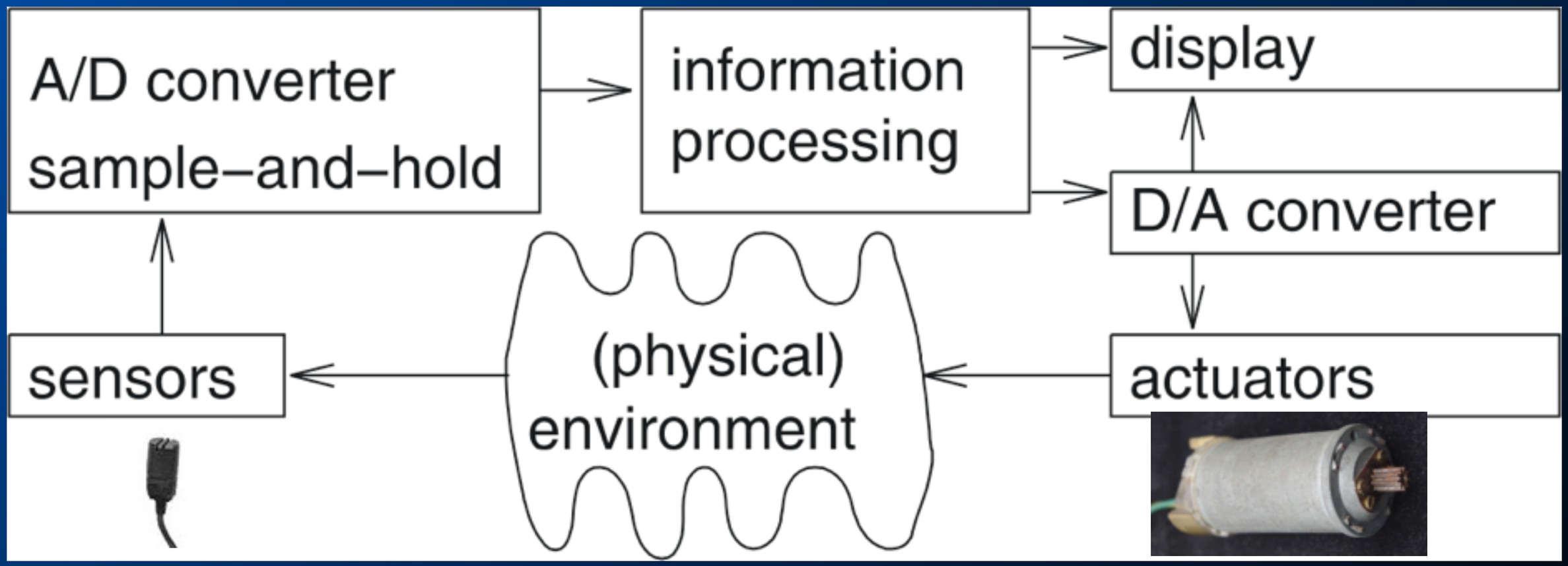
Embedded System Hardware

Agenda

- Composition of the Embedded System
- Embedded Microprocessor
- ARM

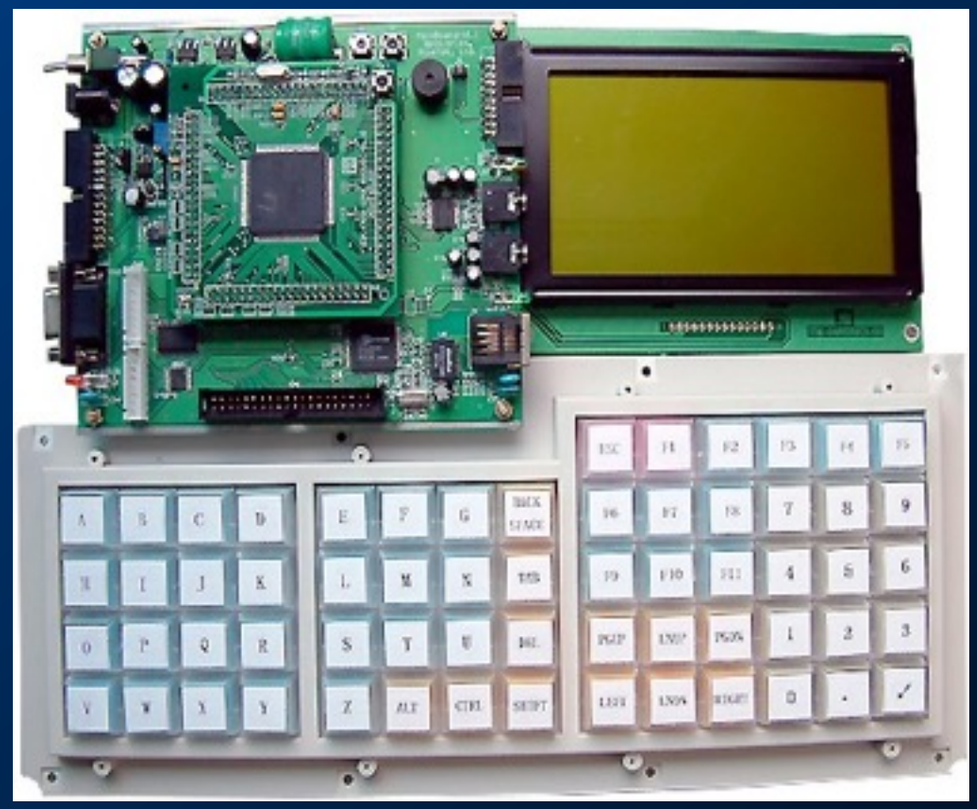
Embedded System Hardware

- Embedded system hardware is frequently used in a loop (***"hardware in a loop"***):



☞ cyber-physical systems

Hardware platform architecture

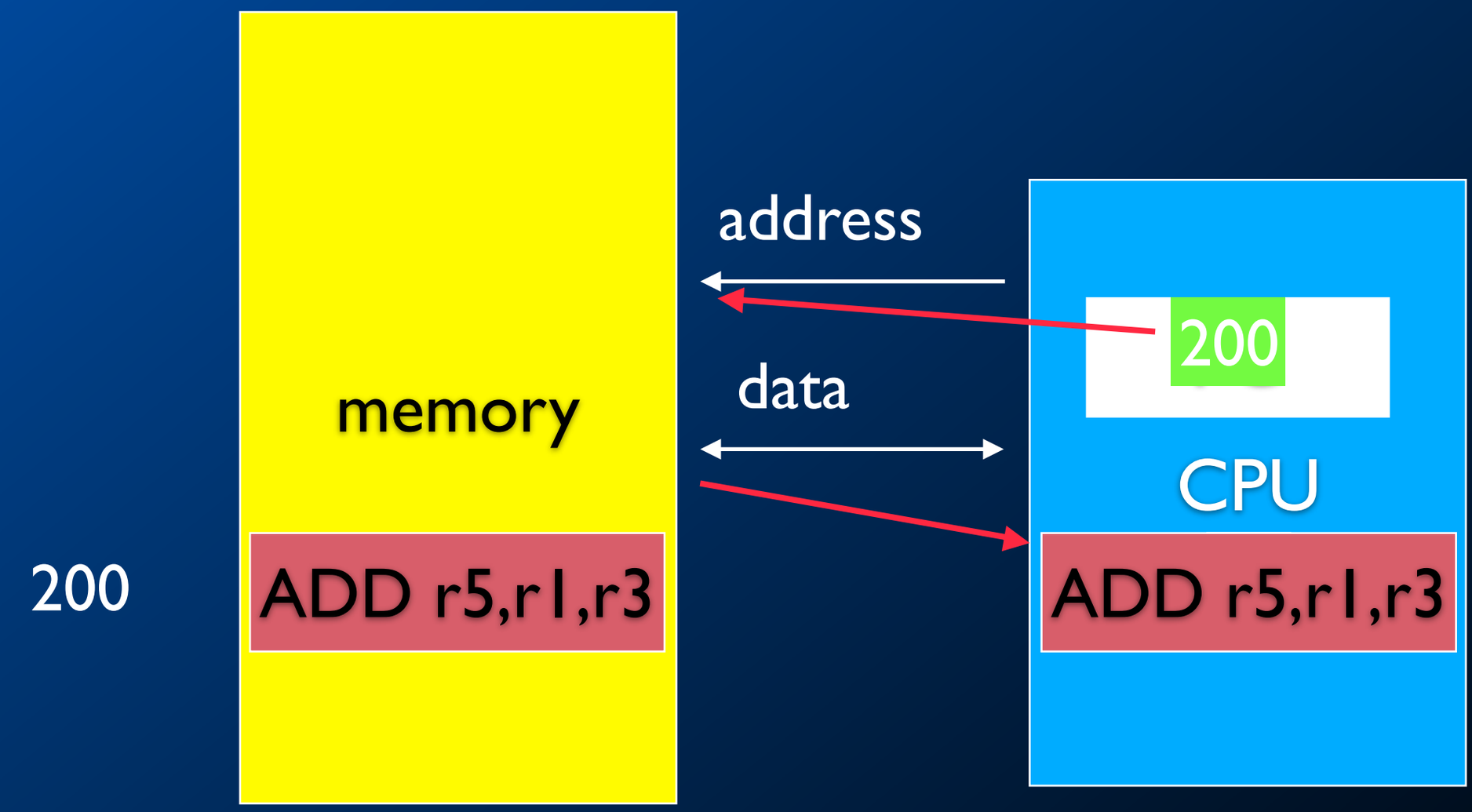


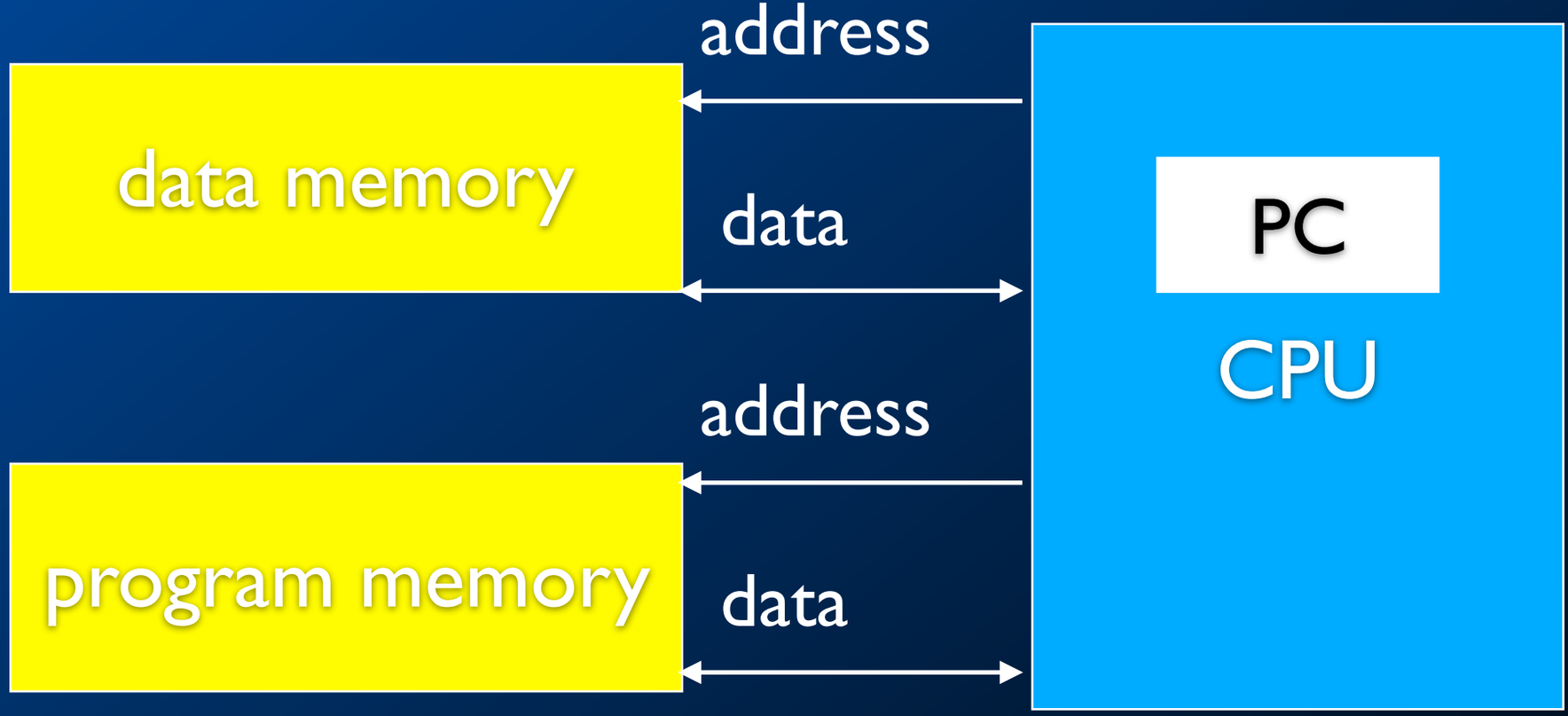
- Contains several elements:
  - CPU;
  - bus;
  - memory;
  - I/O devices: networking, sensors, actuators, etc.
- How big/fast much each one be?

von Neumann architecture

- Memory holds data, instructions.
- Central processing unit (CPU) fetches instructions from memory.
  - Separate CPU and memory distinguishes programmable computer.
- CPU registers help out: program counter (PC), instruction register (IR), general-purpose registers, etc.

CPU + memory




address
data memory
data
PC
CPU
address
program memory
data

von Neumann vs. Harvard

- Harvard can't use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

RISC vs. CISC

- Complex instruction set computer (CISC):
  - many addressing modes;
  - many operations.
- Reduced instruction set computer (RISC):
  - load/store;
  - pipelinable instructions.

Instruction set characteristics

- Fixed vs. variable length.
- Addressing modes.
- Number of operands.
- Types of operands.

Programming model

- Programming model: registers visible to the programmer.
- Some registers are not visible (IR).

Agenda

- Composition of the Embedded System
- Embedded Microprocessor
- ARM

嵌入式微处理器的分类

- 嵌入式微处理器种类繁多，按位数可分为4位、8位、16位、32位和64位。

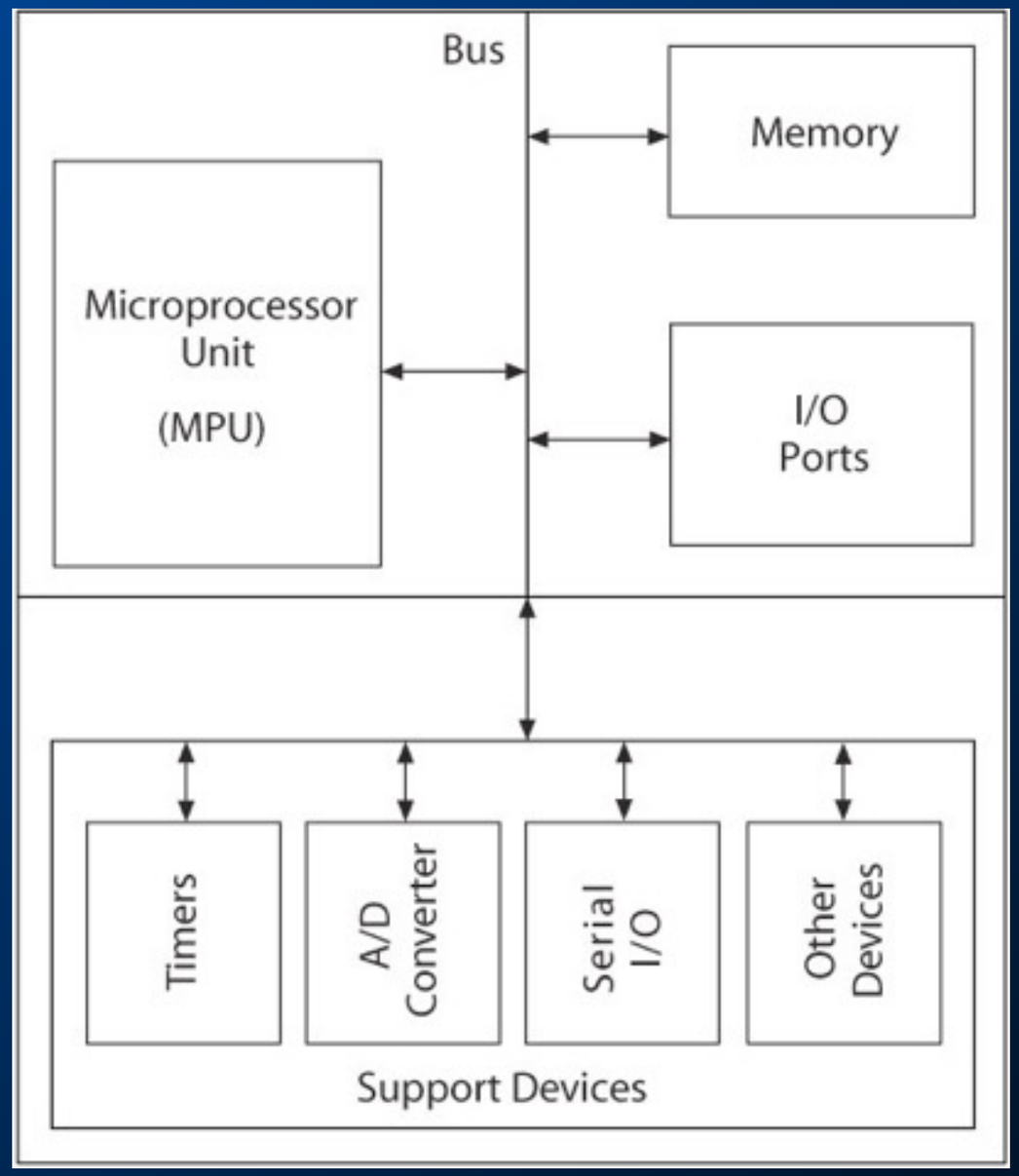
- 根据功能不同，嵌入式微处理器分为四种：
  - 嵌入式微处理单元（MPU）
  - 嵌入式微控制器（MCU）
  - 嵌入式DSP处理器
  - 嵌入式SoC

嵌入式微处理单元（MPU）

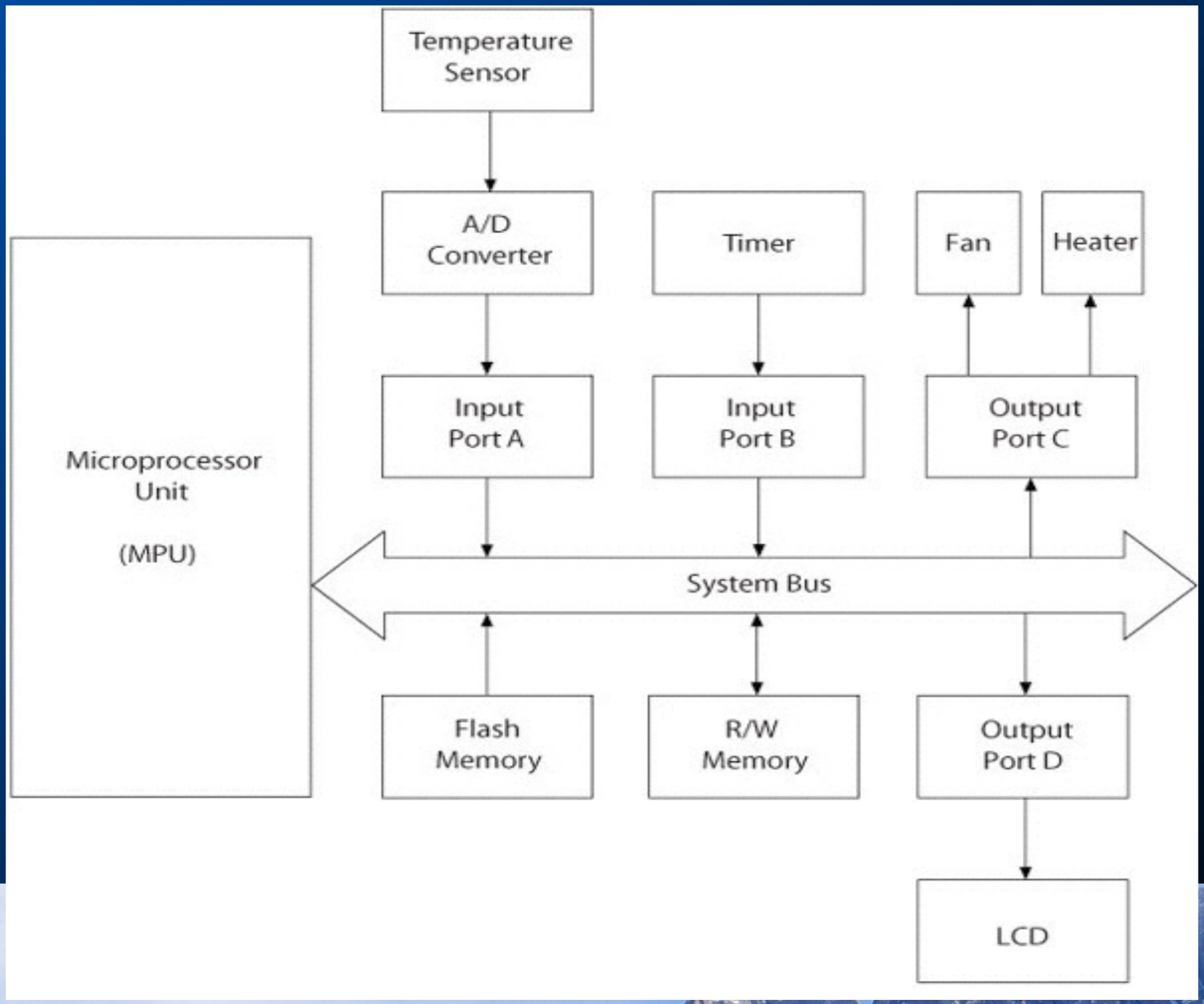
- 嵌入式微处理器就是和通用计算机的处理器对应的CPU。
  - 功能和微处理器基本一样,是具有32位以上的处理器,具有较高的性能.
  - 具有体积小,功耗少,成本低,可靠性高的特点.
  - 有的可提供工业级应用.

- 流行的嵌入式微处理器:
  - ARM(ARM公司): Cortex-A8/A9/A15
  - Power
  - MIPS(MIPS公司)

Block Diagram



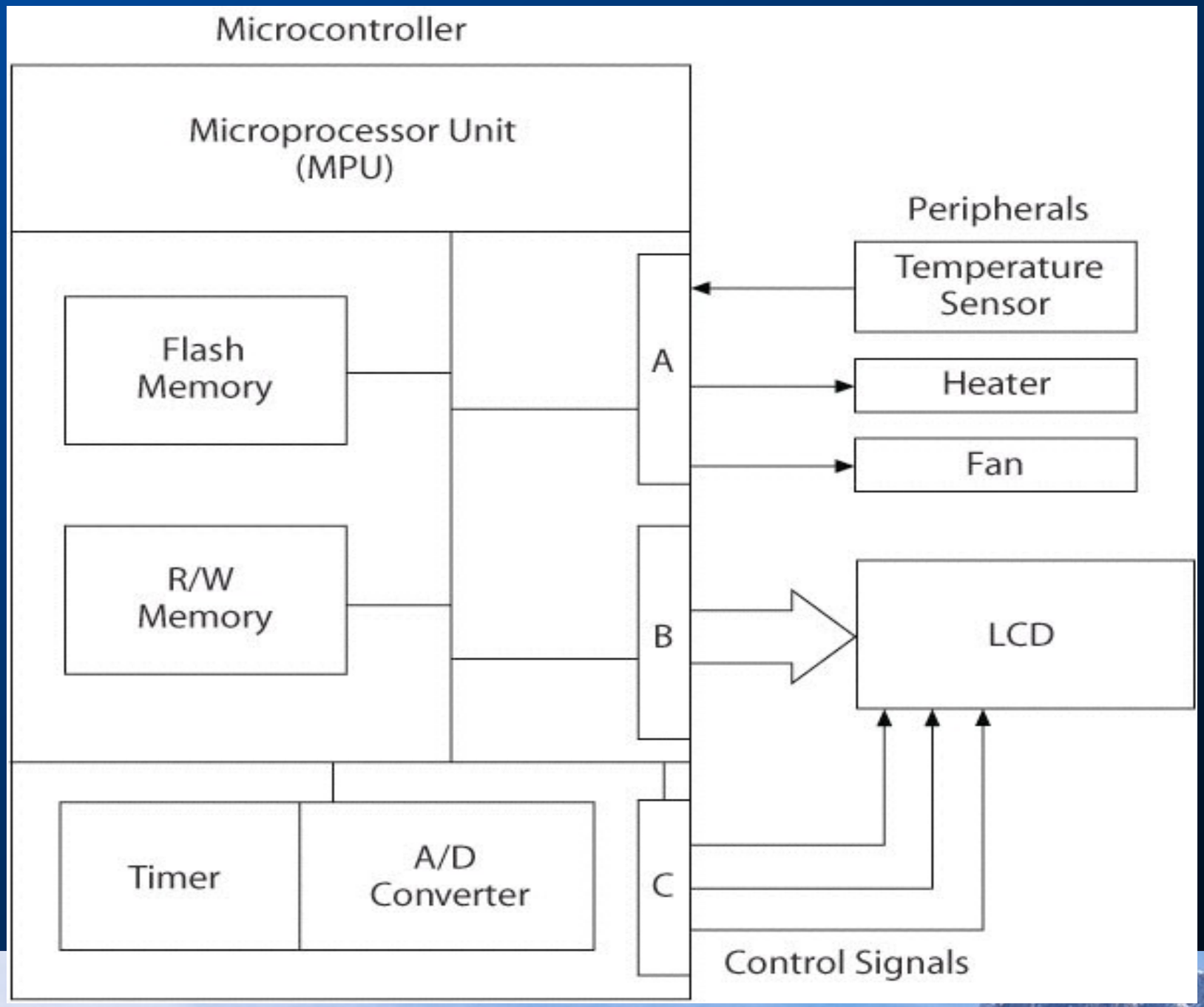
MPU-Based Time and Temperature System



嵌入式微控制器（MCU）

- 嵌入式微控制器就是将整个计算机系统的主要硬件集成到一块芯片中,芯片内部集成ROM/EPROM, RAM, 总线, 总线逻辑, 定时/计数器, Watchdog, I/O, 串行口等各种必要功能和外设.
- 特点：
  - 一个系列的微控制器具有多种衍生产品;
  - 单片化,体积大大减小,功耗和成本降低,可靠性提高;
  - 是目前嵌入式工业的主流,约占嵌入式系统50%的份额;
  - 多是8位和16位处理器
- 流行的嵌入式微控制器
  - 通用系列:8051,Coldfire的MC683xx（32位）， Cortex-M0/3/4/7
  - 半通用系列:支持I2C,CAN BUS及众多专用MCU和兼容系列

MCU-Based Time and Temperature System



嵌入式DSP

- 嵌入式DSP是专门用于信号处理方面的处理器，其在系统结构和指令算法方面进行了特殊设计，具有很高的编译效率和指令执行速度。
- 应用领域：
  - 数字滤波
  - 频谱分析
  - FFT
- 流行的嵌入式DSP
  - TMS320C2000系列（TI）
  - MCS-296（Intel）

嵌入式SoC

- 嵌入式SoC是追求产品系统最大包容的集成器件。绝大多数系统构件都在一个系统芯片内部。
- 特点：
  - 结构简洁
  - 体积小、功耗小
  - 可靠性高
  - 设计生产效率高
- 流行的SoC
  - 通用系列包括Siemens公司的TriCore、Motorola公司的M-Core、某些ARM系列器件、Echelon公司和Motoro-la公司联合研制的Neuron芯片等。
  - 专用SoC一般专用于某个或某类系统中，不为一般用户所知。

TI Embedded Processing Portfolio

TI Embedded Processors

Microcontrollers (MCUs)

16-bit ultra-low power MCUs

MSP430™

Up to 25 MHz

Flash
1 KB to 256 KB

Analog I/O, ADC
LCD, USB, RF

Measurement, Sensing, General Purpose

$0.25 to $9.00


32-bit real-time MCUs

C2000™
Delfino™
Piccolo™

40MHz to 300 MHz

Flash, RAM
16 KB to 512 KB

PWM, ADC, CAN, SPI, I²C

Motor Control, Digital Power, Lighting, Ren. Enrgy

$1.50 to $20.00


ARM®-Based Processors

32-bit ARM Cortex™-M3 MCUs

Stellaris®
ARM® Cortex™-M3

Up to 100 MHz

Flash
8 KB to 256 KB

USB, ENET
MAC+PHY CAN,
ADC, PWM, SPI

Connectivity, Security, Motion Control, HMI, Industrial Automation

$1.00 to $8.00


ARM Cortex-A8 MPUs

Sitara™
ARM® Cortex™-A8
& ARM9

300MHz to >1GHz

Cache,
RAM, ROM
USB, CAN,
PCIe, EMAC

Industrial computing, POS & portable data terminals

$5.00 to $20.00


Digital Signal Processors (DSPs)

DSP
DSP+ARM

C6000™
DaVinci™
video processors
OMAP™

300MHz to >1Ghz
+Accelerator

Cache
RAM, ROM

USB, ENET,
PCIe, SATA, SPI

Floating/Fixed Point

Video, Audio, Voice, Security, Conferencing

$5.00 to $200.00

Multi-core DSP

C6000™

24.000 MMACS

Cache
RAM, ROM

SRIO, EMAC
DMA, PCIe

Telecom test & meas, media gateways, base stations

$40 to $200.00


Ultra Low power DSP

C5000™

Up to 300 MHz
+Accelerator

Up to 320KB RAM
Up to 128KB ROM
USB, ADC
McBSP, SPI, I²C

Audio, Voice

Medical, Biometrics

$3.00 to $10.00


嵌入式微处理器的特点

- 基础是通用微处理器
- 与通用微处理器相比的区别：
  - 体积小、重量轻、可靠性高
  - 功耗低
  - 成本低：片上存储、引脚与封装、代码密度
  - 工作温度、抗电磁干扰、可靠性等方面增强

Popular Microcontroller

- ARM
- MIPS
- PowerPC

Criteria for Selecting microcontroller

- meeting the computing needs of the task efficiently and cost effectively
  - speed, the amount of ROM and RAM, the number of I/O ports and timers, size, packaging, power consumption
  - easy to upgrade
  - cost per unit
- availability of software development tools
  - assemblers, debuggers, C compilers, emulator, simulator, technical support
- wide availability and reliable sources of the microcontrollers.

Agenda

- Composition of the Embedded System
- Embedded Microprocessor
- ARM

ARM Ltd

- Founded in November 1990
  - Spun out of Acorn Computers
- Designs the ARM range of RISC processor cores
- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers.
  - ARM does not fabricate silicon itself
- Also develop technologies to assist with the design-in of the ARM architecture
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc

ARM Partnership Model

ARM Powered Products

ARM7TDMI

Thumb 架构扩展，提供两个独立的指令集：

- ARM 指令，均为 32位
- Thumb指令，均为 16位
- 两种运行状态，用来选择哪个指令集被执行

内核具有Debug扩展结构

EmbeddedICE 逻辑

增强乘法器 (32x8) 支持64位结果

ARM处理器的分类

- 结构体系版本（Architecture）
  - ARM v4T
  - ARM v5TE
  - ARM v6
  - ARM Cortex (v7)
- 按应用特征分类
  - 应用处理器
    - Application Processor
  - 实时控制处理器
    - Real-time Controller
  - 微控制器
    - Micro-controller

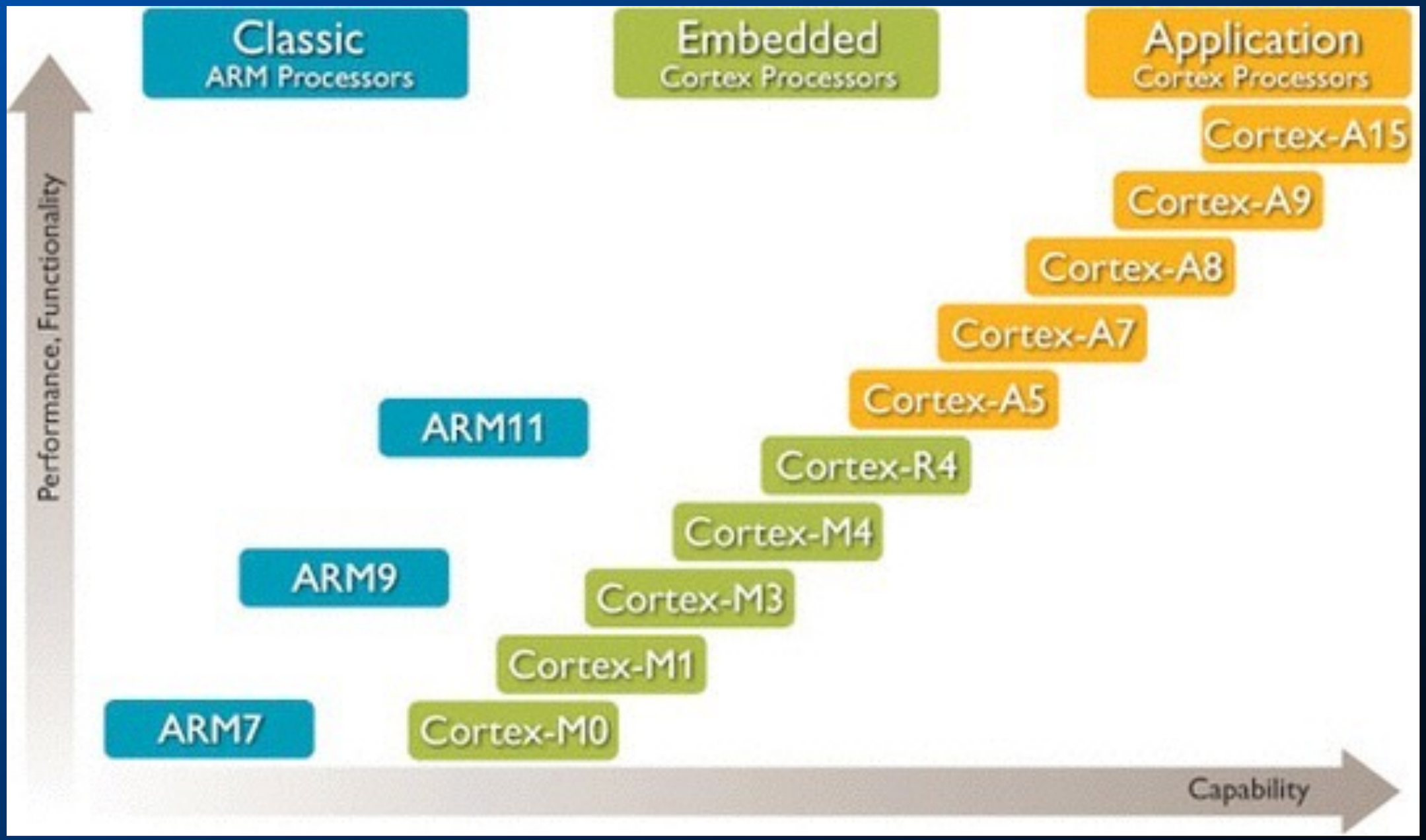
- Processor Family
  - ARM7
  - ARM9
  - ARM10
  - ARM11
  - ARM Cortex

特征：MMU, Cache
最快频率、最高性能、合理功耗

特征：MPU, Cache
实时响应、合理性能、较低功耗

特征：no sub-memory system
一般性能、最低成本、极低功耗

ARM Family


Classic
ARM Processors
Embedded
Cortex Processors
Application
Cortex Processors
Cortex-A15
Cortex-A9
Cortex-A8
Cortex-A7
Cortex-A5
Cortex-R4
Cortex-M4
Cortex-M3
Cortex-M1
Cortex-M0
ARM11
ARM9
ARM7
Performance, Functionality
Capability

ARM Architecture

- Typical RISC architecture:
  - Large uniform register file
  - Load/store architecture
  - Simple addressing modes
  - Uniform and fixed-length instruction fields

ARM Architecture (2)

- Enhancements:
  - Each instruction controls the ALU and shifter
  - Auto-increment and auto-decrement addressing modes
  - Multiple Load/Store
  - Conditional execution

ARM Architecture (3)

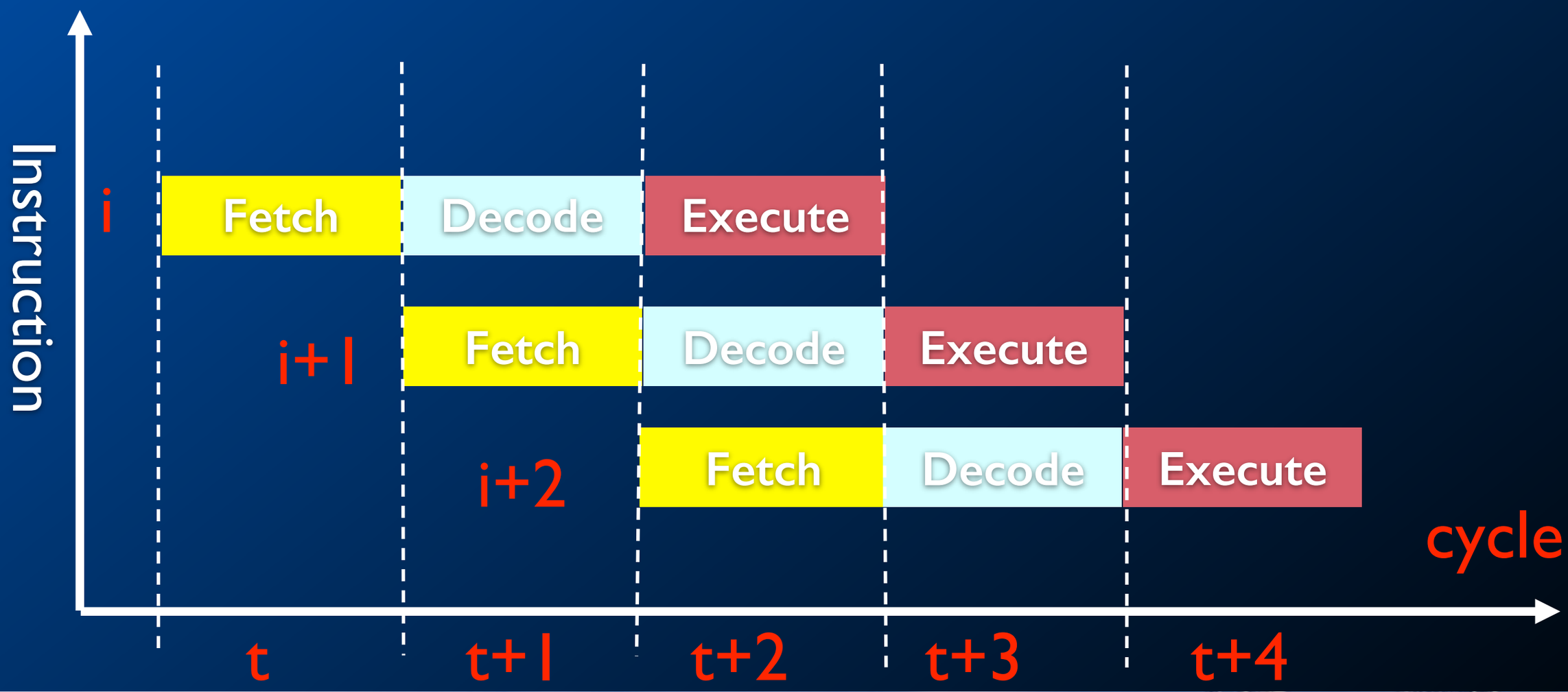
- Results:
  - High performance
  - Low code size
  - Low power consumption
  - Low silicon area

Pipeline Organization

- Increases speed – most instructions executed in single cycle
- Versions:
  - 3-stage (ARM7TDMI and earlier)
  - 5-stage (ARMS, ARM9TDMI)
  - 6-stage (ARM10TDMI)

Pipeline Organization (2)

- 3-stage pipeline: Fetch – Decode - Execute
- Three-cycle latency, one instruction per cycle throughput



Pipeline Organization (3)

- 5-stage pipeline:
  - Reduces work per cycle => allows higher clock frequency
  - Separates data and instruction memory => reduction of CPI (average number of clock Cycles Per Instruction)

Pipeline Organization (4)

- Pipeline flushed and refilled on branch, causing execution to slow down
- Special features in instruction set eliminate small jumps in code to obtain the best flow through pipeline

Operating Modes

- Seven operating modes:
  - User
  - Privileged（特权模式）
    - System (version 4 and above)
    - FIQ
    - IRQ
    - Abort
    - Undefined
    - Supervisor

exception modes（异常模式）

Operating Modes (2)

- User mode:
  - Normal program execution mode
  - System resources unavailable
  - Mode changed by exception only
- Exception modes:
  - Entered upon exception
  - Full access to system resources
  - Mode changed freely

Exceptions

Exception	Mode	Priority	IV Address
Reset	Supervisor	1	0x00000000
Undefined instruction	Undefined	6	0x00000004
Software interrupt	Supervisor	6	0x00000008
Prefetch Abort	Abort	5	0x0000000C
Data Abort	Abort	2	0x00000010
Interrupt	IRQ	4	0x00000018
Fast interrupt	FIQ	3	0x0000001C

Table 1 - Exception types, sorted by Interrupt Vector addresses

ARM Registers

- 31 general-purpose 32-bit registers
- 16 visible, R0 – R15
- Others speed up the exception process

ARM Registers (2)

- Special roles:
  - Hardware
    - R14 – Link Register (LR): optionally holds return address for branch instructions
    - R15 – Program Counter (PC)
  - Software
    - R13 - Stack Pointer (SP)

ARM Registers (3)

- Current Program Status Register (CPSR)
- Saved Program Status Register (SPSR)

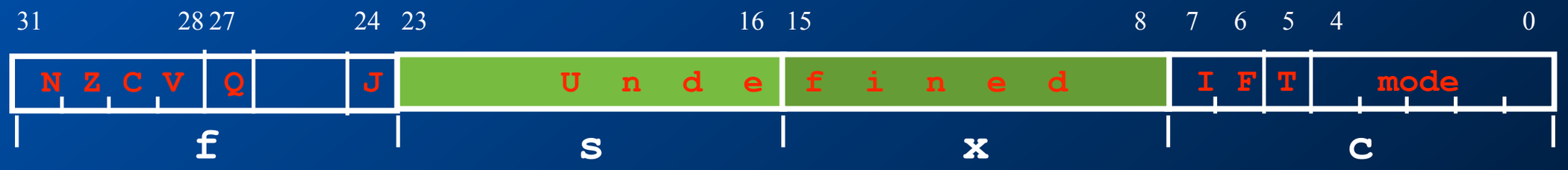
ARM Registers (4)

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8 fia	R8	R8	R8	R8
R9	R9 fia	R9	R9	R9	R9
R10	R10 fia	R10	R10	R10	R10
R11	R11 fia	R11	R11	R11	R11
R12	R12 fia	R12	R12	R12	R12
R13	R13 fia	R13 svc	R13 abt	R13 irq	R13 und
R14	R14 fia	R14 svc	R14 abt	R14 irq	R14 und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR fia	SPSR svc	SPSR abt	SPSR irq	SPSR und

The Registers

- ARM has 37 registers all of which are 32-bits long.
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers
- The current processor mode governs which of several banks is accessible. Each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - the program counter, r15 (pc)
  - the current program status register, cpsr
- Privileged modes (except System) can also access
  - a particular spsr (saved program status register)

Program Status Registers



- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- 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

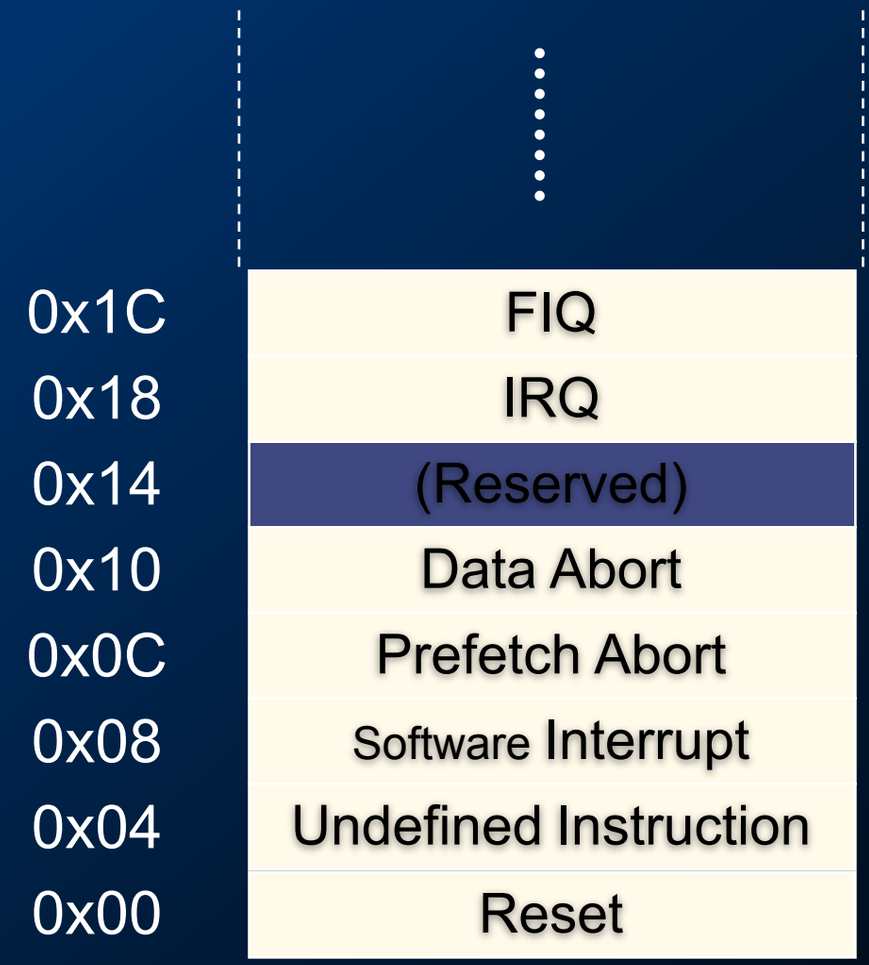
Program Counter (r15)

- 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

Exception Handling

- When an exception occurs, the ARM:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
      - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

  This can only be done in ARM state.



Vector Table

Vector table can be at
0xFFFF0000 on ARM720T
and on ARM9/10 family devices

Endianness

- Relationship between bit and byte/word ordering defines endianness:

little-endian
(default)

big-endian

ARM data types

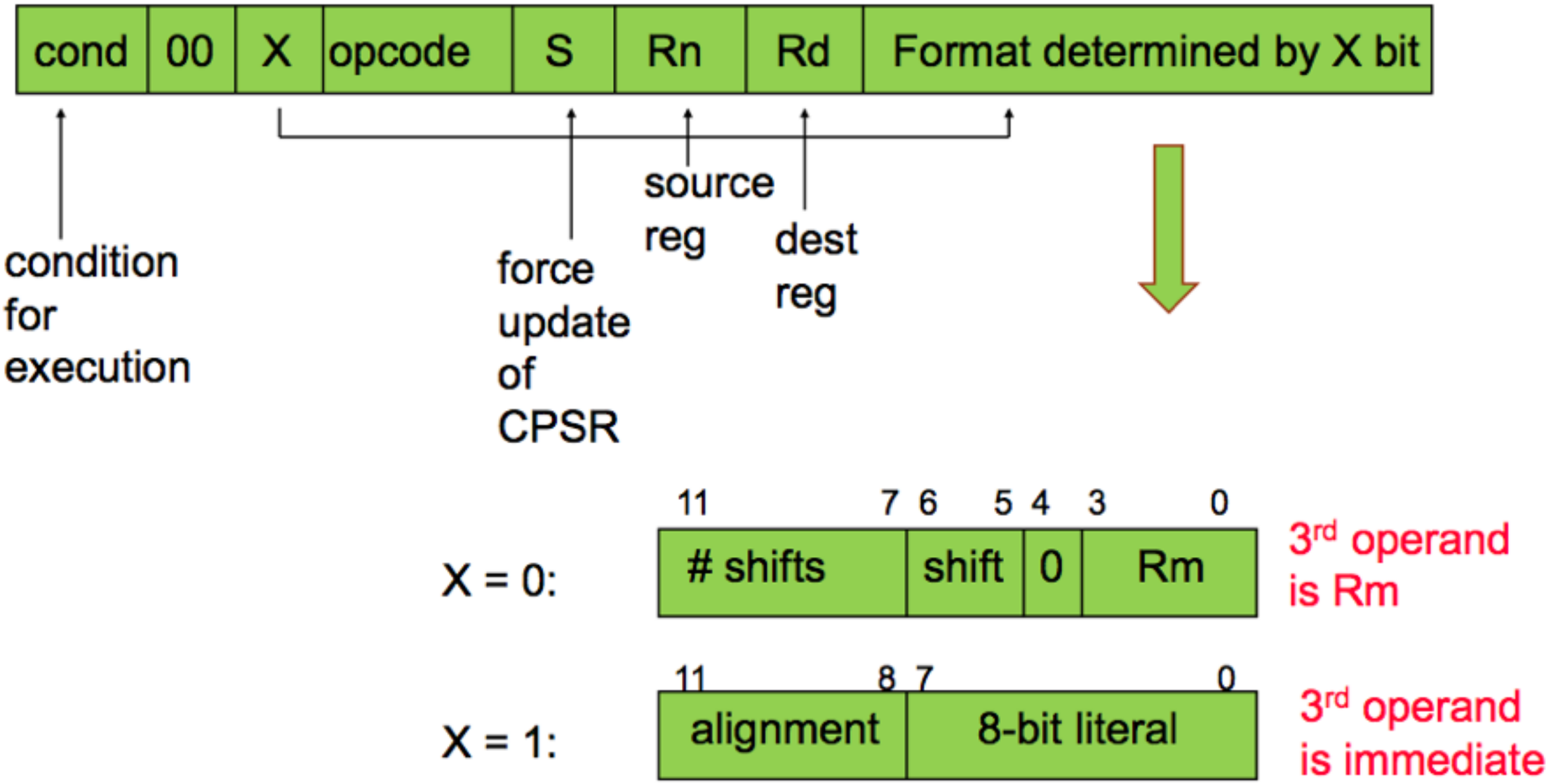
- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
    - Address 4 starts at byte 4.
- Can be configured at power-up as either little- or big-endian mode.

ARM assembly language

- Fairly standard assembly language:

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

ARM Instruction Code Format



Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
    - This improves code density and performance by reducing the number of forward branch instructions.

```
CMP   r3,#0
BEQ   skip
ADD   r0,r1,r2
skip
```

```
CMP   r3,#0
ADDNE r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".
    - loop

```
…
SUBS r1,r1,#1
BNE loop
```

decrement r1 and set flags

if Z flag clear then branch

Condition Codes

- The possible condition codes are listed below:
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);

        CMP     r0,#0
        MOVEQ   r0,#1
        BLEQ    func
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;
if (a>0)  x=1;

        CMP     r0,#0
        MOVEQ   r1,#0
        MOVGT   r1,#1
```

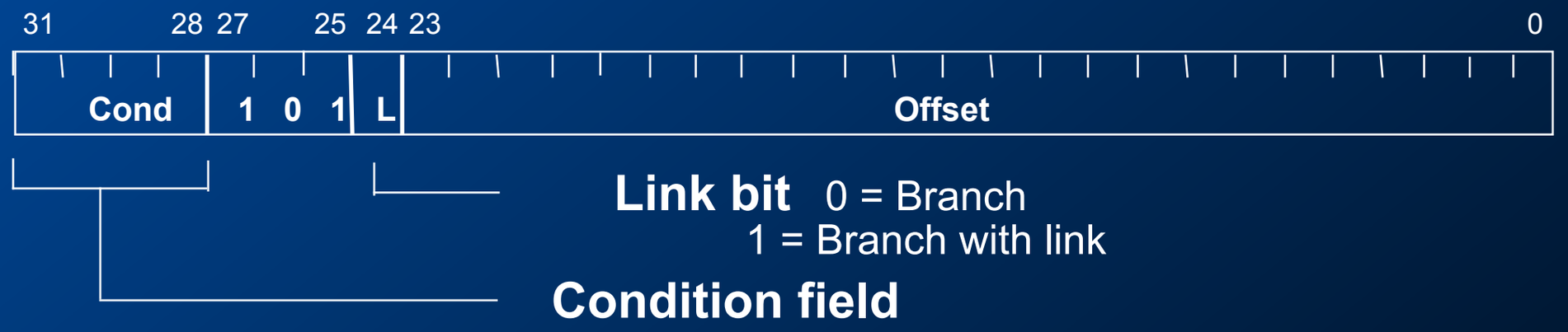
- Use conditional compare instructions

```
if (a==4 || a==10) x=0;

        CMP     r0,#4
        CMPNE   r0,#10
        MOVEQ   r1,#0
```

Branch instructions

- Branch : B{<cond>} label
- Branch with Link : BL{<cond>} subroutine_label



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

Data processing Instructions

- Consist of :
  - Arithmetic: ADD ADC SUB SBC RSB RSC
  - Logical: AND ORR EOR BIC
  - Comparisons: CMP CMN TST TEQ
  - Data movement: MOV MVN
- These instructions only work on registers, NOT memory.
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
    - Comparisons set flags only - they do not specify Rd
    - Data movement does not specify Rn
- Second operand is sent to the ALU via barrel shifter.

The Barrel Shifter

LSL : Logical Left Shift

Multiplication by a power of 2

ASR: Arithmetic Right Shift

Division by a power of 2,
preserving the sign bit

LSR : Logical Shift Right

Division by a power of 2

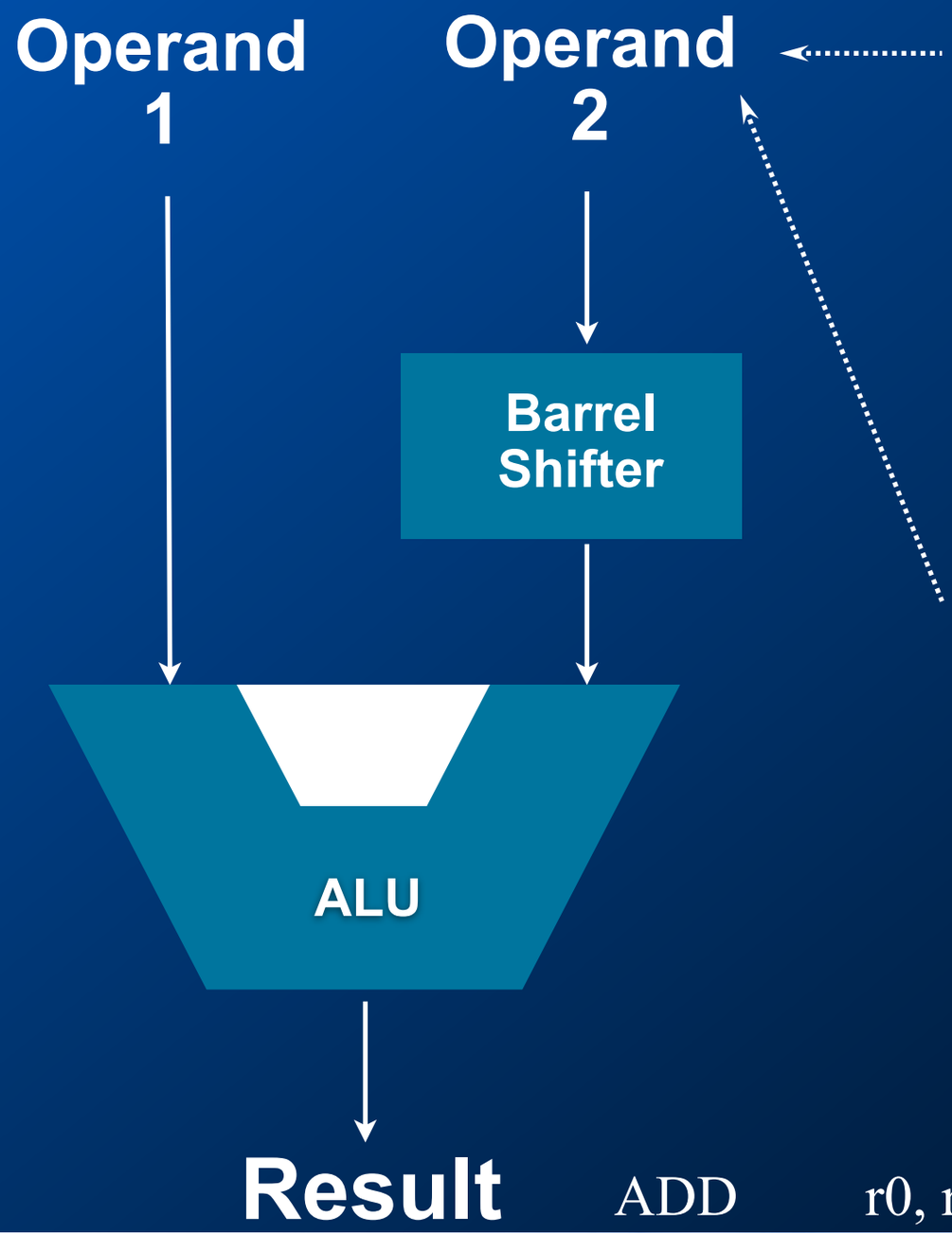
ROR: Rotate Right

Bit rotate with wrap around
from LSB to MSB

RRX: Rotate Right Extended

Single bit rotate with wrap around from CF to MSB

Using the Barrel Shifter: The Second Operand



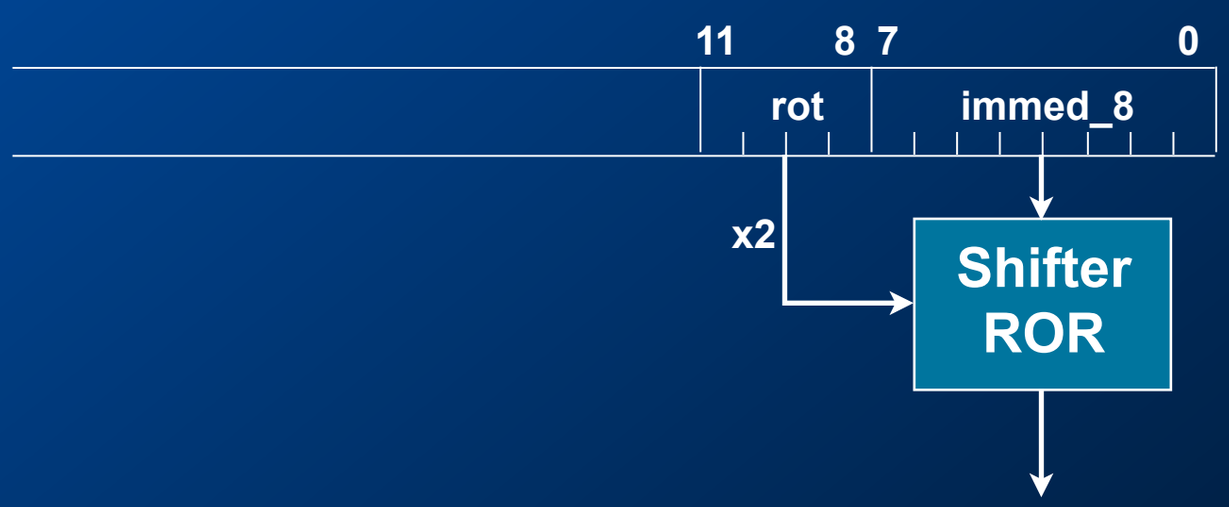
- Register, optionally with shift operation
  - Shift value can be either be:
    - 5 bit unsigned integer
    - Specified in bottom byte of another register.
  - Used for multiplication by constant

- Immediate value
  - 8 bit number, with a range of 0-255.
    - Rotated right through even number of positions
  - Allows increased range of 32-bit constants to be loaded directly into registers

```
ADD     r0, r1, r2, LSL#7
```

Immediate constants (1)

- No ARM instruction can contain a 32 bit immediate constant
  - All ARM instructions are fixed as 32 bits long
- The data processing instruction format has 12 bits available for operand2



- 4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2
- Rule to remember is "8-bits shifted by an even number of bit positions".

Immediate constants (2)

- Examples:

- The assembler converts immediate values to the rotate form:
  - MOV r0,#4096 ; uses 0x40 ror 26
  - ADD r1,r2,#0xFF0000 ; uses 0xFF ror 16

- The bitwise complements can also be formed using MVN:
  - MOV r0, #0xFFFFFFFF ; assembles to MVN r0,#0

- Values that cannot be generated in this way will cause an error.

Loading 32 bit constants

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  - LDR rd, =const
- This will either:
  - Produce a MOV or MVN instruction to generate the value (if possible).

  or
  - Generate a LDR instruction with a PC-relative address to read the constant from a literal pool (Constant data area embedded in the code).
- For example
  - LDR r0,=0xFF => MOV r0,#0xFF
  - LDR r0,=0x55555555 => LDR r0,[PC,#Imm12]

    …

    …

    DCD 0x55555555
- This is the recommended way of loading constants into a register

Multiply

- Syntax:
  - MUL{<cond>}{S} Rd, Rm, Rs — Rd = Rm * Rs
  - MLA{<cond>}{S} Rd,Rm,Rs,Rn — Rd = (Rm * Rs) + Rn
  - [U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs — RdHi,RdLo := Rm*Rs
  - [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs — RdHi,RdLo := (Rm*Rs)+RdHi,RdLo
- Cycle time
  - Basic MUL instruction
    - 2-5 cycles on ARM7TDMI
    - 1-3 cycles on StrongARM/XScale
    - 2 cycles on ARM9E/ARM102xE
  - +1 cycle for ARM9TDMI (over ARM7TDMI)
  - +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
  - +1 cycle for "long"
- Above are "general rules" - refer to the TRM for the core you are using for the exact details

Single register data transfer

LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

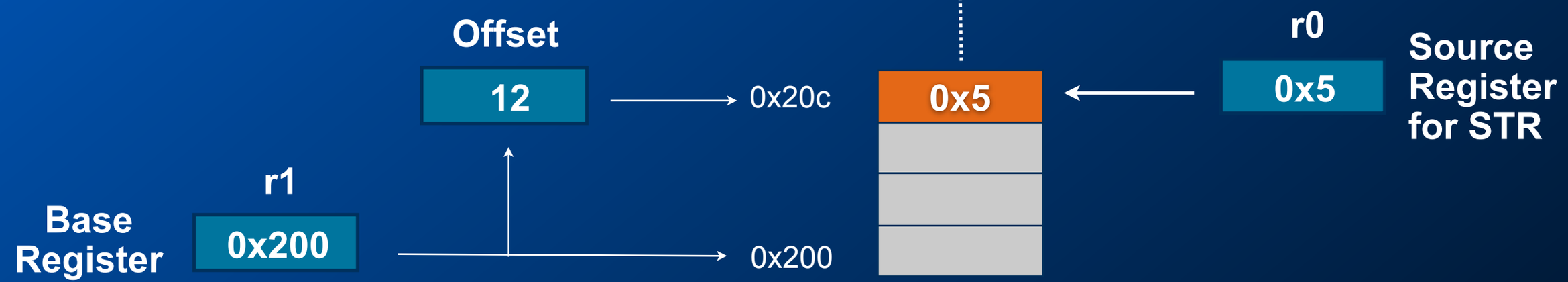
- Memory system must support all access sizes
- Syntax:
  - LDR{<cond>}{<size>} Rd, <address>
  - STR{<cond>}{<size>} Rd, <address>
  - e.g. LDREQB

Address accessed

- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
  - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).
    LDR r0,[r1,#8]
  - A register, optionally shifted by an immediate value
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]
- This can be either added or subtracted from the base register:
    LDR r0,[r1,#-8]
    LDR r0,[r1,-r2]
    LDR r0,[r1,-r2,LSL#2]
- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (ie 0-255 bytes).
  - A register (unshifted).
- Choice of pre-indexed or post-indexed addressing

Pre or Post Indexed Addressing?

Pre-indexed: `STR r0,[r1,#12]`



Auto-update form: `STR r0,[r1,#12]!`

Post-indexed: STR r0,[r1],#12

LDM / STM operation

- Syntax:

  `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>

- 4 addressing modes:

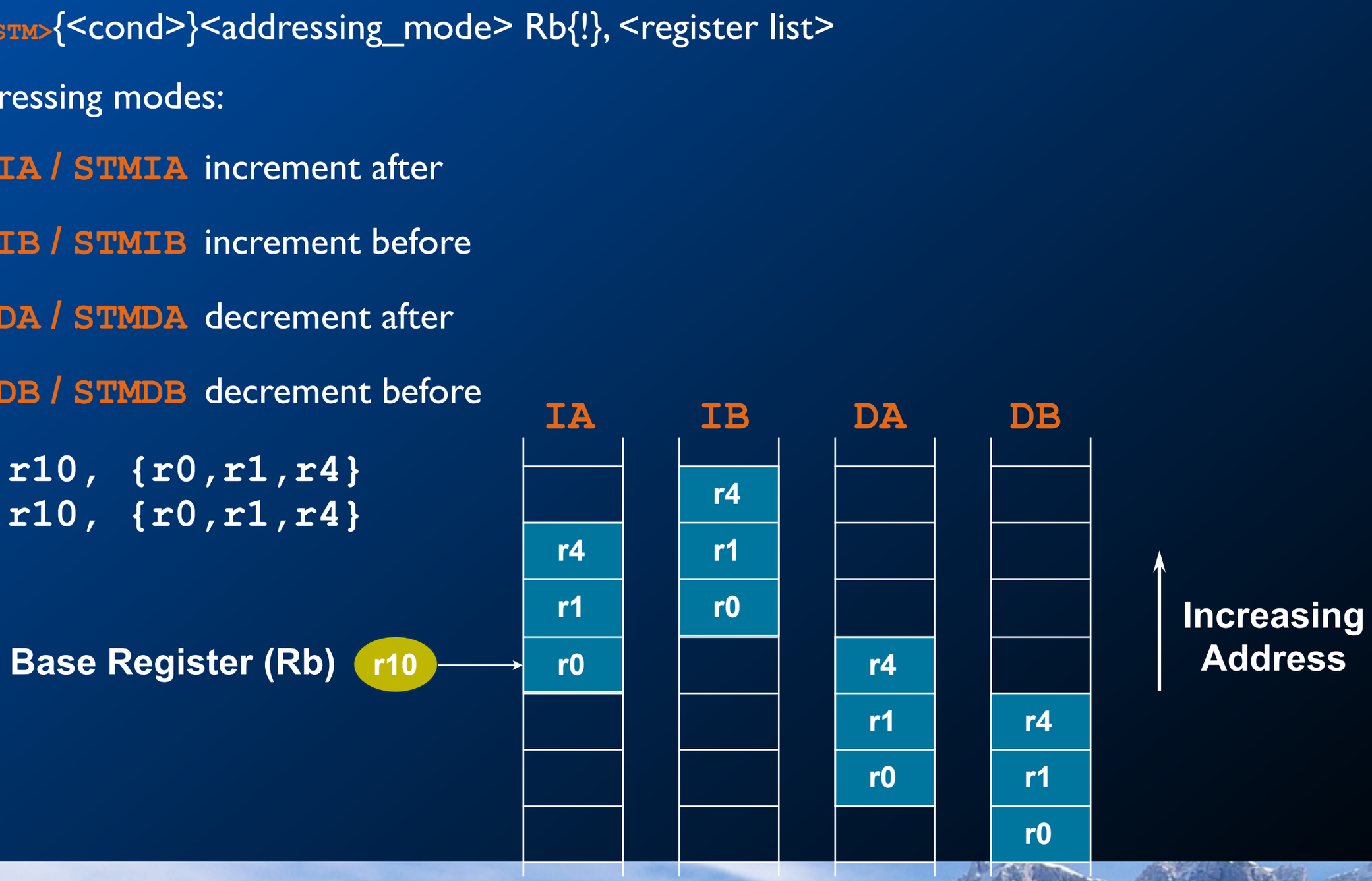
  `LDMIA / STMIA` increment after

  `LDMIB / STMIB` increment before

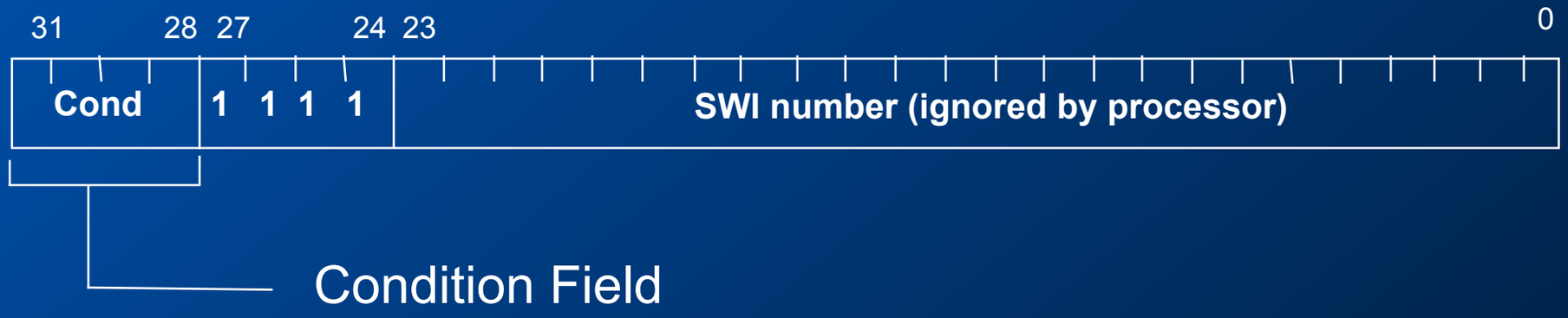
  `LDMDA / STMDA` decrement after

  `LDMDB / STMDB` decrement before

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```

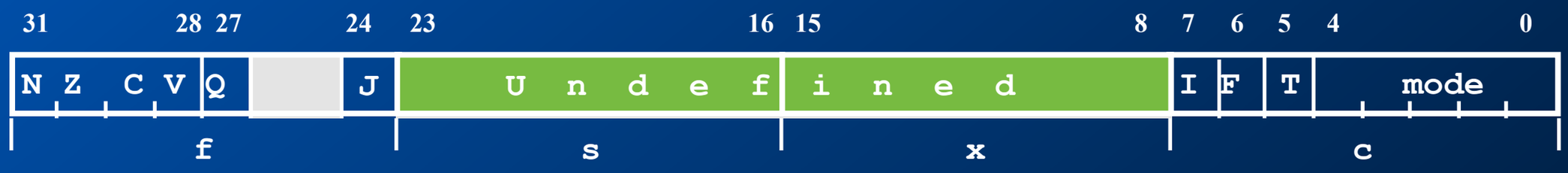


Software Interrupt (SWI)



- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:
  - SWI{<cond>} <SWI number>

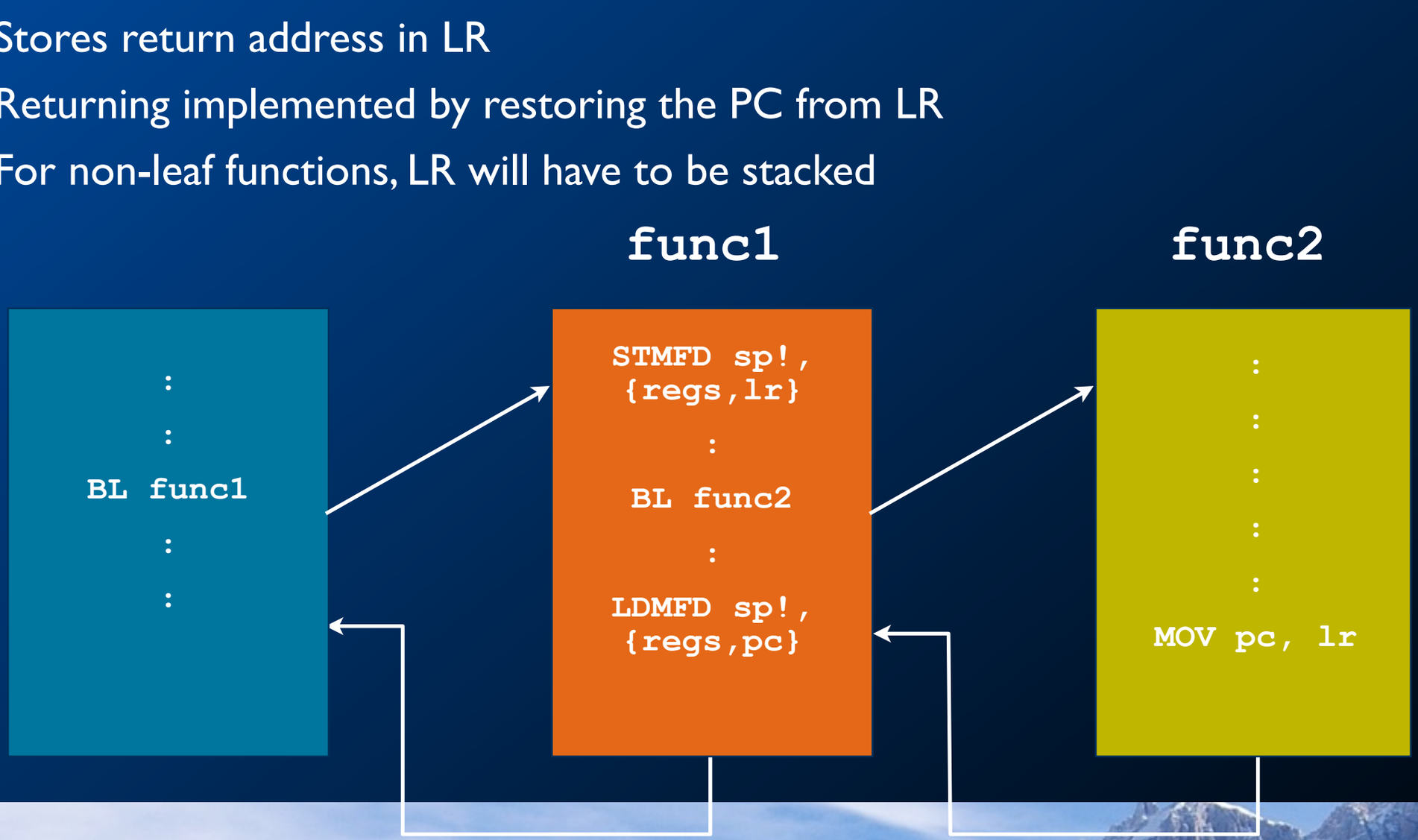
PSR Transfer Instructions



- Read-modify-write strategy.
- MRS and MSR allow contents of CPSR / SPSR to be transferred to/from a general purpose register.
- Syntax:
  - MRS{<cond>} Rd,<psr> ; Rd = <psr>
  - MSR{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm
- where
  - <psr> = CPSR or SPSR
  - [_fields] = any combination of 'fsxc'
- Also an immediate form
  - MSR{<cond>} <psr_fields>,#Immediate
- In User Mode, all bits can be read but only the condition flags (_f) can be written.

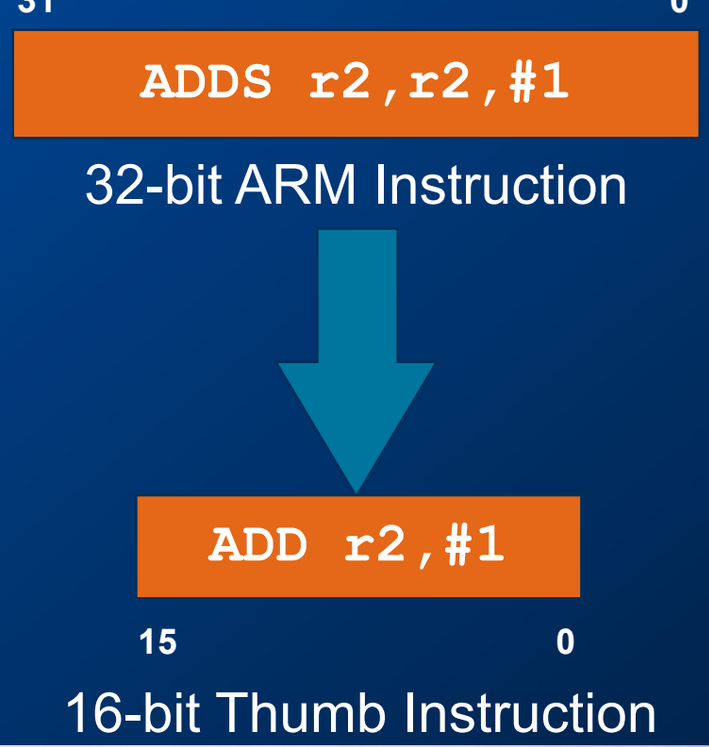
ARM Branches and Subroutines

- B <label>
  - PC relative. ±32 Mbyte range.
- BL <subroutine>
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked



Thumb

- Thumb is a 16-bit instruction set
  - Optimised for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has additional execution state - Thumb
  - Switch between ARM and Thumb using BX instruction



For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

Example: Part 1 of 2440init.s

```
b ResetHandler;0x00
b HandlerUndef ;handler for Undefined mode  ;0x04
b HandlerSWI     ;handler for SWI interrupt  ;0x08
b HandlerPabort ;handler for PAbort    ;0x0c
b HandlerDabort ;handler for DAbort    ;0x10
b .          ;reserved;0x14
b HandlerIRQ  ;handler for IRQ interrupt  ;0x18
b HandlerFIQ  ;handler for FIQ interrupt  ;0x1c
```

Example: Part 2 of 2440init.s

```
USERMODE    EQU  0x10
FIQMODE     EQU  0x11
IRQMODE     EQU  0x12
SVCMODE     EQU  0x13
ABORTMODE   EQU  0x17
UNDEFMODE   EQU  0x1b
MODEMASK    EQU  0x1f
NOINT       EQU  0xc0
```

Example: Part 3 of 2440init.s

```
;function initializing stacks
mrs r0,cpsr
bic r0,r0,#MODEMASK
orr r1,r0,#UNDEFMODE|NOINT
msr cpsr_cxsf,r1  ;UndefMode
ldr sp,=UndefStack  ; UndefStack=0x33FF_5C00

orr r1,r0,#ABORTMODE|NOINT
msr cpsr_cxsf,r1  ;AbortMode
ldr sp,=AbortStack  ; AbortStack=0x33FF_6000

orr r1,r0,#IRQMODE|NOINT
msr cpsr_cxsf,r1  ;IRQMode
ldr sp,=IRQStack  ; IRQStack=0x33FF_7000
```

Thank you!