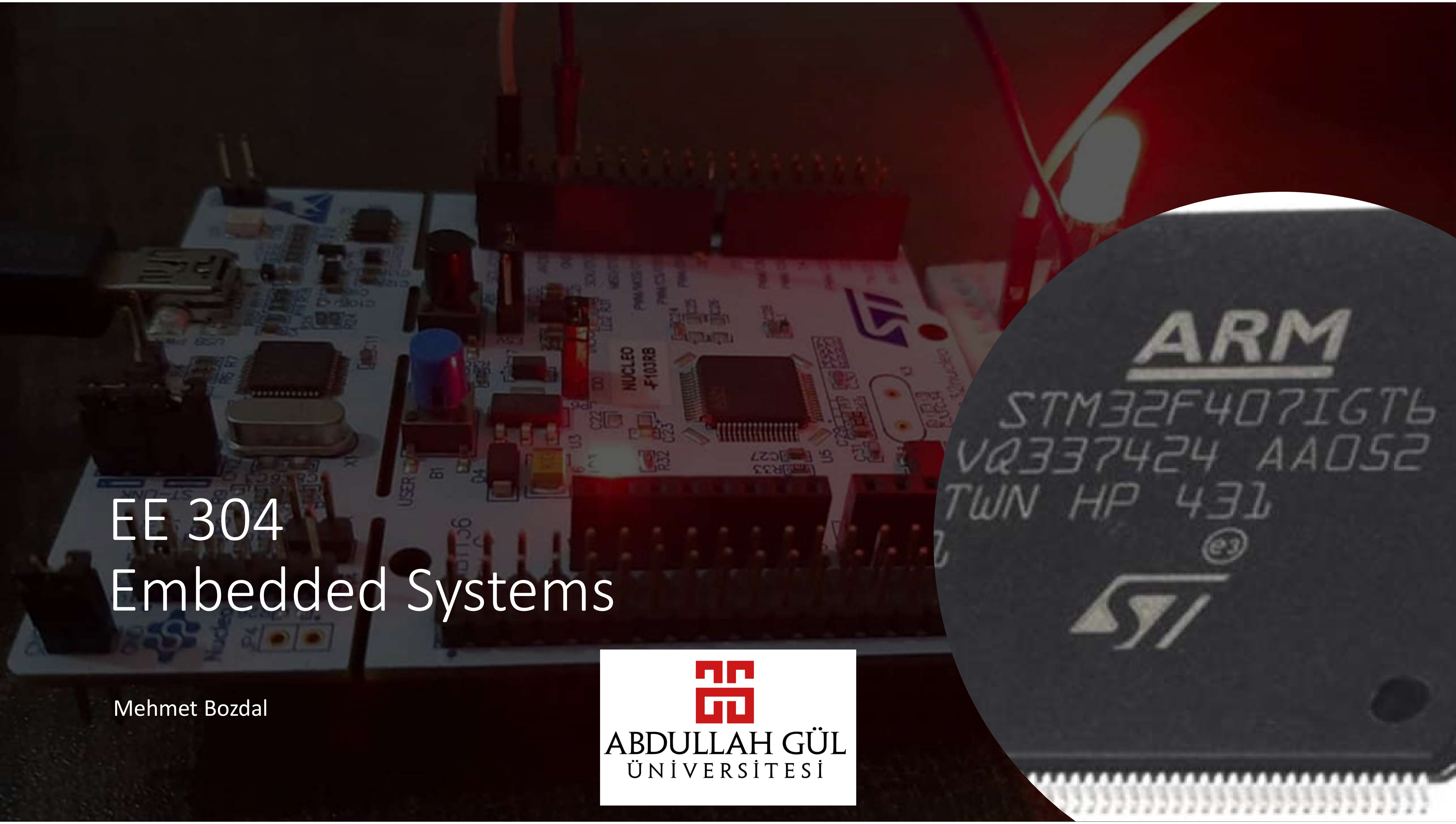


EE 304
Embedded Systems

Mehmet Bozdal

ABDULLAH GÜL
ÜNİVERSİTESİ

Agenda

- Assembly Syntax
    - Instructions
- Addressing Modes
- Subroutines

Laboratory

- Testing the board
- Submission (during the lab session)

Operation Types

1. Arithmetic & logical operations
   - ADD,SUB, MUL,DIV
     - ADD r1,r3,r7; r1=r3+r7
   - AND,LSL
2. Data Movement
3. Memory access (load/store)
   LDR r1, [r0];
4. Control operations
   - BEQ isEqual; if z=1 branch to label isEqual
5. Special instructions
   - NOP (no operation)
   - Enable interrupts
     - CPSIE I; enable interrupts
     - CPSID I; disable interrupts

Arithmetic Operations

- ADD (Addition)
- SUB (Subtraction)
- MUL (Multiplication)
- DIV (Division)

Arithmetic Operations – ADD/SUB

ADD{cond} {Rd,} Rn, #imm12

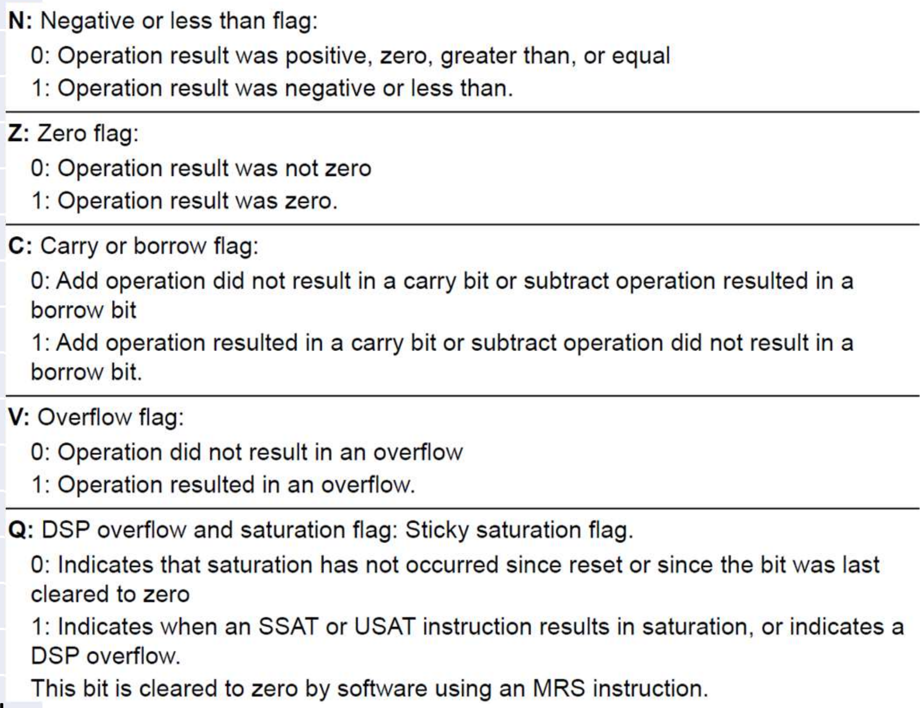
```
ADD   R0,#1; R0=R0+1
ADD   R0,R1,#10; R0=R1+10
ADDGE R5,#100; if N==V then R5=R5+100
ADDEQ R12,R1,#100; if Z=1 then R12=R1+100
```

HW: Subtract two 96-bit integers

Condition Code Suffixes

ADD{cond} {Rd,} Rn, #imm12

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned ≥
CC or LO	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N ≠ V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N ≠ V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix specified



N: Negative or less than flag:
0: Operation result was positive, zero, greater than, or equal
1: Operation result was negative or less than.

Z: Zero flag:
0: Operation result was not zero
1: Operation result was zero.

C: Carry or borrow flag:
0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit
1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.

V: Overflow flag:
0: Operation did not result in an overflow
1: Operation resulted in an overflow.

Q: DSP overflow and saturation flag: Sticky saturation flag.
0: Indicates that saturation has not occurred since reset or since the bit was last cleared to zero
1: Indicates when an SSAT or USAT instruction results in saturation, or indicates a DSP overflow.
This bit is cleared to zero by software using an MRS instruction.

Arithmetic Operations – MUL/DIV

- MUL R1, R2, R3 ; signed multiply R1 = LSB32(R2*R3)
- UMUL R1, R2, R3 ; unsigned multiply R1 = LSB(R2*R3)

- MLA R1, R2, R3, R0 ; signed multiply R1= LSB32(R2*R3) + R0
- MLS R1, R2, R3, R0 ; signed multiply R1= LSB32(R2*R3) - R0

- SDIV R1, R2, R3 ; signed division R1 = R2/R3
- UDIV R1, R2, R3 ; unsigned division R1 = R2/R3

Arithmetic Operations – MUL/DIV

- UMULL R0, R1, R2, R3 ; unsigned R1:R0 = R2*R3
- SMULL R0, R1, R2, R3 ; unsigned R1:R0 = R2*R3

AGÜ

Logic Operations

- **AND:** Logical AND operation.
- **ORR:** Logical OR operation.
- **EOR:** Exclusive OR (XOR) operation.
- **ORN:** NOT OR operation.
- **BIC (Bit Clear):** Bitwise clear operation.
- **BFC (Bit Field Clear):** Clear a specified bit field.
- **BFI (Bit Field Insert):** Insert a bit field from Rn into Rd.
- **MVN (Logical NOT):** Logical NOT operation.

Logic Operations

- These instructions operate at the bit level, performing logic operations for each pair of bits at the same position in the inputs.

		Operation				
		AND	**ORR**	**EOR**	**BIC**	**ORN**
A	B	A&B	A|B	A^B	A&(~B)	A|(~B)
0	0	0	0	0	0	1
0	1	0	1	1	0	0
1	0	0	1	1	1	1
1	1	1	1	0	0	1

AGÜ

Shift Operations

- Register shift operations move the bits in a register left or right by a specified number of bits, called the shift length.

- Arithmetic shift right (ASR)

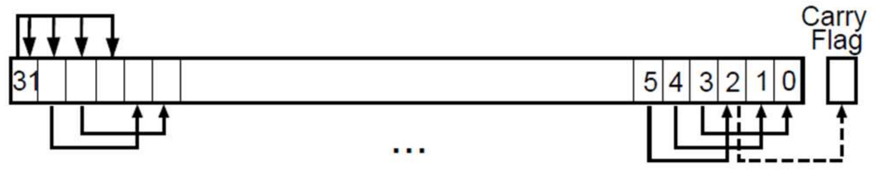


Figure 11-1 ASR #3

- Logical shift right (LSR)

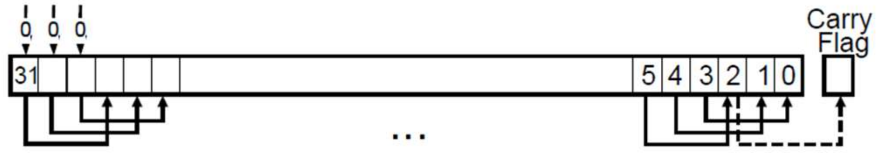


Figure 11-2 LSR #3

Shift Operations

- Logical shift left (LSL)

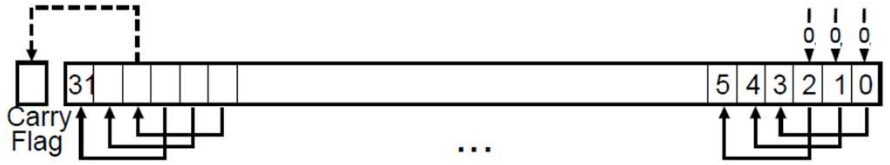


Figure 11-3 LSL #3

- Rotate right (ROR)

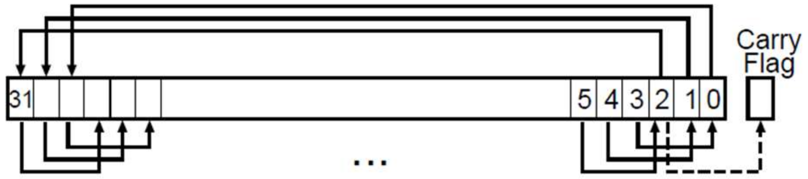


Figure 11-4 ROR #3

- Rotate right with extend (RRX)

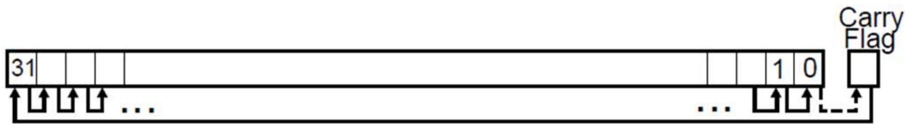


Figure 11-5 RRX

Shift Operations

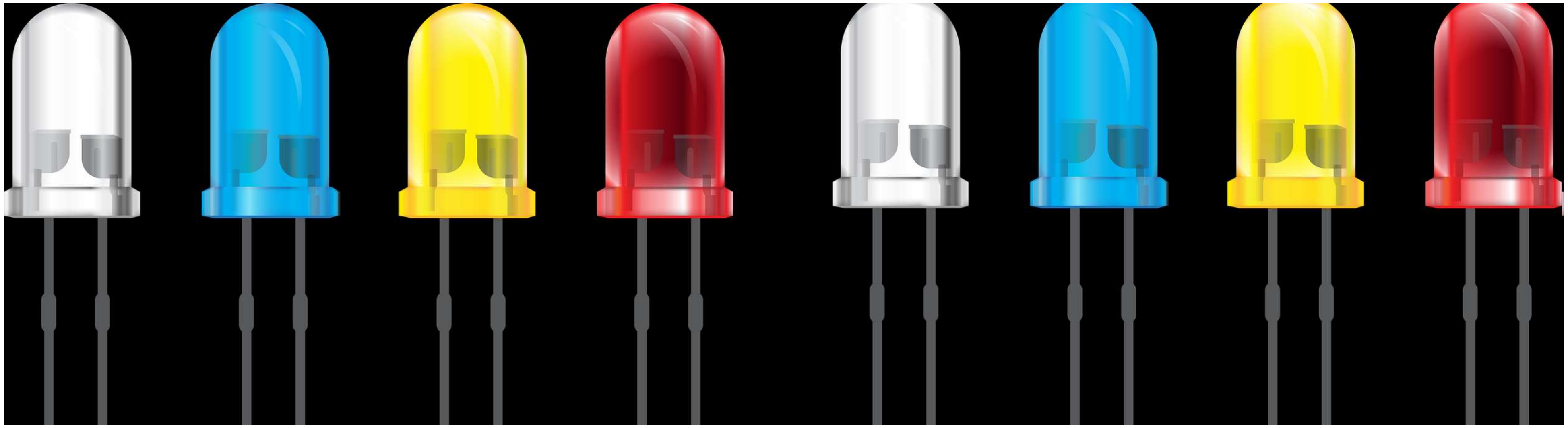
- A and B are 8-bits numbers. Concate these two number to construct a 16-bit number {AB}.

Shift Operations

- A and B are 8-bits numbers. Concate these two number to construct a 16-bit number {AB}.

- Let’s assume A is loaded R0 and B loaded R1
- LSL R0, R0, #8
- ORR R2,R0,R1

Masking & Bit Manipulation



Masking & Bit Manipulation: Setting Bits

- OR operation is used to set bits.
  - `ORR{S}{cond} Rd, Rn, Operand2`
- GPIO_X |= 0xFF;
- GPIO_X |= 0x0C;

- More: armasm User Guide DUI0473M

Masking & Bit Manipulation: Setting Bits

- OR operation is used to set bits.
  - `ORR{S}{cond} Rd, Rn, Operand2`

- GPIO_X |= 0x0C;

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	1	0	0
B7	B6	B5	B4	1	1	B1	B0

Masking & Bit Manipulation: Resetting Bits

- AND operation is used to reset bits.
  - `AND{S}{`*`cond`*`} `*`Rd`*`, `*`Rn`*`, `*`Operand2`*
- GPIO_X &= 0x00;
- GPIO_X &= 0x0C;

Masking & Bit Manipulation: Resetting Bits

- AND operation is used to reset bits.
  - `AND{S}{`*`cond`*`} `*`Rd`*`, `*`Rn`*`, `*`Operand2`*
- GPIO_X &= 0x00;
- GPIO_X &= 0x0C;

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	1	0	0
0	0	0	0	B3	B2	0	0

Masking & Bit Manipulation: Toggle Bits

- EOR operation is used to toggle bits.
  - `EOR{S}{`*`cond`*`}` *`Rd`*`,` *`Rn`*`,` *`Operand2`*
- GPIO_X &= 0x00;
- GPIO_X &= 0x0C;

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	1	0	0
B7	B7	B5	B4	B3’	B2’	B1	B0

Masking & Bit Manipulation: GPIO Read-Write

- There are 8 LEDs connected to GPIOA.
  - Set the 5rd LED on

- GPIOB is configured as input and buttons are connected.
  - Check whether 3rd button is pressed or not

Data Comparison

Sets the condition flags (NZCV) based on the result without storing the result in any register.

- **CMP** Rn, Op2 (Compare): Compares the value in register Rn with Op2. *Equivalent?*
- **CMN** Rn, Op2 (Compare Negative): Compares the value in register Rn with Op2 by adding. *Equivalent?*
- **TST** Rn, Op2 (Test): Performs a bitwise AND operation.
- **TEQ** Rn, Op2 (Test Equivalence): Performs a bitwise exclusive OR (XOR) operation between the values in register Rn and Op2. Used to check if two values are equivalent at the bit level.

Data Movement

1. **MOV** Rd, Operand2: Copies the value of Operand2 into register Rd. Used for straightforward data movement and initialization.

2. **MVN** Rd, Operand2 (Move with NOT): Copies the bitwise negation of Operand2 into register Rd. Inverts all the bits in Operand2 before moving it to Rd.

3. **MRS** Rd, Special_Reg (Move from Special Register): Copies the value from a special system register into general-purpose register Rd. Useful for accessing and storing system-related information.

4. **MSR** Special_Reg, Rm (Move to Special Register): Copies the value from general-purpose register Rm into a special system register. Enables the modification of system control and status registers. **Careful!!**

MOV – Immediate Number

- If the immediate value is less than **16** bits, use MOV with a # sign to set the register value.

```
MOV R0, #0xFF (Hexadecimal)
MOV R0, #0b10011100 (Binary)
MOV R0, #54 (Decimal)
MOV R0, #0d54 (Decimal)
```

How can we load 32 bit number?

Load and Store – Immediate Number

Instruction	**Description**
MOV Rd, #<immed_16>	Move an 16-bit immediate value into the register Rd.
MOVT Rd, #<immed_16>	Move a 16-bit immediate value into the top halfword (bits 31:16) of the register Rd. The bottom halfword remains unaltered.
MOVW Rd, #<immed_16>	Move a 16-bit immediate value into the bottom halfword (bits 15:0) of the register Rd and clear the top halfword (bits 31:16).
LDR Rt, =#<immed_8>	Equivalent to the MOV instruction. Load an 8-bit immediate value into the register Rt.
LDR Rt, =#<immed_32>	A pseudo instruction. Load a 32-bit immediate value into the register Rt.

Load and Store - Memory

- **LDR** (Load Instruction): A load instruction retrieves data from a specified memory address and stores the data into a specific register.

```
; Load the value at memory address 0x1000 into R0
LDR R0, [0x1000]
```

- **STR** (Store Instruction): A store instruction performs the reverse operation; it takes the content of a register and writes it to a specified memory address.

```
; Store the value in R1 into memory address 0x2000
STR R1, [0x2000]
```

Load and Store – Immediate Number

- LDR R0, =0x12345678 ;Pseudo instruction
- LDR R1, [R0]   ;Not a pseudo instruction
- ADR R0, myDataLabel ; R0 = Address of myDataLabel
- MOV32???

- Speed: LDR vs MOV?

Load and Store – Summary

ARM is a Load-Store RISC architecture.

- MOV Rn, #Imm ; Load a (16-bit) immediate value
- MOV Rn, Rm ; Copy one register to another
- LDR Rn, [Rm] ;Rn = value pointed by Rm
- LDR Rn, [Rm,#4] ;Rn = *(Rm+4)

Addressing Modes and Operands

- Immediate addressing
- Indexed addressing
- PC relative addressing

Addressing Modes - Immediate Addressing

- If the data is found in the instruction itself, like **MOV R0,#1**, the instruction uses **immediate addressing mode**.
- Immediate data: prefixed with “#”

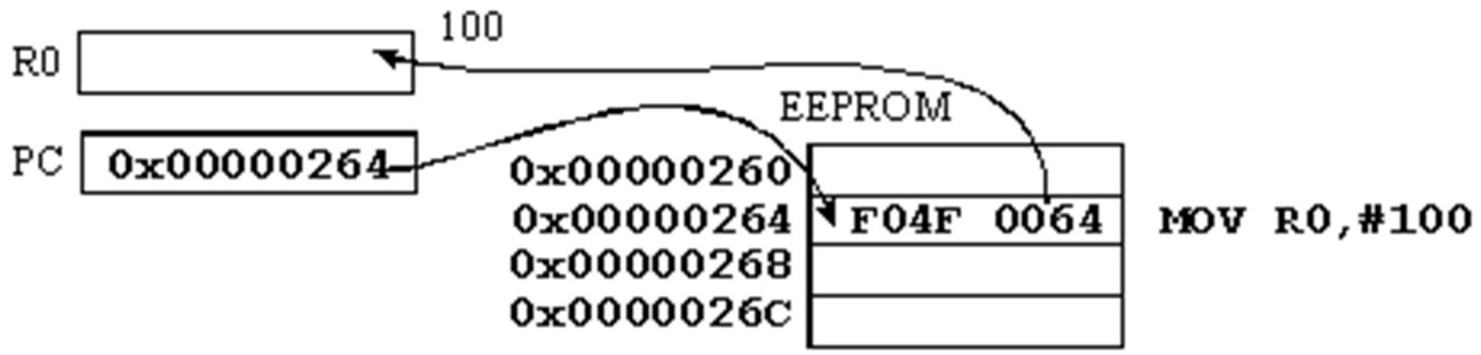
MOVS R0, #0x1F; Set R0 = 0x1F

MOVS R0, #’A’; Set R0 = 0x41 (ASCII code)

Addressing Modes - Immediate Addressing

- With immediate addressing mode, the data itself is contained in the instruction. Once the instruction is fetched no additional memory access cycles are required to get the data.

MOV R0, #100; R0=100

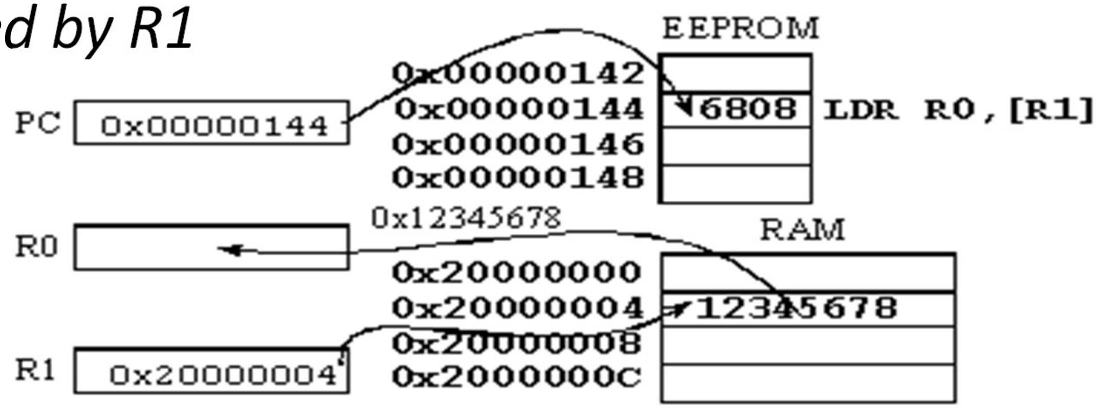


EMBEDDED SYSTEMS: INTRODUCTION TO ARM CORTEX-M MICROCONTROLLERS - Jonathan W. Valvano

Addressing Modes - Indexed Addressing

- Indexed addressing mode uses a register pointer to access memory.
- A register that contains the address or the location of the data is called a pointer or index register.

LDR R0, [R1]; R0=value pointed by R1



EMBEDDED SYSTEMS: INTRODUCTION TO ARM CORTEX-M MICROCONTROLLERS - Jonathan W. Valvano

Addressing Modes - PC Relative Addressing

- The addressing mode that uses the PC as the pointer is called PC-relative addressing mode.
- It is used for branching, for calling functions, and accessing constant data stored in ROM.
- The addressing mode is called PC relative because the machine code contains the address difference between where the program is now and the address to which the program will access.

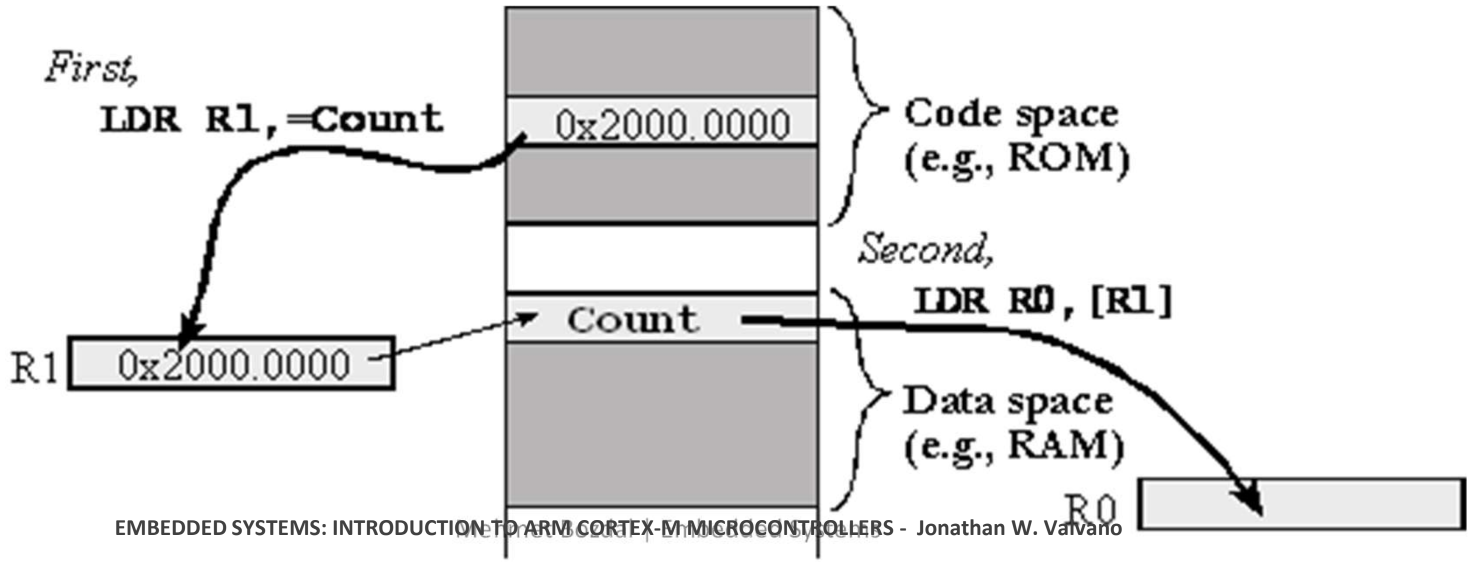
LDR R1,=Count ; R1 points to variable Count, using PC-relative

LDR R1,[PC,#28] ; The assembler translates the =Count into the correct PC-relative access

Addressing Modes - PC Relative Addressing

- Typically, it takes two instructions to access data in RAM or I/O.

```
LDR   R1,=Count     ; R1 points to variable Count, using PC-relative
                    ; LDR R1,[PC,#28]
LDR   R0,[R1]       ; R0= value pointed to by R1
```



Q & A

AGÜ