control Flow Alteration: Branching

Unconditional Branch: Unconditional branch instructions allow the program to transfer control to a specified memory address or label without evaluating any conditions. They are typically used to implement jumps, loops, and function calls. For example, the B (Branch) instruction is an unconditional branch instruction.

```
B some_label  ;Unconditionally jump to the address of 'some_label'
```

Control Flow Alteration: Branching

Conditional Branch: Conditional branch instructions, on the other hand, depend on the evaluation of specific conditions. They change the program's control flow only if certain conditions are met. These instructions are crucial for implementing decision-making logic like conditional statements (e.g., if-else constructs).

BEQ equal_label ;Branch to 'equal_label' if the zero flag is set (equals condition)

Control Flow Alteration: Branching

B loopA ; Branch to loopA - Unconditional branch

BL funC ; Branch with link (Call) to function funC, return address stored in LR

BX LR ; Return from function call

BLX R0 ; Branch with link and exchange (Call) to a address stored in R0

BEQ labelD; Conditionally branch to labelD; if last flag setting instruction set the Z flag, else do not branch.

Control Flow Alteration: Conditional Execution

Selectively execute instructions only when certain conditions are met.

- Conditional execution suffixes, such as EQ (equal), NE (not equal), LT (less than), GT (greater than), etc. These suffixes specify the condition under which the instruction should be executed.

  ADDEQ R1, R2, R3 ; Add R2 and R3 and store the result in R1 if the equal condition is met.

Designing a loop

```
loop_start
    cmp r0, #10         ;Check the loop condition (10 iterations)
    beq loop_end        ;Exit if the counter equals 10
    .
    .
    .
    add r0, r0, #1      ;Increment the counter
    b loop_start        ;Unconditionally jump back to the loop start
loop_end
```

Subroutine Calls

Subroutines Simplify Complex Tasks

- Divide complex tasks into smaller, manageable subtasks.
- Enhance design and debugging processes.
- Facilitate development by allowing independent testing of subtasks.

Code Reuse and Efficiency

- Enable code reuse, saving time and effort.
- Eliminate redundancy in development.
- Enhance program efficiency.

Subroutine Calls

```
; Sum function
sum   PROC
      ADD R0, R0, R1
      BX LR          ; Return from subroutine
      ENDP
```

Subroutine Calls

- The **BL (Branch with Link)** instruction in ARM assembly performs two primary operations:
- It configures the link register (LR) to hold the memory address of the next instruction immediately after the BL instruction. This enables the program to return to the correct location once the subroutine is complete.
- It adjusts the program counter (PC) to point to the memory address of the very first instruction within the subroutine. This facilitates the execution of the subroutine.

Subroutine Calls: Return

- **Branch and Exchange Instruction (BX LR):** In this approach, the "BX LR" instruction is used to transfer control back to the calling code by jumping to the address stored in the link register (LR). It is a clean and straightforward way to return from a subroutine.

- **Popping LR from the Stack into PC**: Alternatively, if the subroutine has pushed the LR onto the stack using a "PUSH" instruction, you can return by popping the value from the stack into the program counter (PC).

Stack

- A stack is a region of memory used to store and manage data in a specific order, and it is essential for managing function calls and maintaining program execution.
- The stack pointer (SP) holds the memory address of the top of the stack.
- It follows the Last-In, First-Out (LIFO) principle.
- A stack is often used for:
  - Function Calls
  - Local Variables
  - Return Addresses
  - Parameter Passing

Stack – PUSH, POP

PUSH (store into stack) and POP (load from stack)

- **`PUSH {reglist}`**
- **`POP {reglist}`**
- **`PUSH {reglist, LR}`**
- **`POP {reglist, PC}`**

- PUSH {R0,R4-R7} ; Push R0,R4,R5,R6,R7 onto the stack
- PUSH {R2,LR} ; Push R2 and the link-register onto the stack
- POP {R0,R6,PC} ; Pop r0,r6 and PC from the stack, then branch to the new PC.

Stack

What is the value of R0?

```
MOV R0, R1
ADD R0, #1
BL myFunction
ADD R0, #1
```

Subroutine Calls: Registers

Register	Purpose/Use	Preservation Requirement
R0 - R3	Scratch registers for general use	Not required to be preserved by subroutines
R4 - R11	General-purpose registers	Must be preserved by subroutines that modify their values
R12 (IP)	Intra-Procedure Call Register	Not recommended for critical values, should be used cautiously
R13 (SP)	Stack Pointer Register (SP)	Must be preserved by subroutines that modify its value
R14 (LR)	Link Register	Not required to be preserved by subroutines, but calling subroutine may need to save it when calling another subroutine
R9	Platform-dependent; often used for local variables	Usage is platform-dependent

Subroutine Calls: Data Exchange

Passing Data

Number of Arguments	**Registers**
Up to 4	r0, r1, r2, r3
2 (64-bit each)	[r0, r1], [r2, r3]
1 (128-bit)	r0, r1, r2, r3
More than 4	r0, r1, r2, r3, Stack

Returning Data

Return Value Size	**Registers Used**
32 bits	r0
64 bits	r0, r1
128 bits	r0, r1, r2, r3

Subroutine Calls: Call by Value or Reference

Passing by Value

- The value of the variable is loaded from memory into register R0.
- While the subroutine modifies the value in register R0, the value stored in memory remains **unchanged**.
- Passing by value means the subroutine cannot directly modify the in-memory variable, as it operates on a copy of the value.

Passing by Reference

- The memory address of the variable is loaded into register R0 and passed to the subroutine.
- Register R0 now acts as a pointer to the variable.
- Since the memory address is sent to the subroutine, the subroutine can change the value of the variable using store instructions (STR).
- When the subroutine exits, the value of the caller's variable in memory has been changed.

Q & A