# Principles and Practices of Microcontroller (Embedded System Design I) -Arm Architecture and ISA

**Gang Chen (陈刚)**

Associate Professor

Institute of Unmanned Systems
School of data and computer science
Sun Yat-Sen University

https://www.usilab.cn/team/chengang/

# Architecture

**In the context of computers,**
**what does *architecture* mean?**

# Architecture has many meanings

- **Computer Organization  (or Microarchitecture)**
  - Control and data paths
  - I/D pipeline design
  - Cache design
  - …

- **System Design (or Platform Architecture)**
  - Memory and I/O buses
  - Memory controllers
  - Direct memory access
  - …

- **Instruction Set Architecture (ISA)**

## What is an *Instruction Set Architecture (ISA)?*

# An ISA defines the hardware/software interface

- A “contract” between architects and programmers
- Instruction set
- Register set
- Memory and addressing modes
- Word sizes
- Data formats
- Operating modes
- Condition codes
- Calling conventions

# ARM Architecture roadmap

## 4T

**ARM7TDMI**
**ARM922T**

Thumb instruction set

## 5TE

**ARM926EJ-S**
**ARM946E-S**
**ARM966E-S**

Improved ARM/Thumb Interworking

DSP instructions

**Extensions:**

Jazelle (5TEJ)

## 6

**ARM1136JF-S**
**ARM1176JZF-S**
**ARM11 MPCore**

SIMD Instructions

Unaligned data support

**Extensions:**

Thumb-2 (6T2)

TrustZone (6Z)

Multicore (6K)

## 7

**Cortex-A8/R4/M3/M1**

Thumb-2

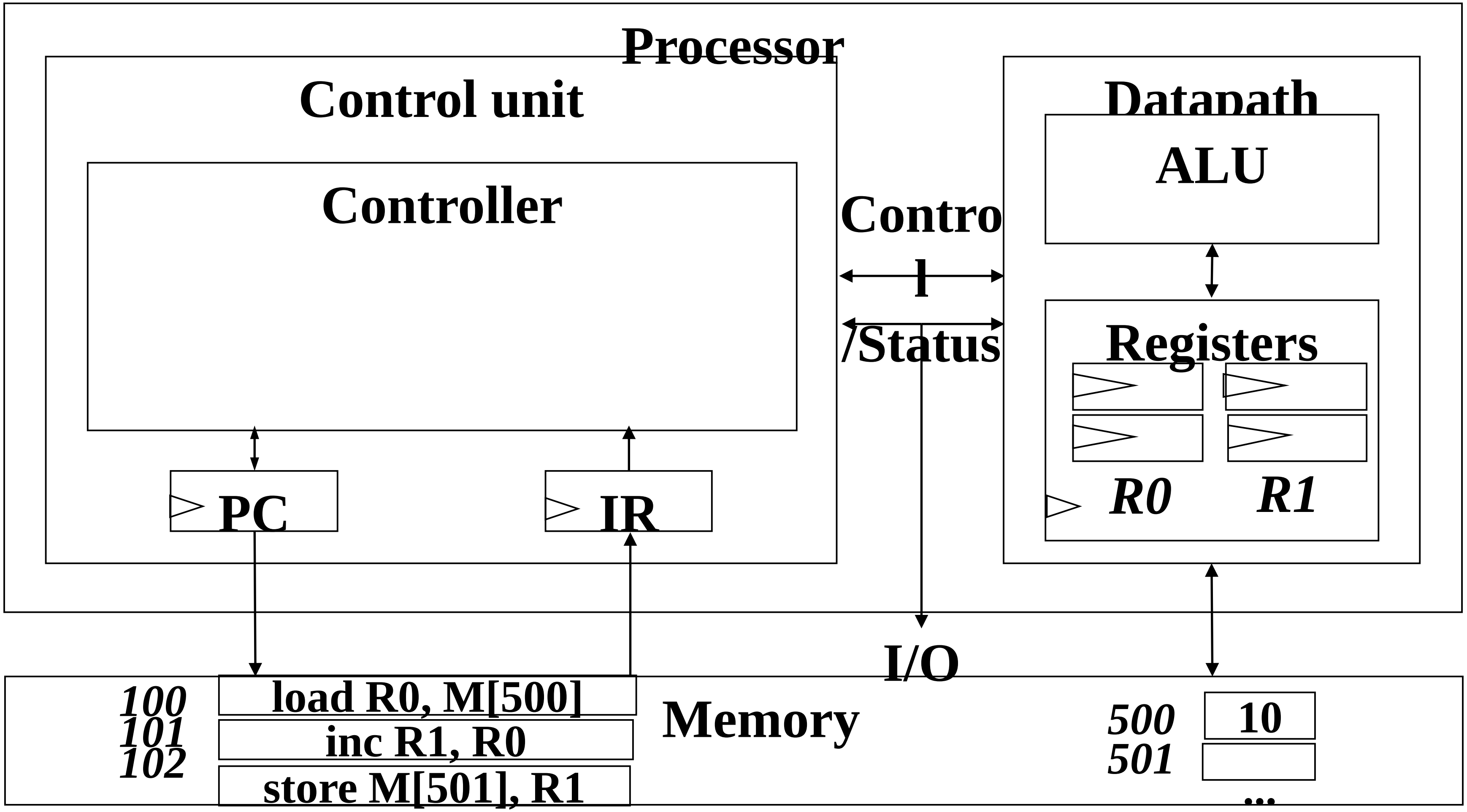
**Extensions:**

v7A (applications) – NEON

v7R (real time) – HW Divide

V7M (microcontroller) – HW Divide and Thumb-2 only

# An example



Processor

Control unit

Controller

PC

IR

Contro
l
/Status

Datapath

ALU

Registers

*R0*

*R1*

I/O

Memory

*100* load R0, M[500]

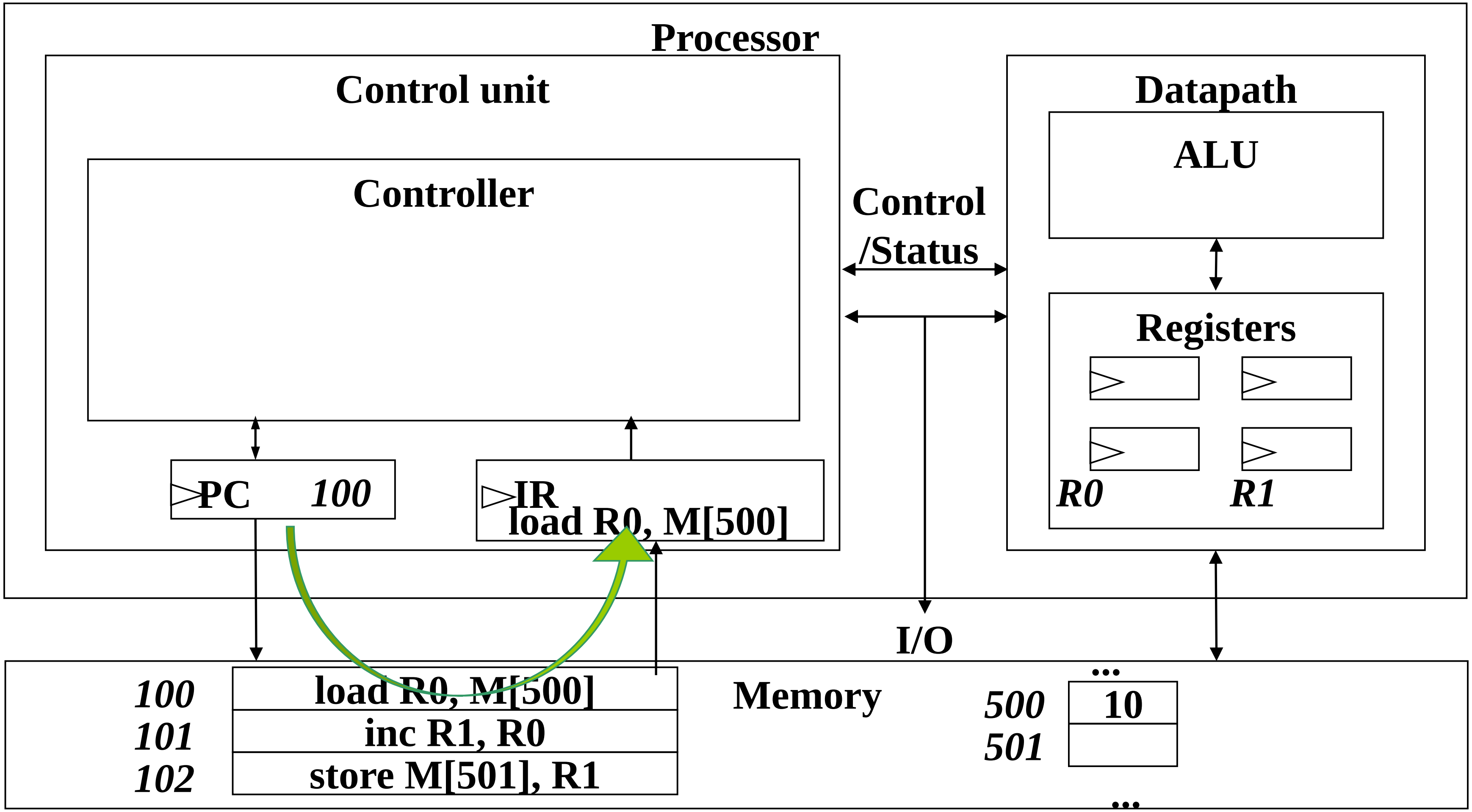
*101* inc R1, R0

*102* store M[501], R1

*500* 10

*501*

...



Processor

Control unit

Controller

PC 100

IR
load R0, M[500]

Control
/Status

Datapath

ALU

Registers

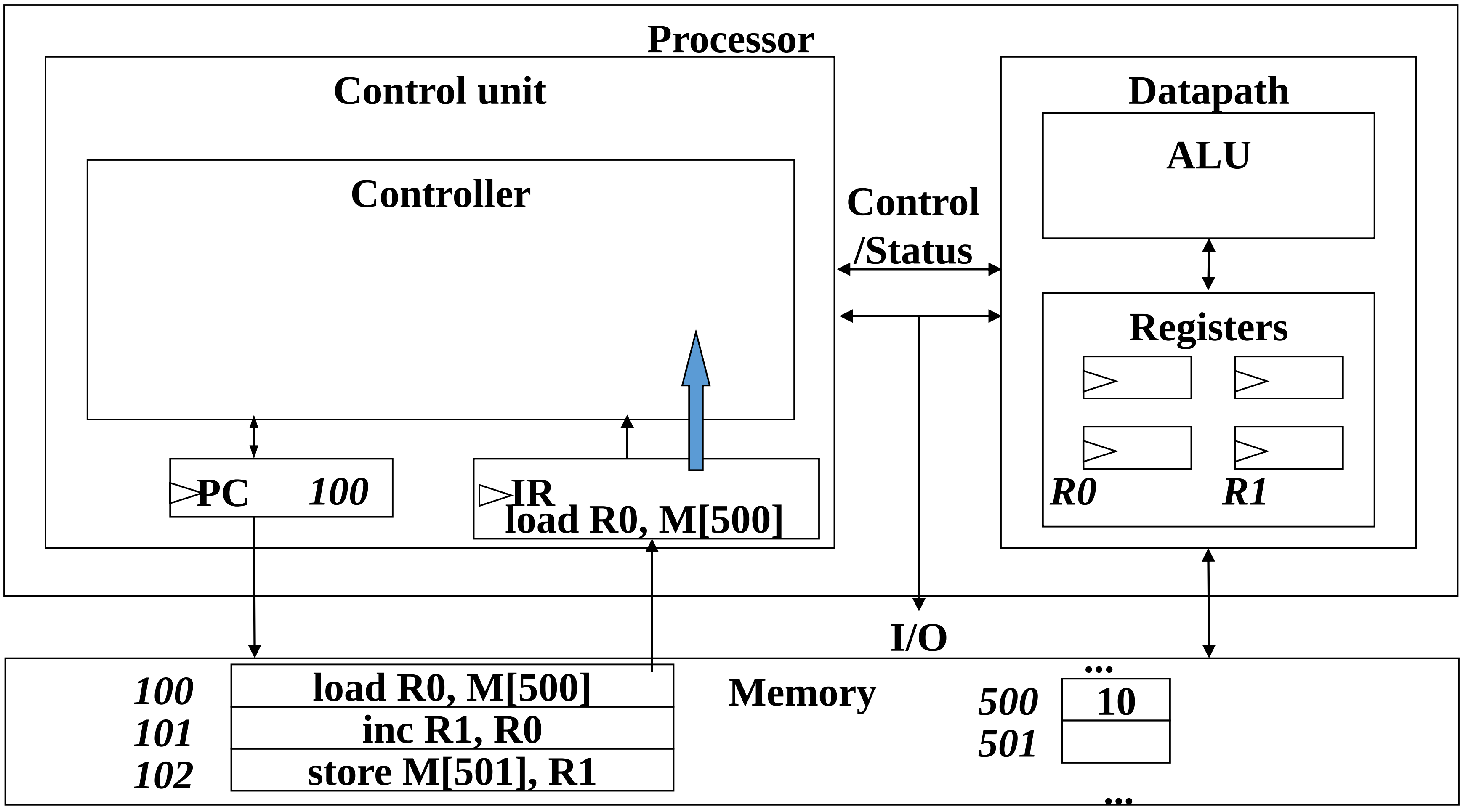
R0 R1

I/O

Memory

100 load R0, M[500]
101 inc R1, R0
102 store M[501], R1

...
500 10
501
...



Processor

Control unit

Controller

PC 100

IR
load R0, M[500]

Control
/Status

Datapath

ALU

Registers

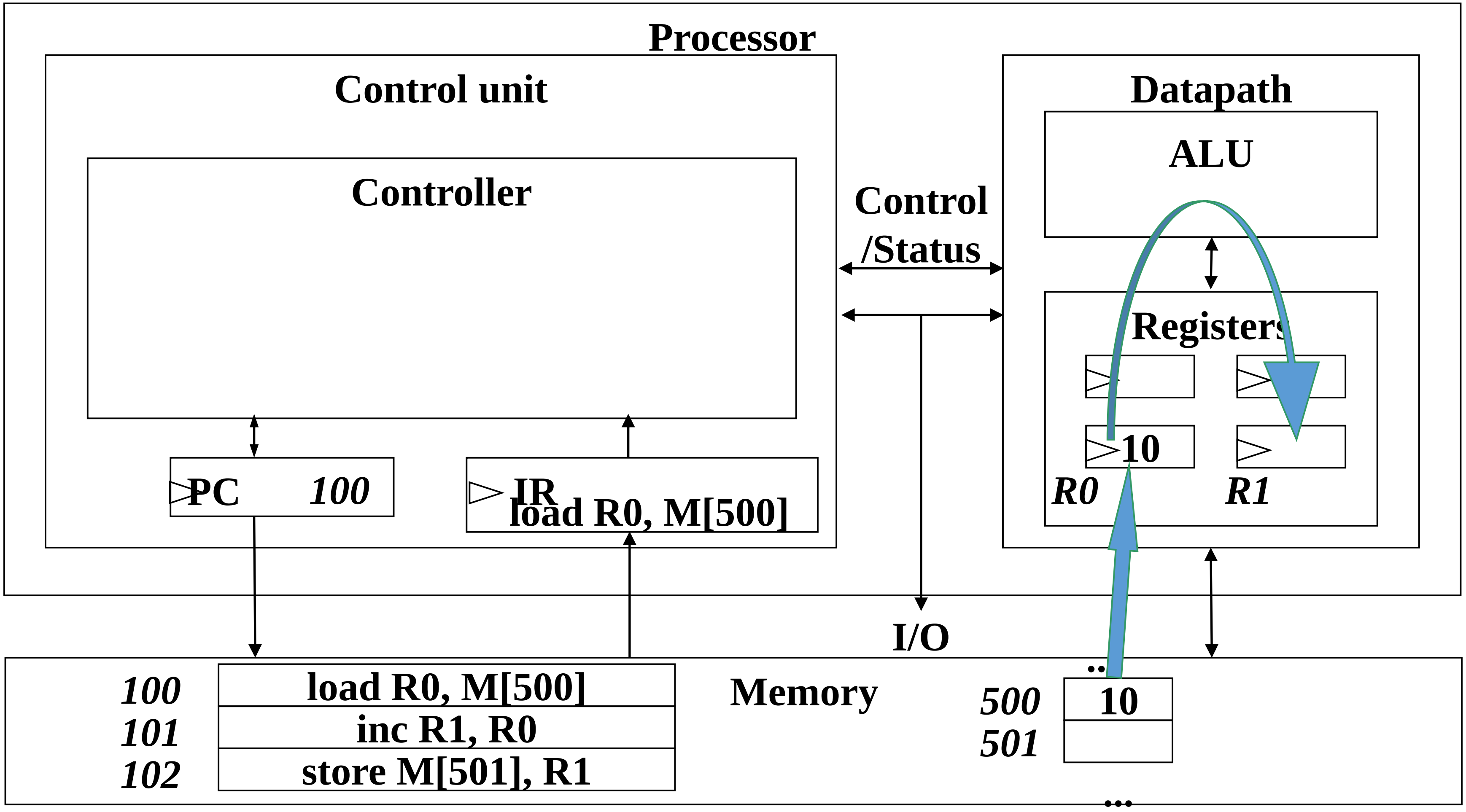
R0 R1

I/O

Memory

100 load R0, M[500]
101 inc R1, R0
102 store M[501], R1

...
500 10
501
...



Processor

Control unit

Controller

PC *100*

IR
load R0, M[500]

Control
/Status

Datapath

ALU

Registers

10

*R0*

*R1*

I/O

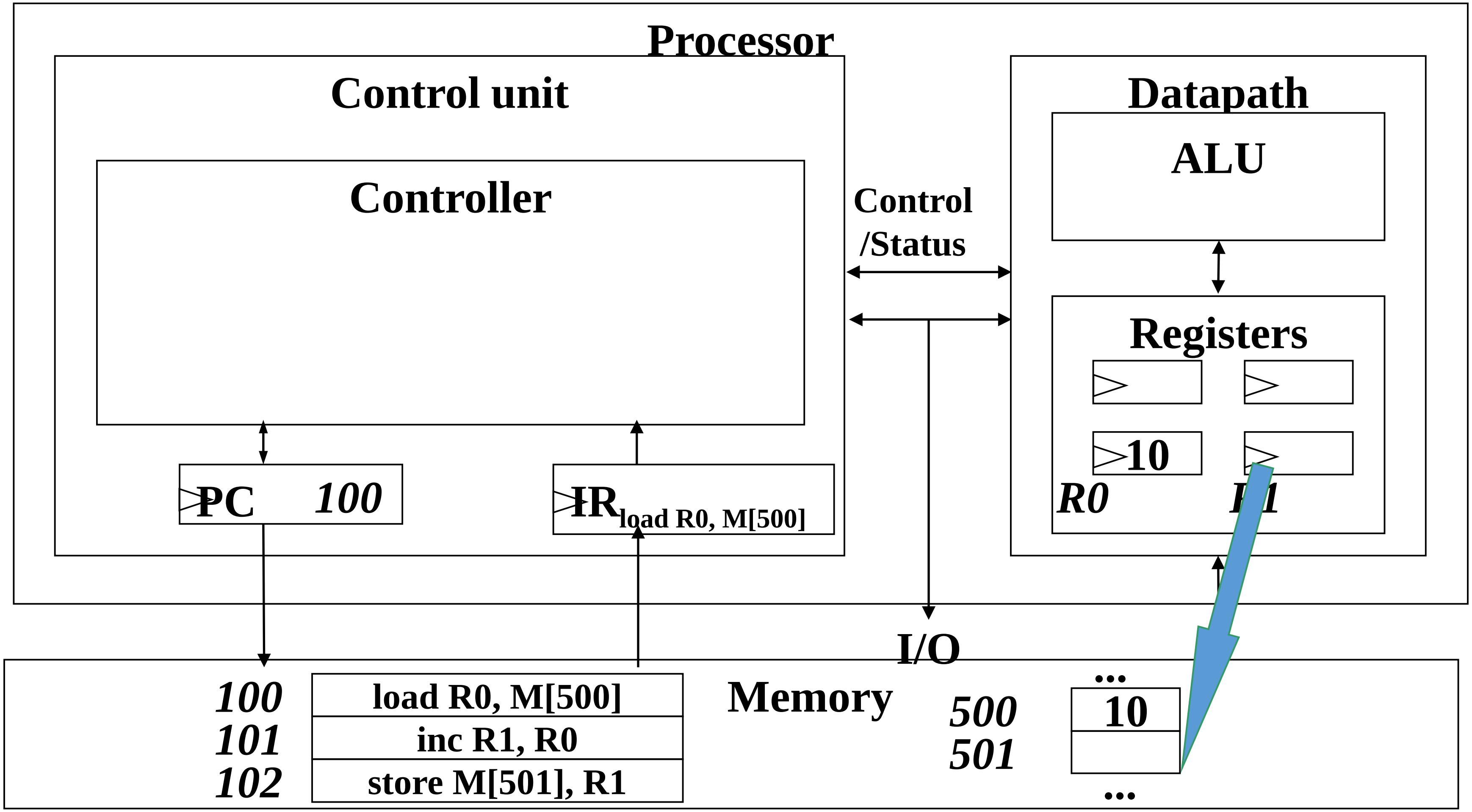
Memory

*100* load R0, M[500]
*101* inc R1, R0
*102* store M[501], R1

..

*500* 10
*501*

...



Processor

Control unit

Controller

PC 100

IR load R0, M[500]

Control /Status

Datapath

ALU

Registers

10

R0

R1

I/O

Memory

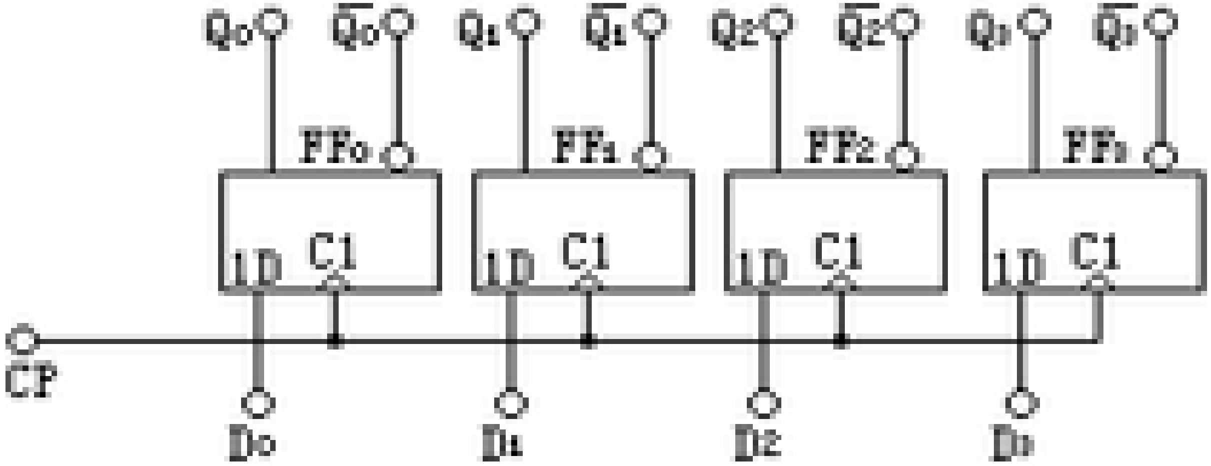
| | |
|---|---|
| 100 | load R0, M[500] |
| 101 | inc R1, R0 |
| 102 | store M[501], R1 |

...

| | |
|---|---|
| 500 | 10 |
| 501 | |

...

# What is register?



由 D 触发器组成的四位数码寄存器

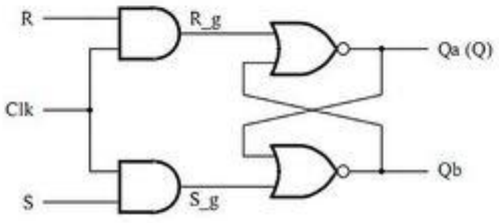


Figure 1. A gated RS latch circuit.

# ARM Cortex-M3 ISA

## Instruction Set

```
ADD Rd, Rn, <op2>
```

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

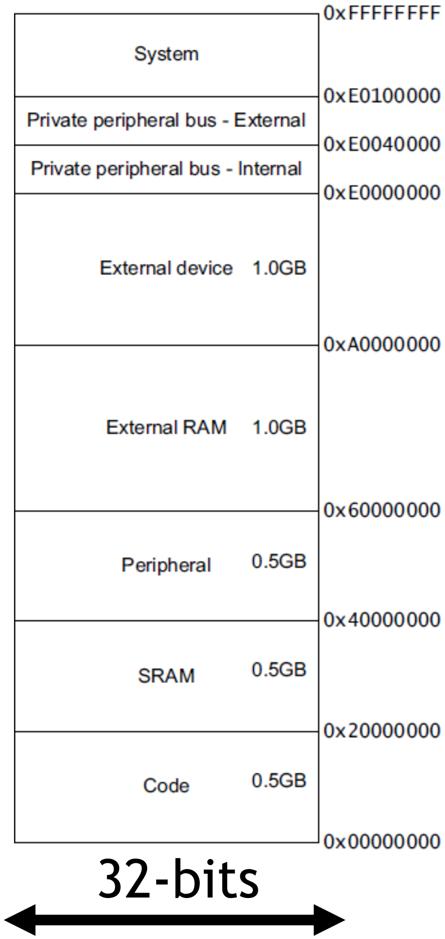
## Register Set

| |
|---|
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13 (SP) |
| R14 (LR) |
| R15 (PC) |
| xPSR |

32-bits

Endianess

## Address Space

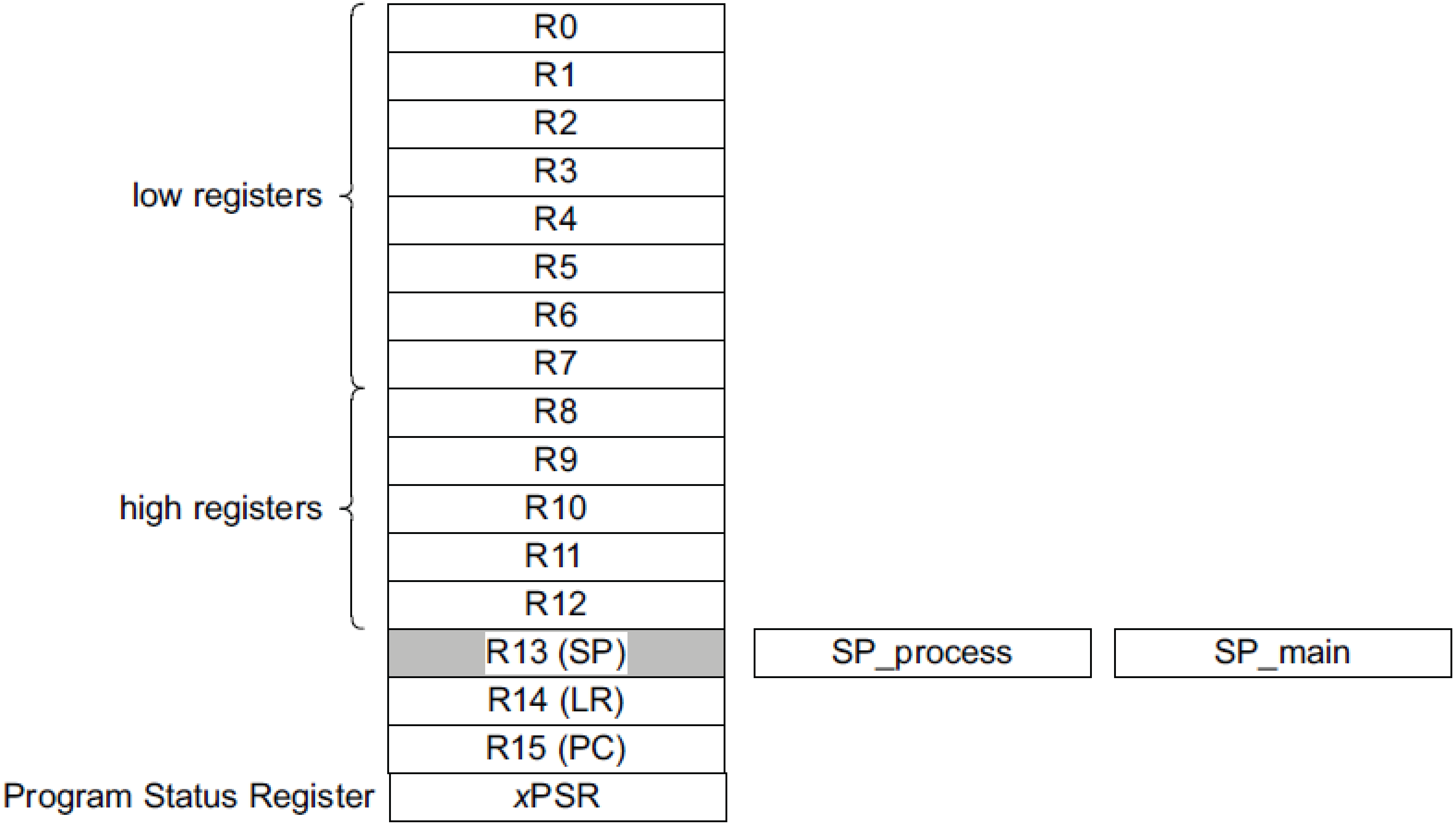


0xFFFFFFFF
System
0xE0100000
Private peripheral bus - External
0xE0040000
Private peripheral bus - Internal
0xE0000000
External device 1.0GB
0xA0000000
External RAM 1.0GB
0x60000000
Peripheral 0.5GB
0x40000000
SRAM 0.5GB
0x20000000
Code 0.5GB
0x00000000

32-bits

Endianess

# Registers



low registers

R0

R1

R2

R3

R4

R5

R6

R7

high registers

R8

R9

R10

R11

R12

R13 (SP)

SP_process

SP_main

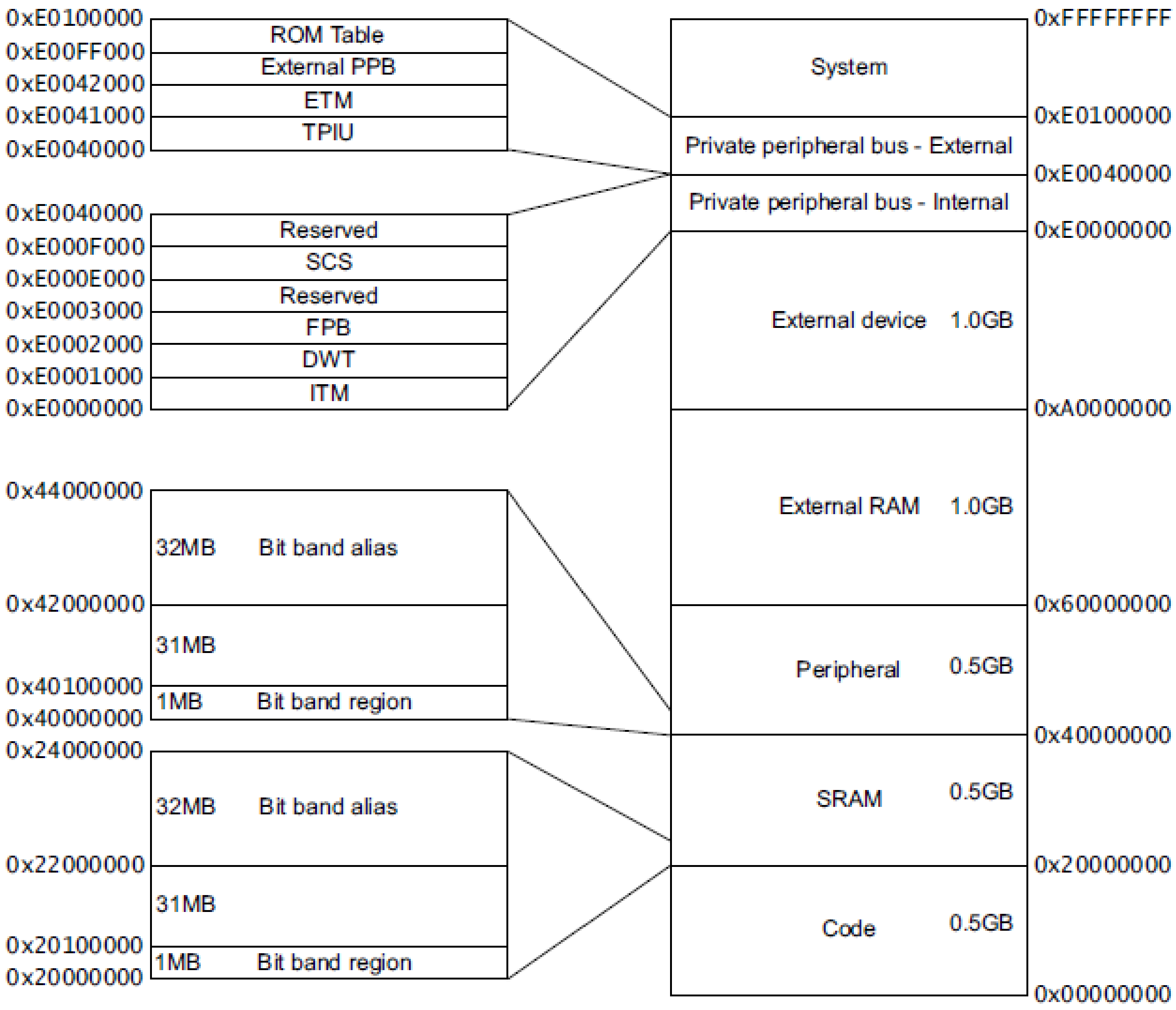
R14 (LR)

R15 (PC)

Program Status Register

*x*PSR

# Address Space



# Register R14

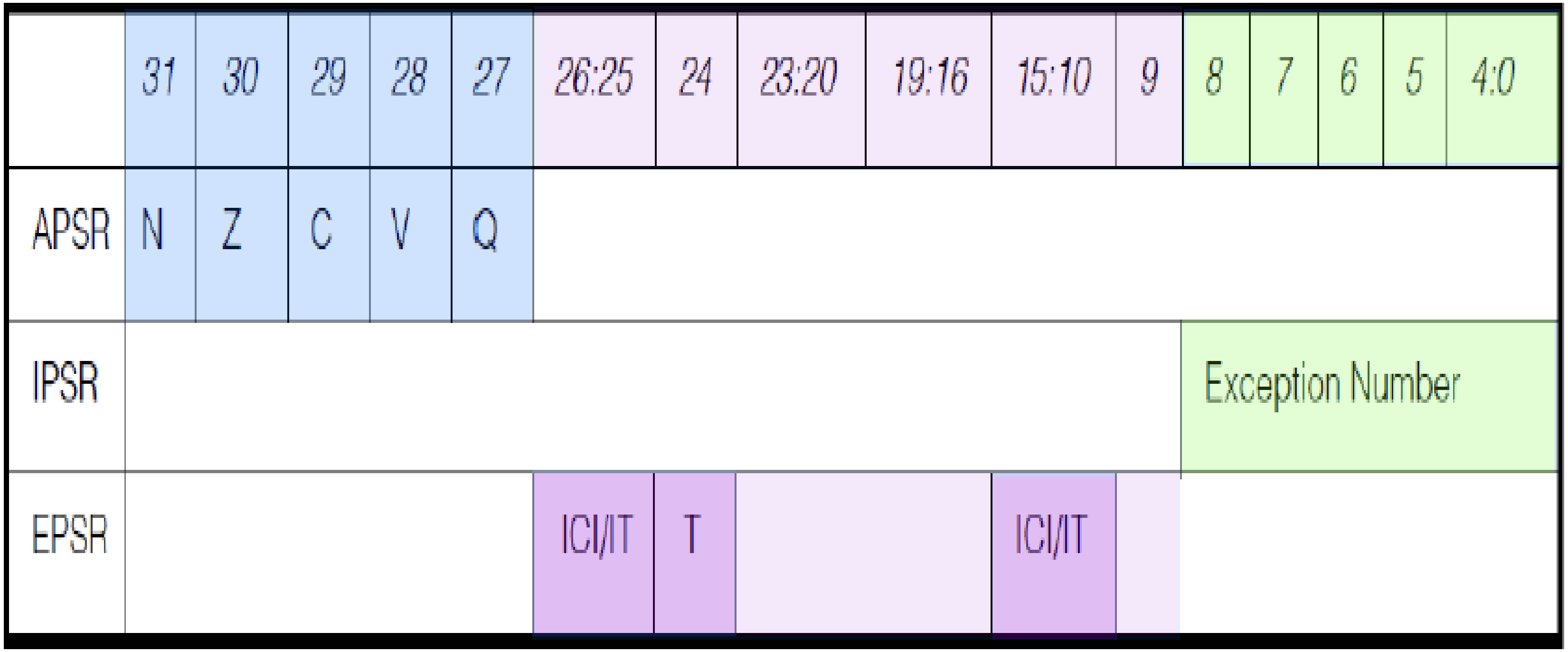
- R14 是连接寄存器（LR）。在一个汇编程序中，你可以把它写成LR 和R14。LR 用于在调用子程序时存储返回地址。例如，当你在使用BL(分支并连接，Branch and Link)指令时，就自动填充LR 的值。
- main ;主程序
- …
- BL function1 ; 使用“分支并连接”指令呼叫function1
- ; PC= function1，并且LR=main 的下一条指令地址
- …
- Function1
- … ; function1 的代码
- BX LR ; 函数返回（如果function1 要使用LR，必须在使用前PUSH，; 否则返回时程序就可能跑飞）

# Register R15： PC

- R15 是程序计数器， 在汇编代码中你也可以使用名字“PC”来访问它。因为CM3 内部使用了指令流水线，读PC 时返回的值是当前指令的地址+4。比如说：
- 0x1000: MOV R0, PC ; R0 = 0x1004

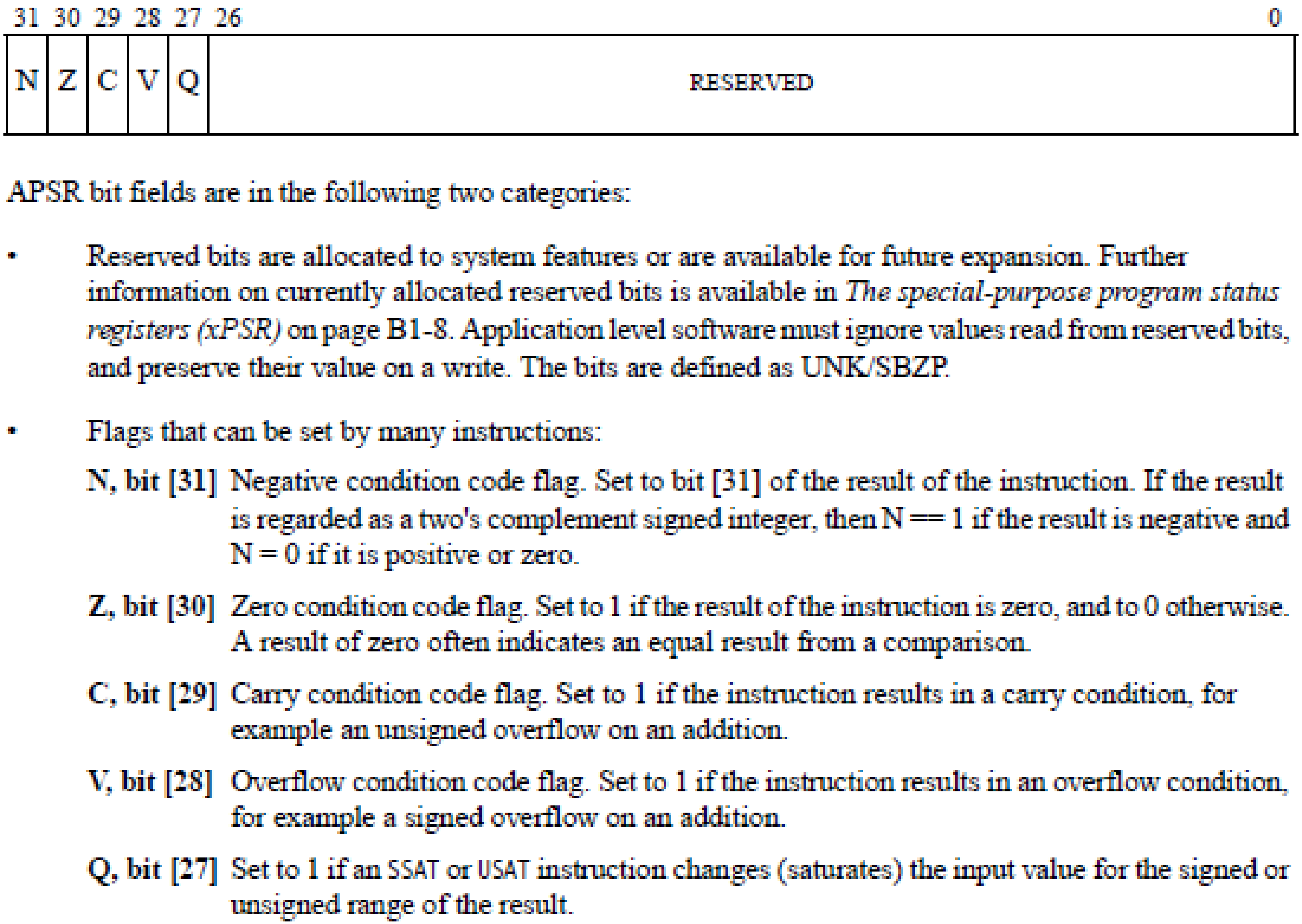
# State Register

- **Application Program Status Register (APSR)**
- **Interrupt PSR（IPSR）**
- **Execution PSR（EPSR）**



| | 31 | 30 | 29 | 28 | 27 | 26:25 | 24 | 23:20 | 19:16 | 15:10 | 9 | 8 | 7 | 6 | 5 | 4:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APSR | N | Z | C | V | Q | | | | | | | | | | | |
| IPSR | | | | | | | | | | | | Exception Number | | | | |
| EPSR | | | | | | ICI/IT | T | | | ICI/IT | | | | | | |

# Application Program Status Register (APSR)



| 31 | 30 | 29 | 28 | 27 | 26 … 0 |
|---|---|---|---|---|---|
| N | Z | C | V | Q | RESERVED |

APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- Flags that can be set by many instructions:

  **N, bit [31]** Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  **Z, bit [30]** Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  **C, bit [29]** Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  **V, bit [28]** Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  **Q, bit [27]** Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

# Updating the APSR

- **SUB Rx, Ry**
  - Rx = Rx - Ry
  - APSR unchanged
- **SUBS**
  - Rx = Rx - Ry
  - APSR N or Z bits might be set
- **ADD Rx, Ry**
  - Rx = Rx + Ry
  - APSR unchanged
- **ADDS**
  - Rx = Rx + Ry
  - APSR C or V bits might be set

# Least Significant Byte (LSB) and Most Significant Byte (MSB)

- **ARM support data types**
- **unsigned and signed char (8 bit)**
- **unsigned and signed (16 bit) -2 byte boundary**
  - Addressing: A、A＋1
- **unsigned and signed int (32 bit) -4 byte boundary**
  - Addressing: A、A＋1、A＋2 and A＋3

# Least Significant Byte (LSB) and Most Significant Byte (MSB)

<table>
<tr><th>31 24</th><th>23 16</th><th>15 8</th><th>7 0</th></tr>
<tr><td colspan="4">字单元A</td></tr>
<tr><td colspan="2">半字单元A</td><td colspan="2">半字单元A+2</td></tr>
<tr><td>字节单元A</td><td>字节单元A+1</td><td>字节单元A+2</td><td>字节单元A+3</td></tr>
</table>

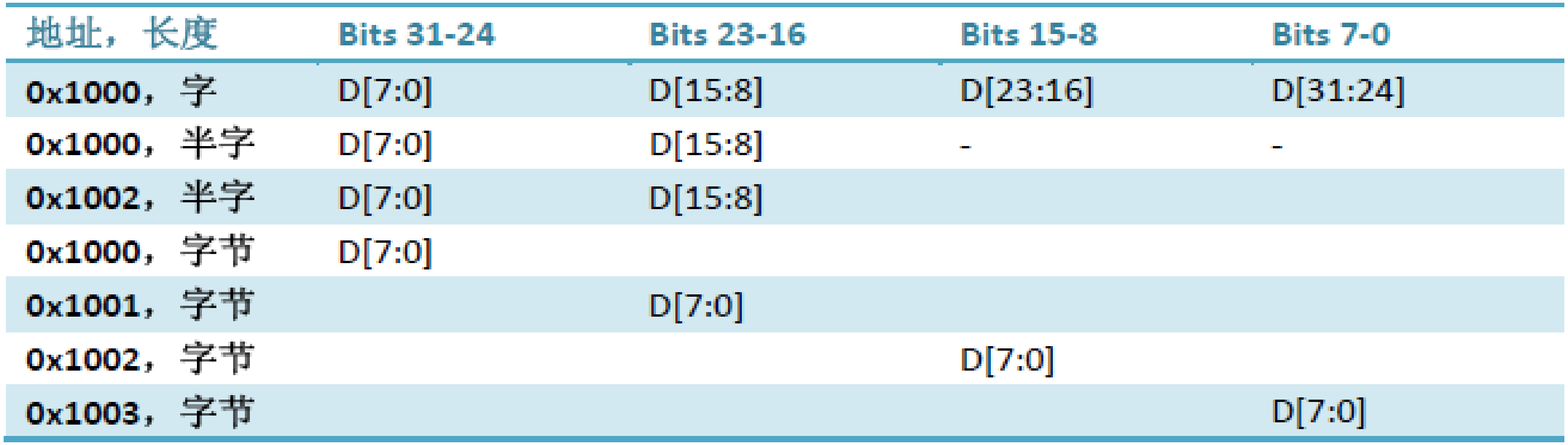
Most Significant Byte (MSB)

<table>
<tr><th>31 24</th><th>23 16</th><th>15 8</th><th>7 0</th></tr>
<tr><td colspan="4">字单元A</td></tr>
<tr><td colspan="2">半字单元A+2</td><td colspan="2">半字单元A</td></tr>
<tr><td>字节单元A+3</td><td>字节单元A+2</td><td>字节单元A+1</td><td>字节单元A</td></tr>
</table>

Least Significant Byte (LSB)

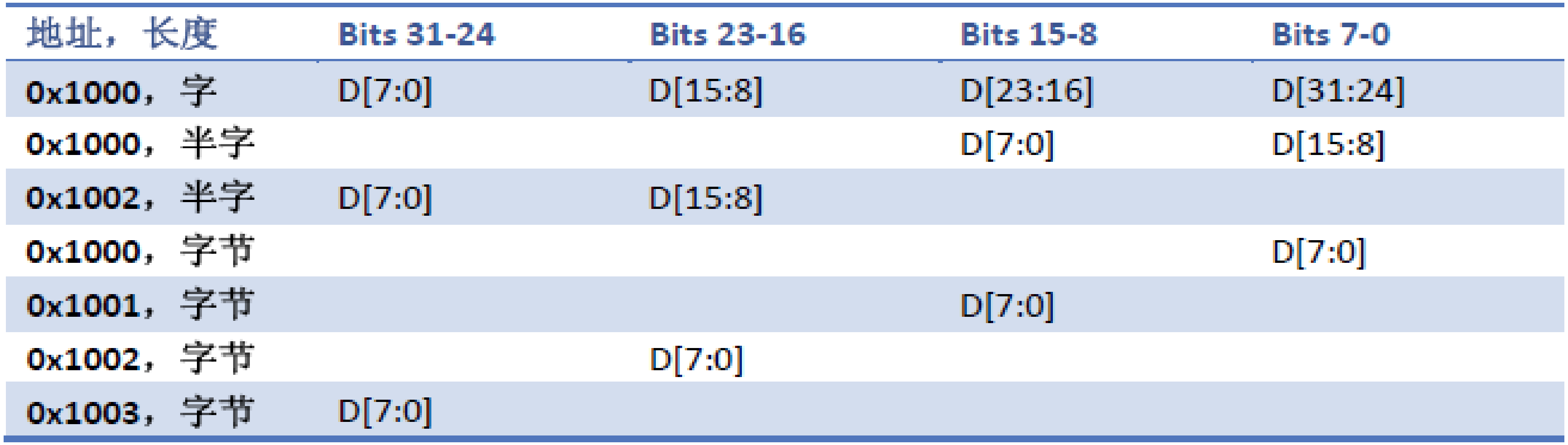
# Least Significant Byte (LSB) and Most Significant Byte (MSB)

表 5.4　CM3 的字节不变大端：存储器视图



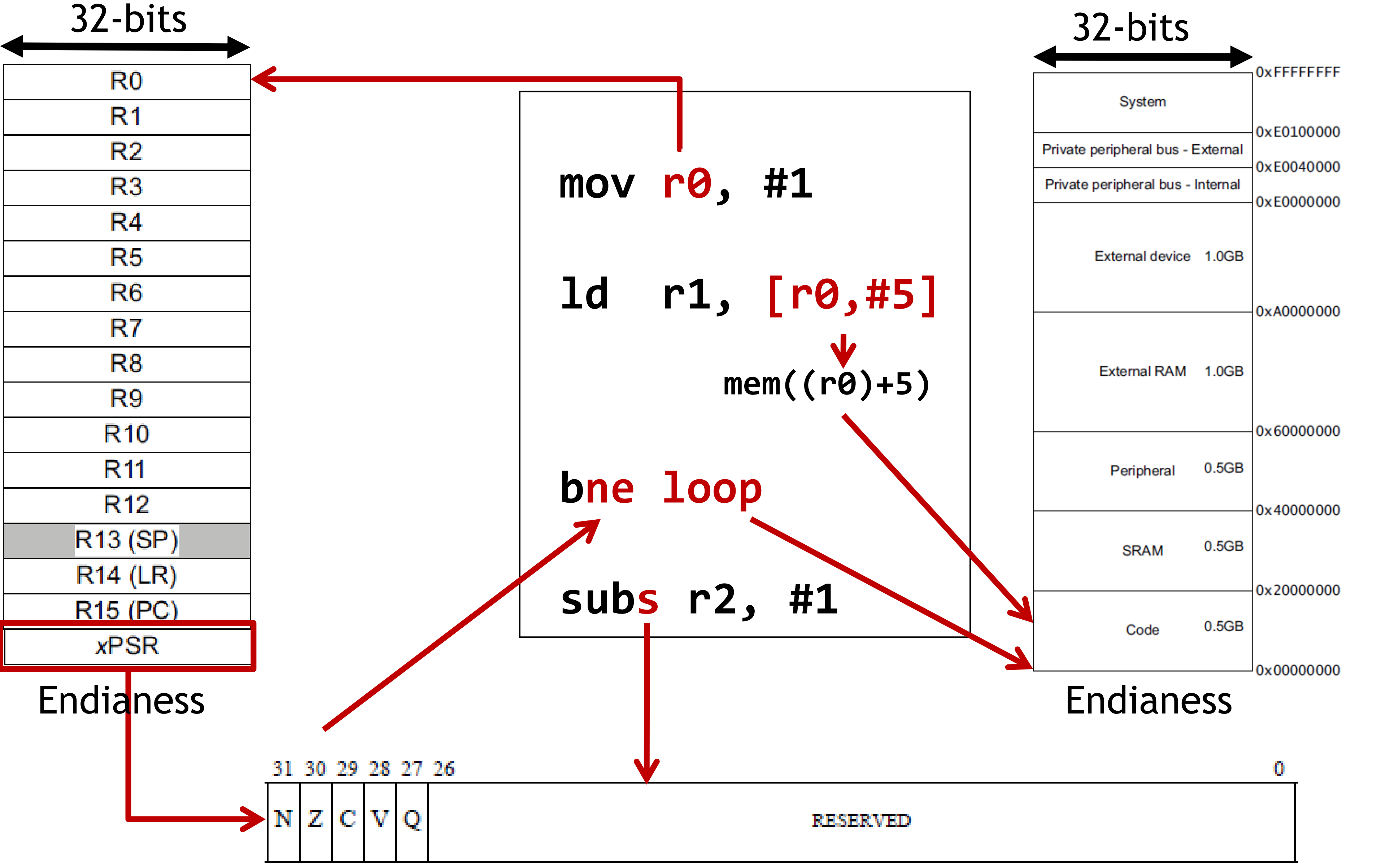
| 地址，长度 | Bits 31-24 | Bits 23-16 | Bits 15-8 | Bits 7-0 |
|---|---|---|---|---|
| 0x1000，字 | D[7:0] | D[15:8] | D[23:16] | D[31:24] |
| 0x1000，半字 | D[7:0] | D[15:8] | - | - |
| 0x1002，半字 | D[7:0] | D[15:8] | | |
| 0x1000，字节 | D[7:0] | | | |
| 0x1001，字节 | | D[7:0] | | |
| 0x1002，字节 | | | D[7:0] | |
| 0x1003，字节 | | | | D[7:0] |

表 5.5　CM3 的字节不变大端：在 AHB 上的数据



| 地址，长度 | Bits 31-24 | Bits 23-16 | Bits 15-8 | Bits 7-0 |
|---|---|---|---|---|
| 0x1000，字 | D[7:0] | D[15:8] | D[23:16] | D[31:24] |
| 0x1000，半字 | | | D[7:0] | D[15:8] |
| 0x1002，半字 | D[7:0] | D[15:8] | | |
| 0x1000，字节 | | | | D[7:0] |
| 0x1001，字节 | | | D[7:0] | |
| 0x1002，字节 | | D[7:0] | | |
| 0x1003，字节 | D[7:0] | | | |

# Major elements of an Instruction Set Architecture (registers, memory, word size, endianess, conditions, instructions, addressing modes)



32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
xPSR

Endianess

```
mov r0, #1

ld  r1, [r0,#5]

          mem((r0)+5)

bne loop

subs r2, #1
```

32-bits

| | |
|---|---|
| System | 0xFFFFFFFF |
| Private peripheral bus - External | 0xE0100000 |
| Private peripheral bus - Internal | 0xE0040000 |
| External device 1.0GB | 0xE0000000 |
| External RAM 1.0GB | 0xA0000000 |
| Peripheral 0.5GB | 0x60000000 |
| SRAM 0.5GB | 0x40000000 |
| Code 0.5GB | 0x20000000 |
| | 0x00000000 |

Endianess

| 31 | 30 | 29 | 28 | 27 | 26 … 0 |
|---|---|---|---|---|---|
| N | Z | C | V | Q | RESERVED |

# The instruction set

- **The full instruction set is shown to the right.**

- **How to learn instruction set?**
  - Break down into groups

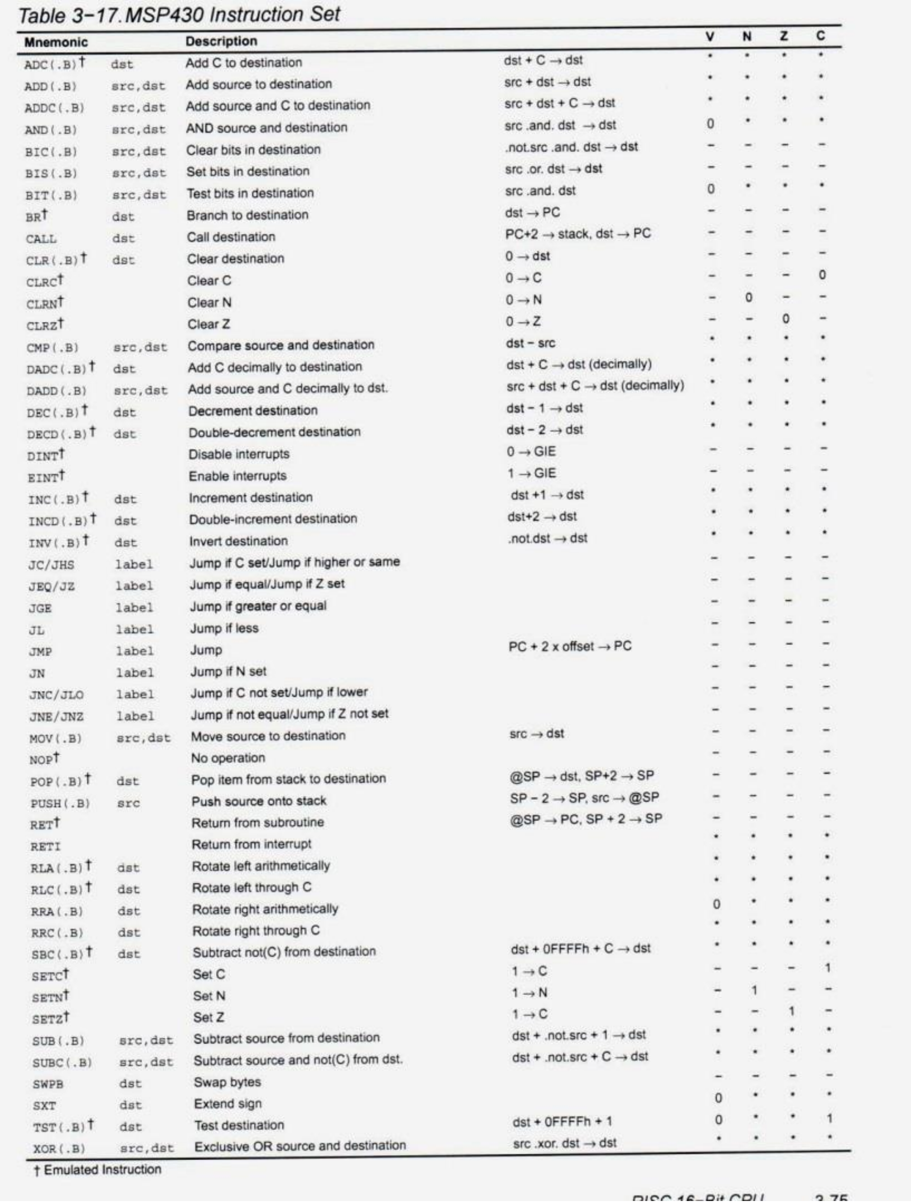


*Table 3−17. MSP430 Instruction Set*

| Mnemonic | | Description | | V | N | Z | C |
|---|---|---|---|---|---|---|---|
| ADC(.B)† | dst | Add C to destination | dst + C → dst | • | • | • | • |
| ADD(.B) | src,dst | Add source to destination | src + dst → dst | • | • | • | • |
| ADDC(.B) | src,dst | Add source and C to destination | src + dst + C → dst | • | • | • | • |
| AND(.B) | src,dst | AND source and destination | src .and. dst → dst | 0 | • | • | • |
| BIC(.B) | src,dst | Clear bits in destination | .not.src .and. dst → dst | – | – | – | – |
| BIS(.B) | src,dst | Set bits in destination | src .or. dst → dst | – | – | – | – |
| BIT(.B) | src,dst | Test bits in destination | src .and. dst | 0 | • | • | • |
| BR† | dst | Branch to destination | dst → PC | – | – | – | – |
| CALL | dst | Call destination | PC+2 → stack, dst → PC | – | – | – | – |
| CLR(.B)† | dst | Clear destination | 0 → dst | – | – | – | – |
| CLRC† | | Clear C | 0 → C | – | – | – | 0 |
| CLRN† | | Clear N | 0 → N | – | 0 | – | – |
| CLRZ† | | Clear Z | 0 → Z | – | – | 0 | – |
| CMP(.B) | src,dst | Compare source and destination | dst – src | • | • | • | • |
| DADC(.B)† | dst | Add C decimally to destination | dst + C → dst (decimally) | • | • | • | • |
| DADD(.B) | src,dst | Add source and C decimally to dst. | src + dst + C → dst (decimally) | • | • | • | • |
| DEC(.B)† | dst | Decrement destination | dst – 1 → dst | • | • | • | • |
| DECD(.B)† | dst | Double-decrement destination | dst – 2 → dst | • | • | • | • |
| DINT† | | Disable interrupts | 0 → GIE | – | – | – | – |
| EINT† | | Enable interrupts | 1 → GIE | – | – | – | – |
| INC(.B)† | dst | Increment destination | dst +1 → dst | • | • | • | • |
| INCD(.B)† | dst | Double-increment destination | dst+2 → dst | • | • | • | • |
| INV(.B)† | dst | Invert destination | .not.dst → dst | • | • | • | • |
| JC/JHS | label | Jump if C set/Jump if higher or same | | – | – | – | – |
| JEQ/JZ | label | Jump if equal/Jump if Z set | | – | – | – | – |
| JGE | label | Jump if greater or equal | | – | – | – | – |
| JL | label | Jump if less | | – | – | – | – |
| JMP | label | Jump | PC + 2 x offset → PC | – | – | – | – |
| JN | label | Jump if N set | | – | – | – | – |
| JNC/JLO | label | Jump if C not set/Jump if lower | | – | – | – | – |
| JNE/JNZ | label | Jump if not equal/Jump if Z not set | | – | – | – | – |
| MOV(.B) | src,dst | Move source to destination | src → dst | – | – | – | – |
| NOP† | | No operation | | – | – | – | – |
| POP(.B)† | dst | Pop item from stack to destination | @SP → dst, SP+2 → SP | – | – | – | – |
| PUSH(.B) | src | Push source onto stack | SP – 2 → SP, src → @SP | – | – | – | – |
| RET† | | Return from subroutine | @SP → PC, SP + 2 → SP | – | – | – | – |
| RETI | | Return from interrupt | | • | • | • | • |
| RLA(.B)† | dst | Rotate left arithmetically | | • | • | • | • |
| RLC(.B)† | dst | Rotate left through C | | • | • | • | • |
| RRA(.B) | dst | Rotate right arithmetically | | 0 | • | • | • |
| RRC(.B) | dst | Rotate right through C | | • | • | • | • |
| SBC(.B)† | dst | Subtract not(C) from destination | dst + 0FFFFh + C → dst | • | • | • | • |
| SETC† | | Set C | 1 → C | – | – | – | 1 |
| SETN† | | Set N | 1 → N | – | 1 | – | – |
| SETZ† | | Set Z | 1 → C | – | – | 1 | – |
| SUB(.B) | src,dst | Subtract source from destination | dst + .not.src + 1 → dst | • | • | • | • |
| SUBC(.B) | src,dst | Subtract source and not(C) from dst. | dst + .not.src + C → dst | • | • | • | • |
| SWPB | dst | Swap bytes | | – | – | – | – |
| SXT | dst | Extend sign | | 0 | • | • | • |
| TST(.B)† | dst | Test destination | dst + 0FFFFh + 1 | 0 | • | • | 1 |
| XOR(.B) | src,dst | Exclusive OR source and destination | src .xor. dst → dst | • | • | • | • |

† Emulated Instruction

RISC 16–Bit CPU 3-75

# Instruction classes

- Branching
- Data processing
- Load/store
- Exceptions
- Miscellaneous

# Addressing Modes

- **Offset Addressing**
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- **Pre-indexed Addressing**
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- **Post-indexed Addressing**
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

# <offset> options

- **An immediate constant**
  - #10

- **An index register**
  - <Rm>

- **A shifted index register**
  - <Rm>, LSL #<shift>

# Updating the Application Program Status Register (aka condition codes or APRS)

- **sub r0, r1**
  - r0 ← r0 – r1
  - APSR remain unchanged
- **subs r0, r1**
  - r0 ← r0 – r1
  - APSR N or Z bits could change

- **add r0, r1**
  - r0 ← r0 + r1
  - APSR remain unchanged
- **adds r0, r1**
  - r0 ← r0 + r1
  - APSR C or V bits could change

# The endianess religious war: 284 years and counting!

- **Modern version**
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war

- **Historical Inspiration**
  - Jonathan Swift
  - *Gullivers Travels*
  - Published in 1726
  - Satire on Henry-VIII's split with the Church

- Little-Endian
  - LSB is at lower address

```
                                Memory     Value
                                Offset   (LSB) (MSB)
                                ======   ===========
uint8_t a  = 1;                 0x0000   01 02 FF 00
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;        0x0004   78 56 34 12
```

- Big-Endian
  - MSB is at lower address

```
                                Memory     Value
                                Offset   (LSB) (MSB)
                                ======   ===========
uint8_t a  = 1;                 0x0000   01 02 00 FF
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;        0x0004   12 34 56 78
```

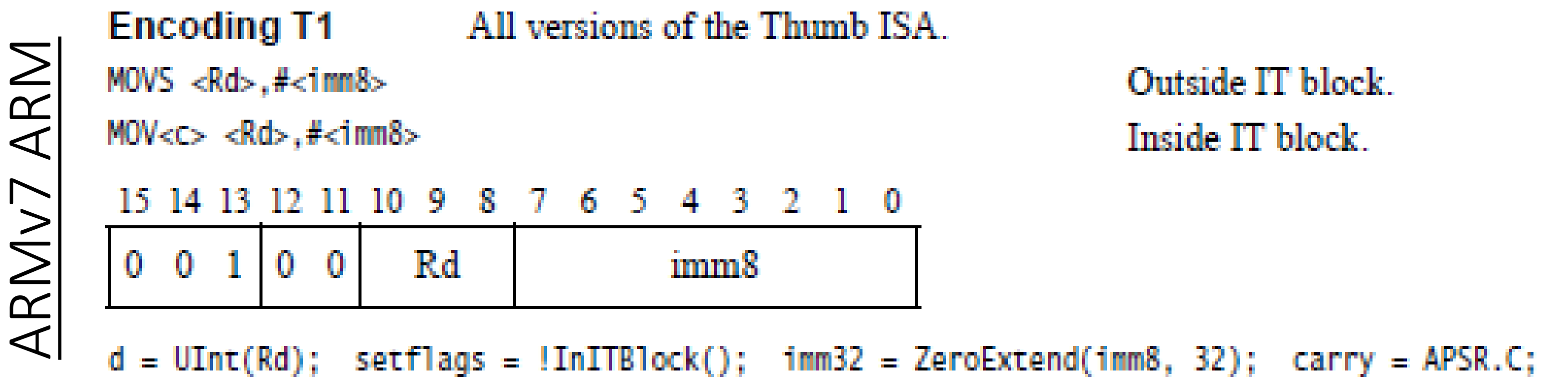
# Instruction encoding

- **Instructions are encoded in machine language opcodes**
- **Sometimes**
  - Necessary to hand generate opcodes
  - Necessary to verify assembled code is correct
- **How?**

```
Instructions
movs r0, #10

movs r1, #0
```

```
Register Value                 Memory Value
001|00|000|00001010            (LSB) (MSB)
(msb)               (lsb)      0a 20 00 21
001|00|001|00000000
```



ARMv7 ARM

Encoding T1 All versions of the Thumb ISA.

MOVS <Rd>,#<imm8> Outside IT block.

MOV<c> <Rd>,#<imm8> Inside IT block.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | Rd | | | imm8 | | | | | | | |

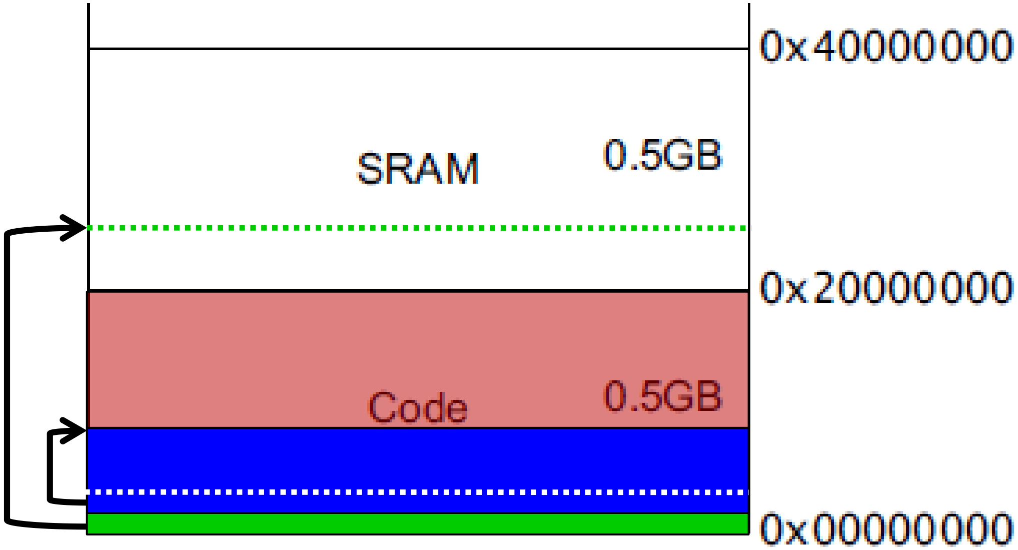
```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

# What happens after a power-on-reset (POR)?

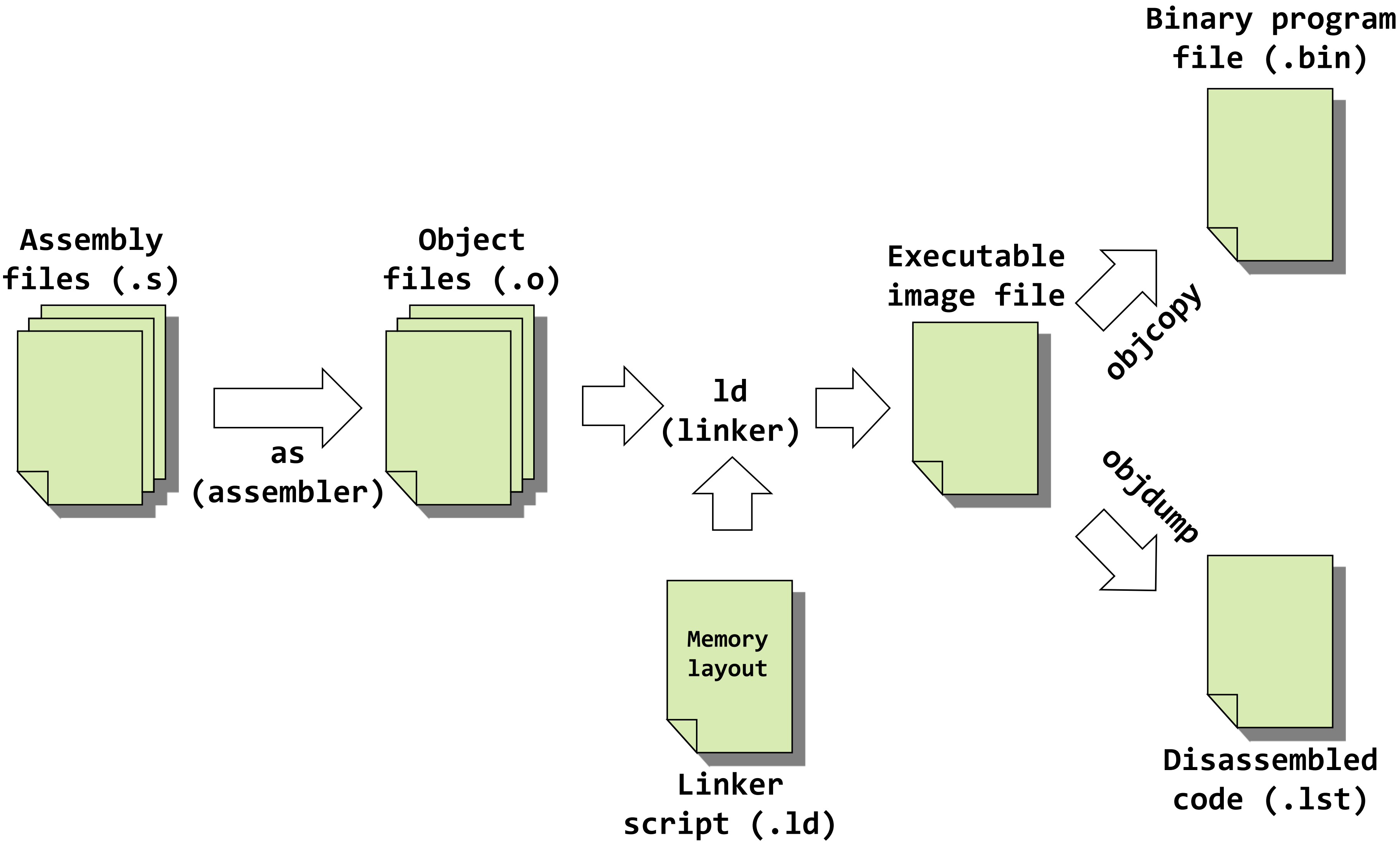
- **On the ARM Cortex-M3**
- **SP and PC are loaded from the code (.text) segment**
- **Initial stack pointer**
  - LOC: 0x00000000
  - POR: SP ← mem(0x00000000)
- **Interrupt vector table**
  - *Initial* base: 0x00000004
  - Vector table is relocatable
  - Entries: 32-bit values
  - Each entry is an address
  - Entry #1: reset vector
    - LOC: 0x0000004
    - POR: PC ← mem(0x00000004)
- **Execution begins**

```
          .equ      STACK_TOP, 0x20000800
          .text
          .syntax   unified
          .thumb
          .global   _start
          .type     start, %function

_start:
          .word     STACK_TOP, start
start:
          movs r0, #10
          ...
```



# How does an assembly language program get turned into a executable program image?



# Accessing memory locations from C

- Memory has an address and value
- Can equate a pointer to desired address
- Can set/get de-referenced value to change memory

```
#define  SYSREG_SOFT_RST_CR  0xE0042030

uint32_t *reg = (uint32_t *)(SYSREG_SOFT_RST_CR);

main () {
  *reg |= 0x00004000; // Reset GPIO hardware
  *reg &= ~(0x00004000);
}
```

# Some useful C keywords

- **const**
  - Makes variable value or pointer parameter unmodifiable
  - const foo = 32;
- **register**
  - Tells compiler to locate variables in a CPU register if possible
  - register int x;
- **static**
  - Preserve variable value after its scope ends
  - Does not go on the stack
  - static int x;
- **volatile**
  - Opposite of const
  - Can be changed in the background
  - volatile int I;