

به نام خدا

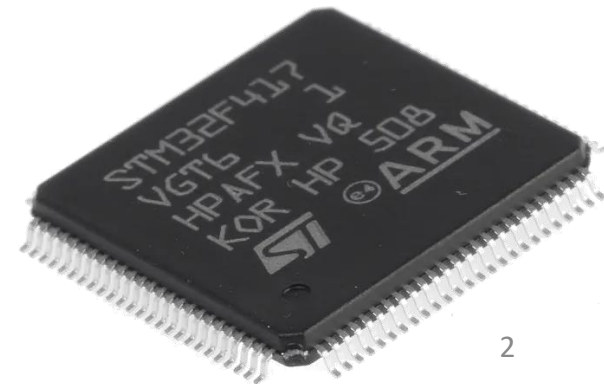
آشنایی (و برنامه نویسی) با میکروکنترلرهای ARM

Dr. Aref Karimiafshar
A.karimiafshar@iut.ac.ir



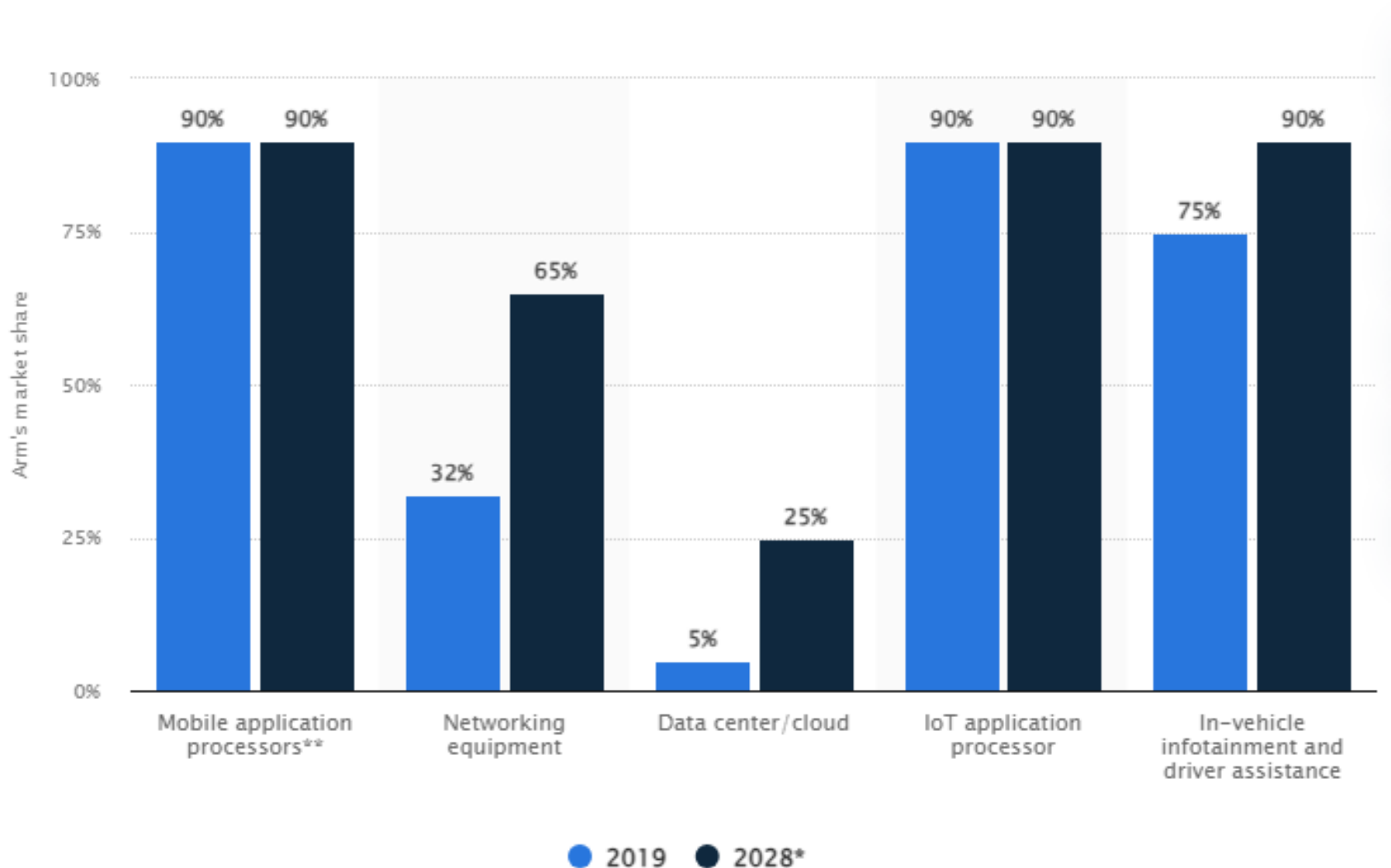
ARM processor

- ARM is a family of RISC architectures.
 - **ARM** is the abbreviation of **Advanced RISC Machines**
- ARM does not manufacture its own VLSI devices.
 - licenses
- ARM7- von Neuman Architecture
- ARM9 –Harvard Architecture



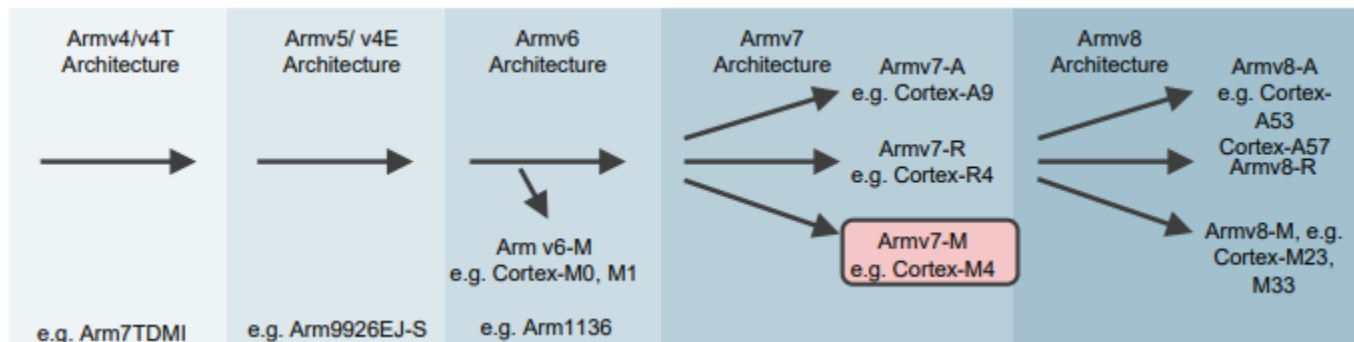
ARM's market share

across different technology



ARM processors vs. ARM architectures

- Arm architecture
 - Describes the details of instruction set, programmer's model, exception model, and memory map
 - Documented in the Architecture Reference Manual
- Arm processor
 - Developed using one of the Arm architectures
 - More implementation details, such as timing information
 - Documented in processor's Technical Reference Manual



Equipment Adopting Arm Cores

M



Tele-parking



Intelligent toys



Utility Meters



IR Fire Detector



Exercise Machines



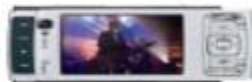
Energy Efficient Appliances



Intelligent Vending



R

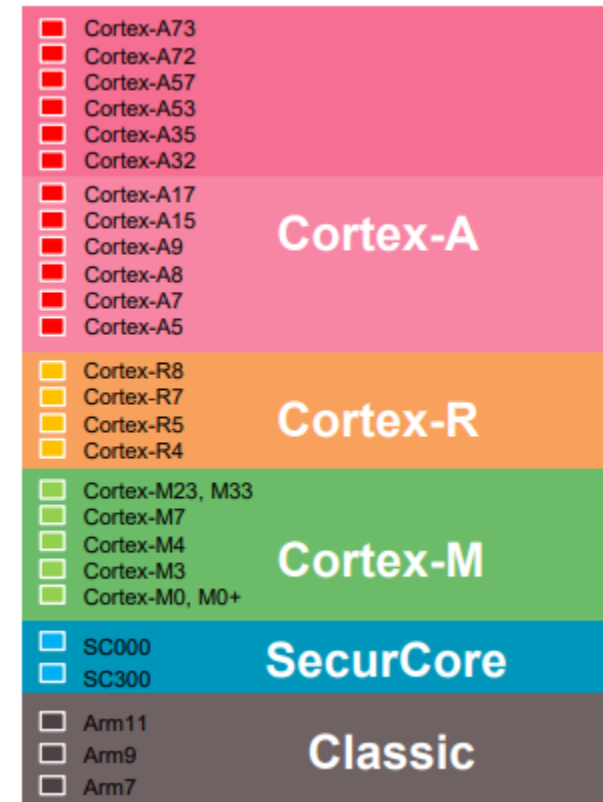


A

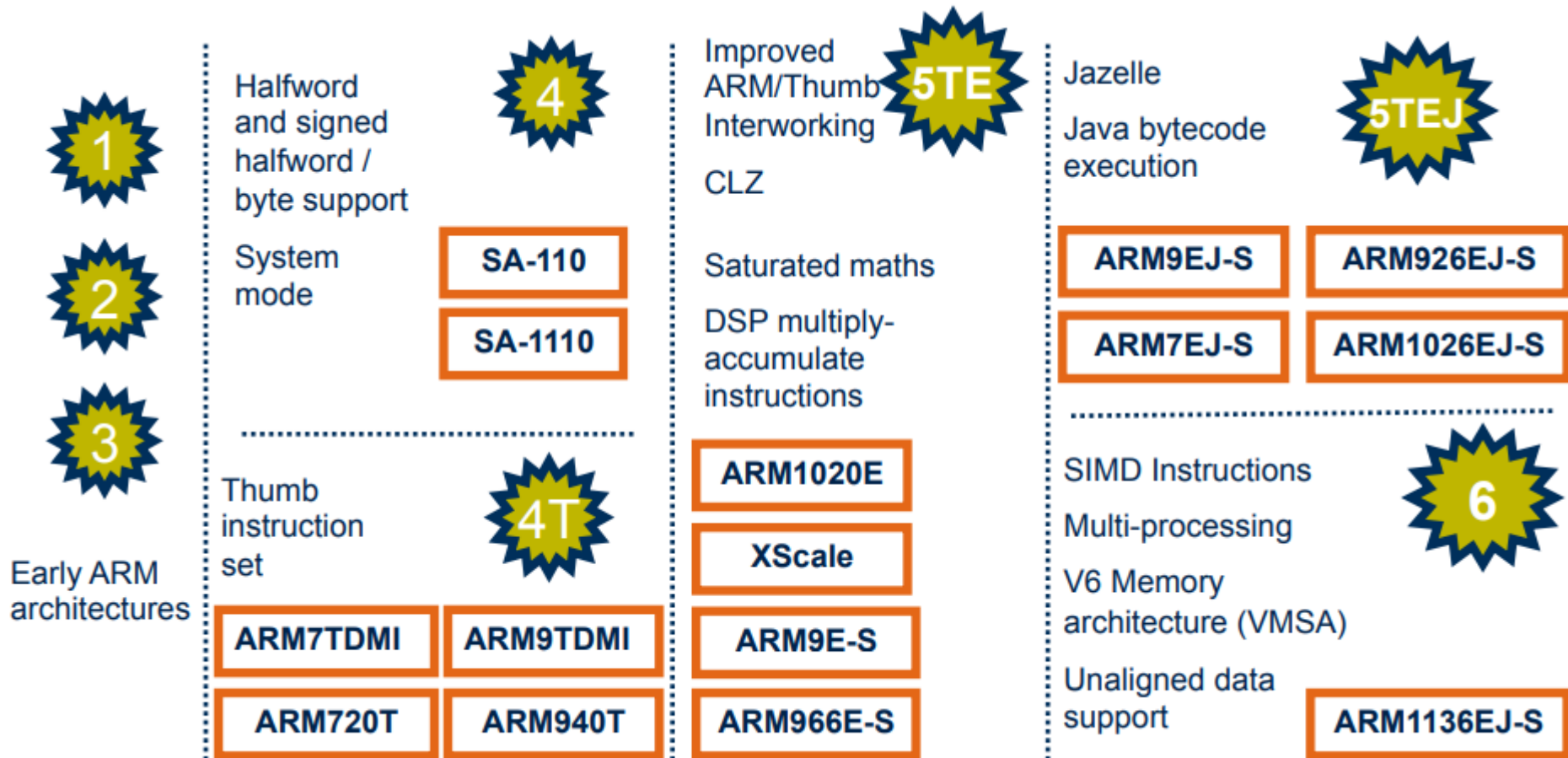


ARM processor families

- **Cortex-A series (Application)**
 - High performance processors capable of full Operating System (OS) support
 - Applications include smartphones, digital TV, smart books
- **Cortex-R series (Real-time)**
 - High performance and reliability for real-time applications;
 - Applications include automotive braking system, powertrains
- **Cortex-M series (Microcontroller)**
 - Cost-sensitive solutions for deterministic microcontroller applications
 - Applications include microcontrollers, smart sensors
- **SecurCore series**
 - High security applications
- Earlier **classic** processors including Arm7, Arm9, Arm11 families



Development of ARM Architecture

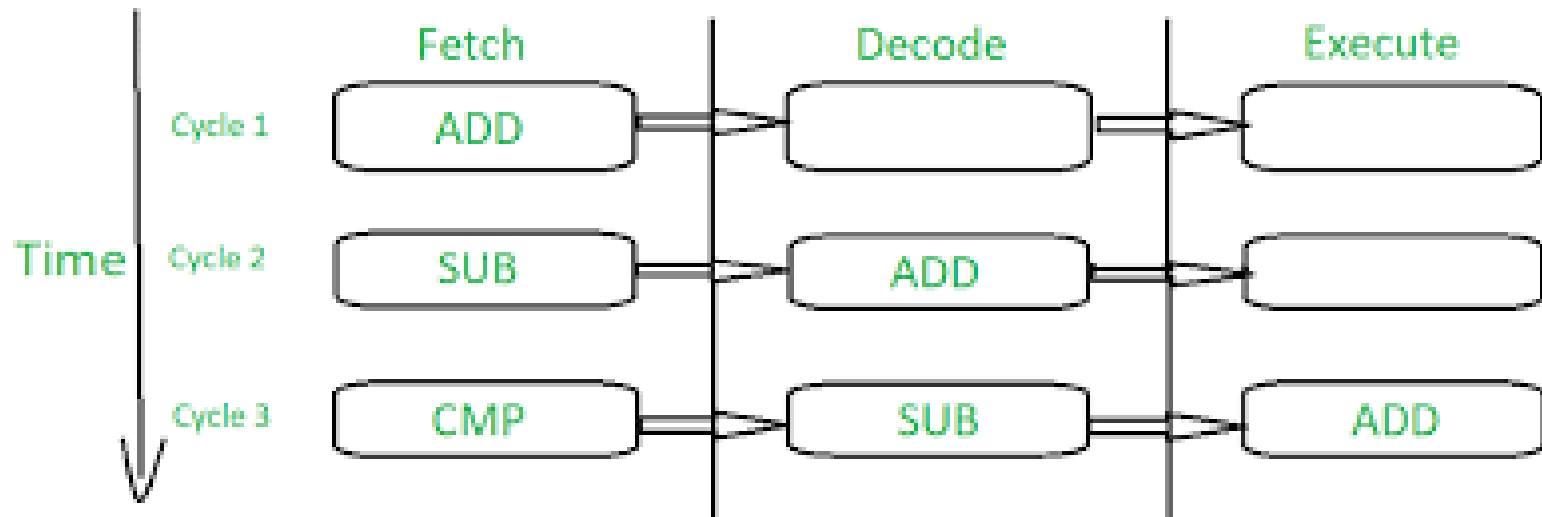


Naming Rule of ARM

- ARM {x} {y} {z} {T} {D} {M} {I} {E} {J} {F} {-S}
 - x: series
 - y: memory management / protection unit
 - z: cache
 - T: Thumb decoder
 - D: JTAG debugger
 - M: fast multiplier
 - I: support hardware debug
 - E: enhance instructions (based on TDMI)
 - J: Jazelle
 - F: vector floating point unit
 - S: synthesiable, suitable for EDA tools

ARM single-cycle instruction

3-stage pipeline operation

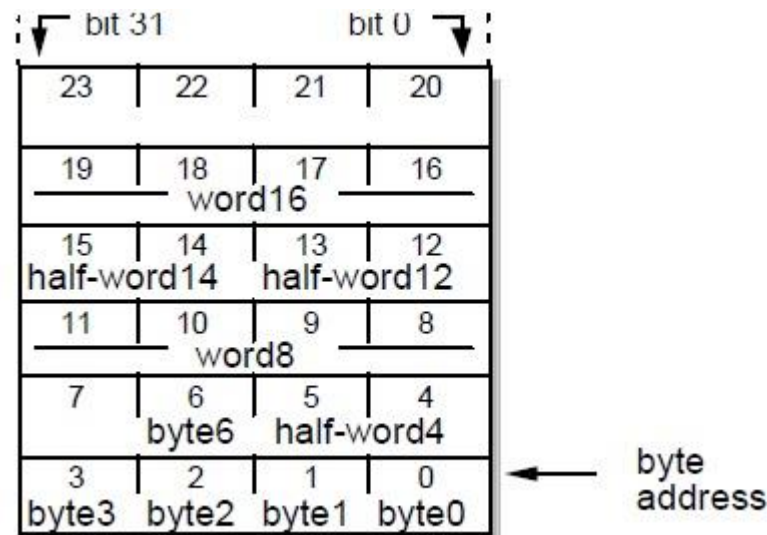


Processor Operating States

- The ARM7TDMI processor has two operating states:
- ARM - 32-bit
 - word-aligned ARM instructions are executed in this state.
- Thumb -16-bit
 - halfword-aligned Thumb instructions are executed in this state

The Memory System

- 4 G address space
 - 8-bit bytes, 16-bit half-words, 32-bit words
 - Support both little-endian and big-endian



Operating Modes

- The ARM7TDMI processor has seven modes of operations:
 - User mode(usr)
 - Normal program execution mode
 - Fast Interrupt mode(fiq)
 - Supports a high-speed data transfer or channel process.
 - Interrupt mode(irq)
 - Used for general-purpose interrupt handling.
 - Supervisor mode(svc)
 - Protected mode for the operating system.
 - Abort mode(abt)
 - implements virtual memory and/or memory protection
 - System mode(sys)
 - A privileged user mode for the operating system. (runs OStasks)
 - Undefined mode(und)
 - supports a software emulation of hardware coprocessors
- Except user mode, all are known as privileged mode

Registers

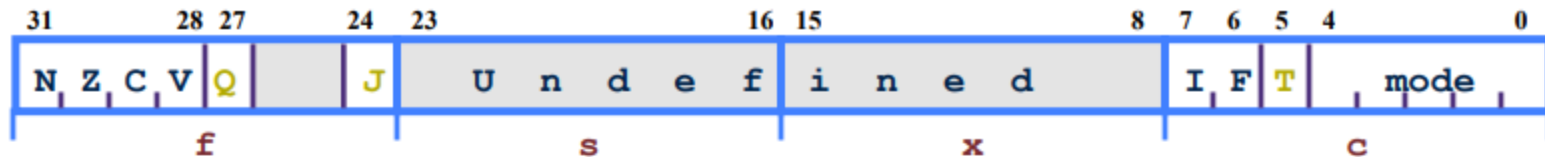
- 37 registers
 - 31 general 32 bit registers, including PC
 - 6 status registers
 - 15 general registers (R0 to R14), and one status registers and program counter are visible at any time –when you write user-level programs
 - R13 (SP)
 - R14 (LR)
 - R15 (PC)
- The visible registers depend on the processor mode
- The other registers (the banked registers) are switched in to support IRQ, FIQ, Supervisor, Abort and Undefined mode processing

Registers

User	FIQ	IRQ	SVC	Undef	Abort
r0	User mode r0-r7, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8				
r9	r9				
r10	r10				
r11	r11				
r12	r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
r15 (pc)					
cpsr					
	spsr	spsr	spsr	spsr	spsr

Note: System mode uses the User mode register set

Program Status Registers



■ Condition code flags

- N = **N**egative result from ALU
- Z = **Z**ero result from ALU
- C = ALU operation **C**arried out
- V = ALU operation **oV**erflowed

■ Sticky Overflow flag - Q flag

- Architecture 5TE/J only
- Indicates if saturation has occurred

■ J bit

- Architecture 5TEJ only
- J = 1: Processor in Jazelle state

■ Interrupt Disable bits.

- I = 1: Disables the IRQ.
- F = 1: Disables the FIQ.

■ T Bit

- Architecture xT only
- T = 0: Processor in ARM state
- T = 1: Processor in Thumb state

■ Mode bits

- Specify the processor mode

Program Counter (r15)

- **When the processor is executing in ARM state:**
 - All instructions are 32 bits wide
 - All instructions must be word aligned
 - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned).
- **When the processor is executing in Thumb state:**
 - All instructions are 16 bits wide
 - All instructions must be halfword aligned
 - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned).
- **When the processor is executing in Jazelle state:**
 - All instructions are 8 bits wide
 - Processor performs a word access to read 4 instructions at once

ARM assembly language

- Fairly standard assembly language:

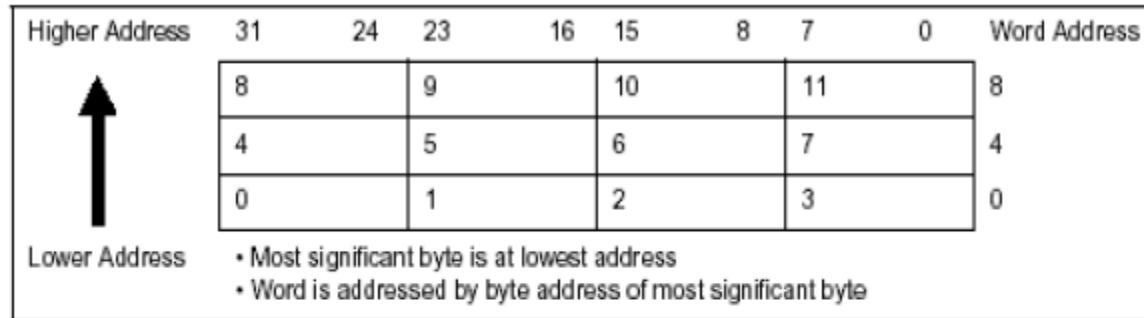
```
        LDR r0,[r8] ; a comment  
label ADD r4,r0,r1
```

ARM data types

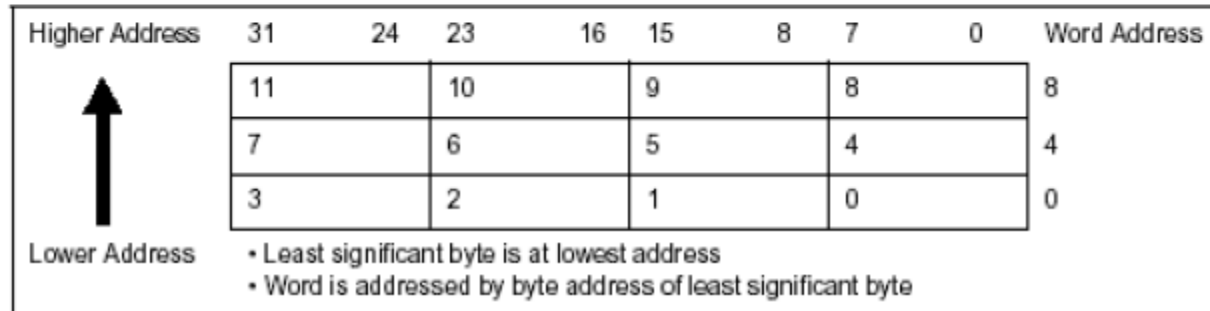
- 32-bit word
- Word can be divided into four 8-bit bytes
- ARM addresses can be 32 bits long
- Address refers to byte
 - Address 4 starts at byte 4
- Can be configured at power-up as either little- or bit-endian mode

Big Endian and Little Endian

Big endian



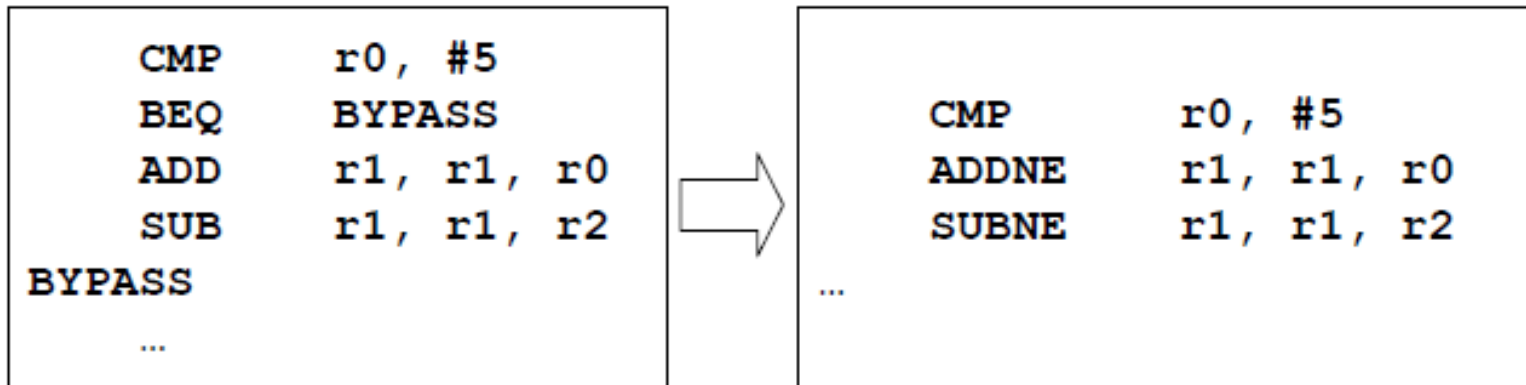
Little endian



Notable Features of ARM Instruction Set

- The load-store architecture
- 3-address data processing instructions
- Conditional execution of every instruction
- The inclusion of every powerful load and store multiple register instructions
- Single-cycle execution of all instruction
- Open coprocessor instruction set extension

Conditional Execution



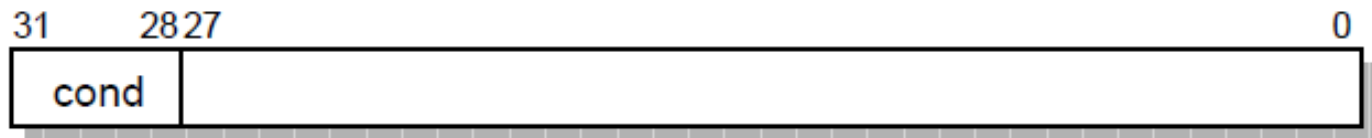
Conditional Execution

- The conditional execution code is faster and smaller

```
; if ((a==b) && (c==d)) e++;  
;  
; a is in register r0  
; b is in register r1  
; c is in register r2  
; d is in register r3  
; e is in register r4  
  
    CMP     r0, r1  
    CMPEQ   r2, r3  
    ADDEQ   r4, r4, #1
```

The ARM Condition Code Field

- Every instruction is conditionally executed
- Each of the 16 values of the condition field causes the instruction to be executed or skipped according to the values of the N, Z, C and V flags in the CPSR



N: Negative Z: Zero C: Carry V: oVerflow

ARM Condition Codes

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Data processing instructions

- Arithmetic and logical operations
 - The applied rules
 - All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself
 - The result, if there is one, is 32 bits wide and is placed in a Register
(An exception: long multiply instructions produce a 64 bits result)
 - Each of the operand registers and the result register are independently specified in the instruction
(This is, the ARM uses a '3-address' format for these instructions)

Simple Register Operands

```
ADD  r0, r1, r2      ; r0 := r1 + r2
```



The semicolon here indicates that everything to the right of it is a comment and should be ignored by the assembler

The values in the register may be considered to be unsigned integer or signed 2's-complement values

Arithmetic Operations

- These instructions perform binary arithmetic on two 32-bit operands
- The carry-in, when used, is the current value of the C bit in the CPSR

ADD	r0, r1, r2	$r0 := r1 + r2$
ADC	r0, r1, r2	$r0 := r1 + r2 + C$
SUB	r0, r1, r2	$r0 := r1 - r2$
SBC	r0, r1, r2	$r0 := r1 - r2 + C - 1$
RSB	r0, r1, r2	$r0 := r2 - r1$
RSC	r0, r1, r2	$r0 := r2 - r1 + C - 1$

Bit-Wise Logical Operations

AND r0, r1, r2	r0 := r1 AND r2
ORR r0, r1, r2	r0 := r1 OR r2
EOR r0, r1, r2	r0 := r1 XOR r2
BIC r0, r1, r2	r0 := r1 AND (NOT r2)

- BIC stands for 'bit clear'

Register Movement Operations

- These instructions ignore the first operand, which is omitted from the assembly language format, and simply move the second operand to the destination

MOV r0, r2	r0 := r2
MVN r0, r2	r0 := NOT r2

Comparison Operations

- These instructions do not produce a result, but just set the condition code bits (N, Z, C, and V) in the CPSR according to the selected operation

CMP	r1, r2	compare	set cc on $r1 - r2$
CMN	r1, r2	compare negated	set cc on $r1 + r2$
TST	r1, r2	bit test	set cc on $r1 \text{ AND } r2$
TEQ	r1, r2	test equal	set cc on $r1 \text{ XOR } r2$

Shifted Register Operands

- These instructions allows the second register operand to be subject to a shift operation before it is combined with the first operand

```
ADD    r3, r2, r1, LSL #3    ; r3 := r2 + 8 * r1
```

- They are still single ARM instructions, executed in a single clock cycle
- Most processors offer shift operations as separate instructions, but the ARM combines them with a general ALU operation in a single instruction

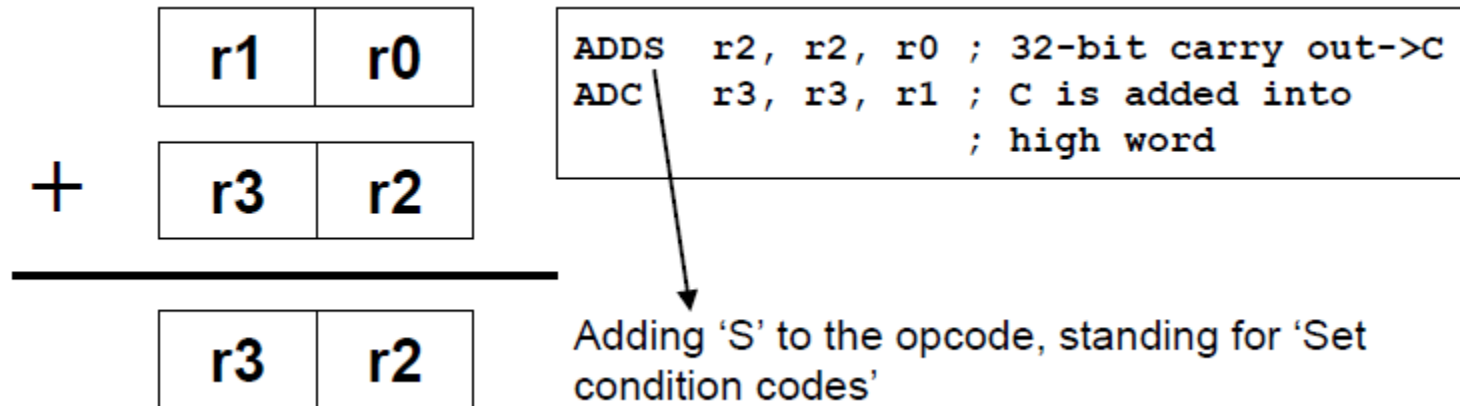
Shifted Register Operands

- It is possible to use a register value to specify the number of bits the second operand should be shifted by
- Only the bottom 8 bits of r2 are significant

```
ADD    r5, r5, r3, LSL r2    ; r5:=r5+r3*2^r2
```


Setting the Condition Codes

- Any data processing instruction can set the condition codes (N, Z, C, and V) if the programmer wishes it to



Addressing mode

- The ARM data transfer instructions are all based around register-indirect addressing
 - Based-plus-offset addressing
 - Based-plus-index addressing

```
LDR    r0, [r1]    ; r0 := mem32[r1]  
STR    r0, [r1]    ; mem32[r1] := r0
```

Data Transfer Instructions

- Move data between ARM registers and memory
- Three basic forms of data transfer instruction
 - Single register load and store instructions
 - Multiple register load and store instructions
 - Single register swap instructions

Single Register Load / Store Instructions

- These instructions provide the most flexible way to transfer single data items between an ARM register and memory
- The data item may be a **byte**, a **32-bit word**, **16-bit half-word**

```
LDR    r0, [r1]    ; r0 := mem32[r1]  
STR    r0, [r1]    ; mem32[r1] := r0
```

Single Register Load / Store Instructions

LDR	Load a word into register	$Rd \leftarrow \text{mem32}[\text{address}]$
STR	Store a word in register into memory	$\text{Mem32}[\text{address}] \leftarrow Rd$
LDRB	Load a byte into register	$Rd \leftarrow \text{mem8}[\text{address}]$
STRB	Store a byte in register into memory	$\text{Mem8}[\text{address}] \leftarrow Rd$
LDRH	Load a half-word into register	$Rd \leftarrow \text{mem16}[\text{address}]$
STRH	Store a half-word in register into memory	$\text{Mem16}[\text{address}] \leftarrow Rd$
LDRSB	Load a signed byte into register	$Rd \leftarrow \text{signExtend}(\text{mem8}[\text{address}])$
LDRSH	Load a signed half-word into register	$Rd \leftarrow \text{signExtend}(\text{mem16}[\text{address}])$

Base-plus-offset Addressing

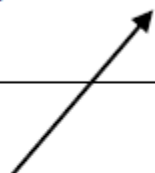
- Pre-indexed addressing mode
 - It allows one base register to be used to access a number of memory locations which are in the same area of memory

```
LDR    r0, [r1, #4]    ; r0 := mem32[r1 + 4]
```

Base-plus-offset Addressing

- Auto-indexing (Preindex with writeback)
 - No extra time
 - The time and code space cost of the extra instruction are avoided

```
LDR    r0, [r1, #4]!    ; r0 := mem32[r1 + 4]  
                        ; r1 := r1 + 4
```



The exclamation “!” mark indicates that the instruction should update the base register after initiating the data transfer

Base-plus-offset Addressing

- Post-indexed addressing mode
 - The exclamation “!” is not needed

```
LDR    r0, [r1], #4    ; r0 := mem32[r1]  
                        ; r1 := r1 + 4
```


Multiple Register Load / Store Instructions

- Enable large quantities of data to be transferred more efficiently
- They are used for procedure entry and exit to save and restore workspace registers
- Copy blocks of data around memory

```
LDMIA    r1, {r0, r2, r5}    ; r0 := mem32[r1]  
                                   ; r2 := mem32[r1 + 4]  
                                   ; r5 := mem32[r1 + 8]
```

The base register r1 should be word-aligned

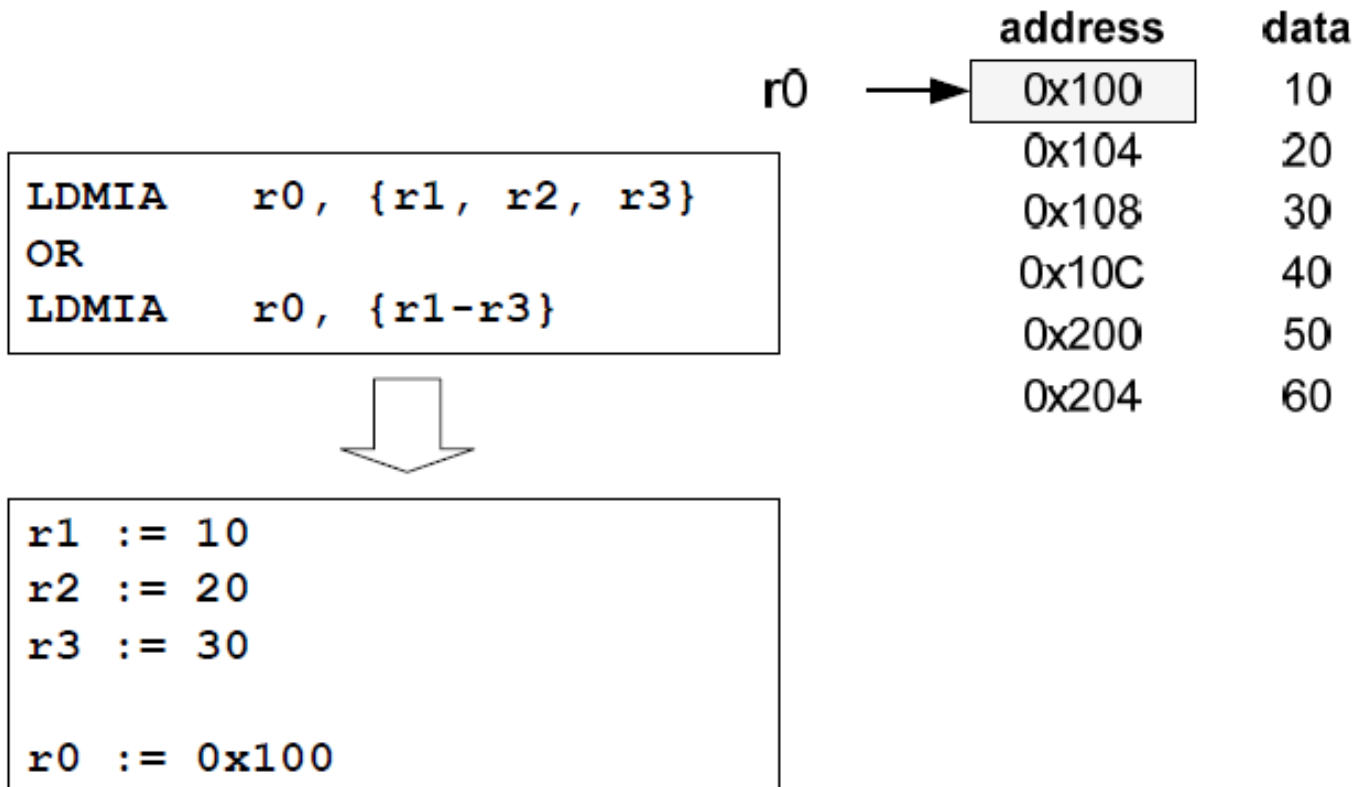
Multiple Register Load / Store Instructions

LDM	Load multiple registers
STM	Store multiple registers

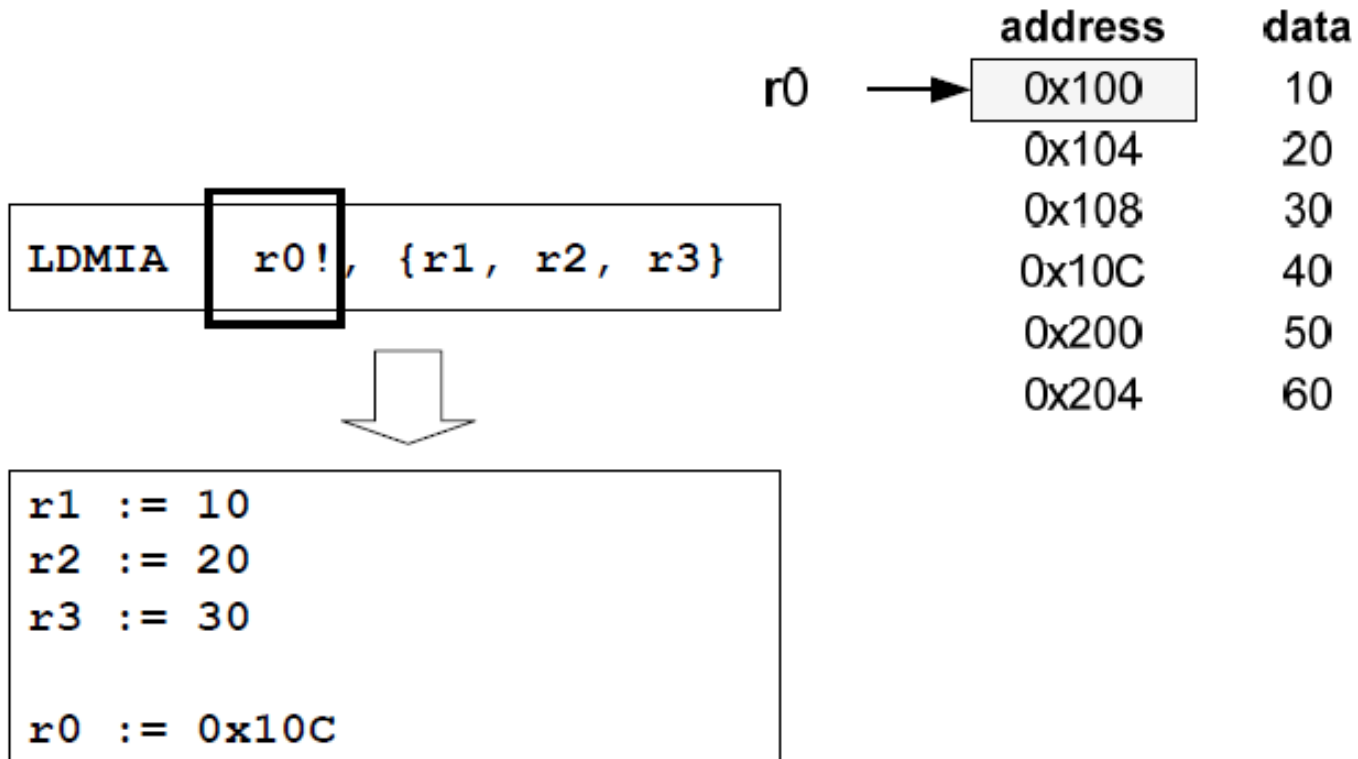
Addressing mode	Description	Starting address	End address	Rn!
IA	Increment After	R_n	$R_n + 4 * N - 4$	$R_n + 4 * N$
IB	Increment Before	$R_n + 4$	$R_n + 4 * N$	$R_n + 4 * N$
DA	Decrement After	$R_n - 4 * R_n + 4$	R_n	$R_n - 4 * N$
DB	Decrement Before	$R_n - 4 * N$	$R_n - 4$	$R_n - 4 * N$

Addressing mode for multiple register load and store instructions

Example



Example



Single Register Swap Instructions

- Allow a value in a register to be exchanged with a value in memory
- Effectively do both a load and a store operation in one instruction
- They are little used in user-level programs
- Atomic operation
- Application
 - Implement semaphores (multi-threaded / multi-processor environment)

Single Register Swap Instructions

`SWP{B} Rd, Rm, [Rn]`

SWP	WORD exchange	tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp
SWPB	Byte exchange	tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp

Control Flow Instructions

```
        B        LABEL
        ...
        ...
LABEL ...
```

```
        MOV      r0, #0        ; initialize counter
LOOP    ...
        ADD      r0, r0, #1    ; increment loop counter
        CMP      r0, #10      ; compare with limit
        BNE      LOOP        ; repeat if not equal
        ...                  ; else fall through
```

Branch Conditions

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Branch and Link Instructions

- BL instruction save the return address into r14 (lr)

```
BL      subroutine    ; branch to subroutine
CMP     r1, #5        ; return to here
MOVEQ   r1, #0
...

subroutine                ; subroutine entry point
...
MOV     pc, lr         ; return
```

Branch and Link Instructions

- Problem
 - If a subroutine wants to call another subroutine, the original return address, r14, will be overwritten by the second BL instruction
- Solution
 - Push r14 into a stack
 - The subroutine will often also require some work registers, the old values in these registers can be saved at the same time using a store multiple instruction

پایان

موفق و پیروز باشید