

Sistemas Encaixados

Tema 2: Arquitectura dos Sistemas Encaixados

Oscar García Lorenzo

Escola Politécnica Superior de Enxeñaría



Arquitectura dos Sistemas Encaixados

1 Linguaxe Ensamblador

- Arm
- Compoñentes dun microcontrolador
- C e Ensamblador
- Formato de instrucións
- Tipos de instruccións
- directivas

2 Arimética e lóxica

- Desprazamento e Rotación
- Aritméticas
- Lóxicas
- Outros

3 Load e Store

- Números Inmediatos
- Acceso a memoria

4 Saltos e condicional

5 Programación estructurada e subrutinas

Índice

1 Linguaxe Ensamblador

- Arm
- Compoñentes dun microcontrolador
- C e Ensamblador
- Formato de instrucións
- Tipos de instruccións
- directivas

2 Aritmética e lóxica

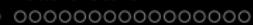
- Desprazamento e Rotación
- Aritméticas
- Lóxicas
- Outros

3 Load e Store

- Números Inmediatos
- Acceso a memoria

4 Saltos e condicional

5 Programación estructurada e subrutinas



Conxuntos de linguaxes ensamblador ARM

- **Thumb:**

- Instrucións de 16 bits
- Código denso
- Menos rexistros, operacións
- Baixo custo e baixo consumo

- **ARM32:**

- Instrucións de 32 bits
- Código menos denso
- Máis flexibilidade, código rápido
- Máis custo e máis consumo



Conxuntos de linguaxes ensamblador ARM

- **Thumb-2:**

- Thumb + algunas instrucións ARM32
- Código denso + rendemento

- **ARM64:**

- Instrucións de 64 bits
- Para propósito xeral (PCs, Móviles)

ARM Cortex

- **Cortex-A:** (Application)
 - Teléfonos intelixentes, tablets, PCs
 - Linux, Android, Windows
 - ARMv7-A, ARMv8-A
 - **Cortex-R:** (Real-time)
 - Fiabilidade, tolerancia a fallos, comportamento determinista
 - Onde non so fai falla facer cousa, pero facelas nun tempo determinado
 - **Cortex-M:** (Microcontroller)
 - Compromiso entre custo, rendemento e gasto enerxético
 - Soen vir acompañados da memoria e periféricos entrada/saída

ARM Cortex-M

- Cortex-M0, Cortex-M0+, Cortex-M1
 - ARMv6-M
 - Cortex-M3, Cortex-M4, Cortex-M7
 - ARMv7-M
 - Retrocompatibles
 - Código que funciona en M3 funciona en M4
 - FPU
 - Opcional en M4 e M7
 - Operações especiais para digital signal processing (DSP)
 - ARMv7-M
 - Outras arquitecturas teñen que cambiar modo Thumb-2 para executalo
 - Esta soporta Thumb-2 (e Thumb)
 - Pode mesturar instrucións de 16 e 32 bit

Linguaxe ensamblador

- Veremos a lingaxe Thumb-2
 - Veremos instrucións de 32 bits
 - Veremos pouco de instrucións para a FPU (que é opcional) de modo teórico
 - Veremos pouco de instrucións para DSP (que só está en algúns modelos) de modo teórico
- Nas prácticas usaremos un simulador de Thumb
 - Algunhas cousas de teoría non se poderán facer

Buses

Bus

Un conxunto de cables físicos para transmitir datos ou sinais de control

- Requieren un protocolo que indique como se realiza a comunicación
- O seu ancho de banda depende do seu ancho (en bits) e a frecuencia de reloxo
- Unha ponte conecta dous buses diferentes

Máis en detalle no tema 4

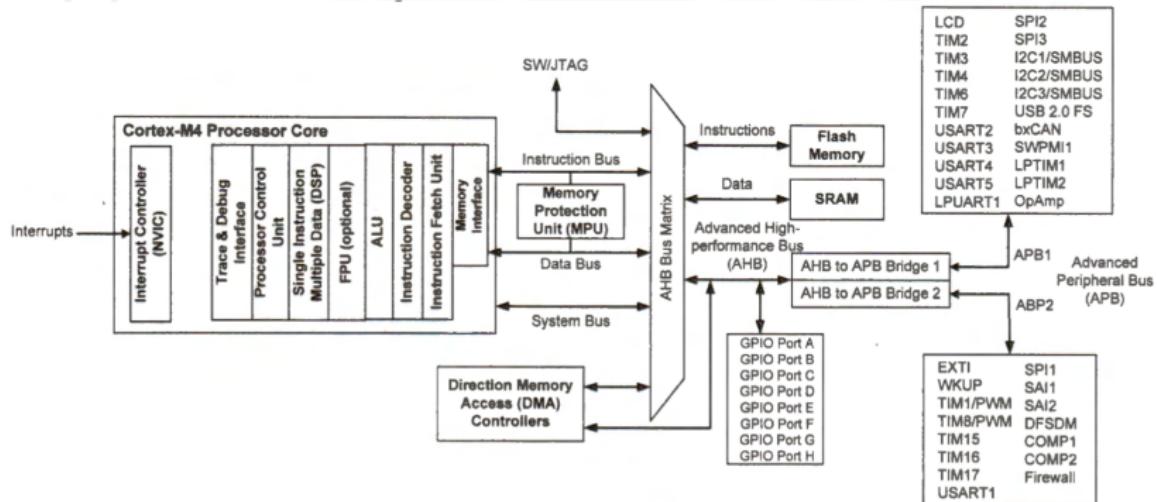
Organización

- Un núcleo baseado na arquitectura Cortex-M (poden ser outras)
- O núcleo comunicase coa memoria Flash, memoria RAM, GPIO, etc mediante unha matriz de buses (crossbar switch).
- A matriz de buses (crossbar switch) é un sistema de interconexión que permite múltiples comunicacóns á vez
- Os periféricos conéctanse a un bus ponte:
 - Advanced high-performance bus (AHB): para comunicacóns importantes
 - Advanced peripheral bus (APB): comunicacóns con periféricos
- General-purpose input/output (GPIO):
 - O seu comportamento cámbiase mediante software
 - Diferentes modelos segundo o fabricante

Compoñentes do núcleo

- Arithmetic logic unit (ALU)
 - Realiza operacións matemáticas e lóxicas con enteros
 - 2 entradas 1 saída
- Unidade de control
 - Xera sinais e coordina os compoñentes
- Controlador de interrupcións (NVIC)
 - Permite parar a execución que se leva a cabo para xestionar unha interrupción (xa o veremos no tema 3)
- Unidade de descodificación de instruccións
 - Permite obter e descodificar as instruccións
- Interface de memoria
- Interface de debug
- DSP e FPU dependen do procesador, e so no M7 son relativamente completas

Organización Cortex-M



C e Ensamblador

- C é unha linguaxe de alto nivel
 - Abstrae moitos conceptos
 - Máis fácil de programar
- Thumb está moi próximo á máquina
 - Control completo e detallado
 - Algunhas instrucións non teñen equivalente en C
 - Máis eficiente, (moito) más difícil de programar
- O mesmo programa en C pode compilarse para diferentes arquitecturas
- O ensamblador require coñecer detalles moi concretos sobre a arquitectura

Tipos de conxuntos de instrucións

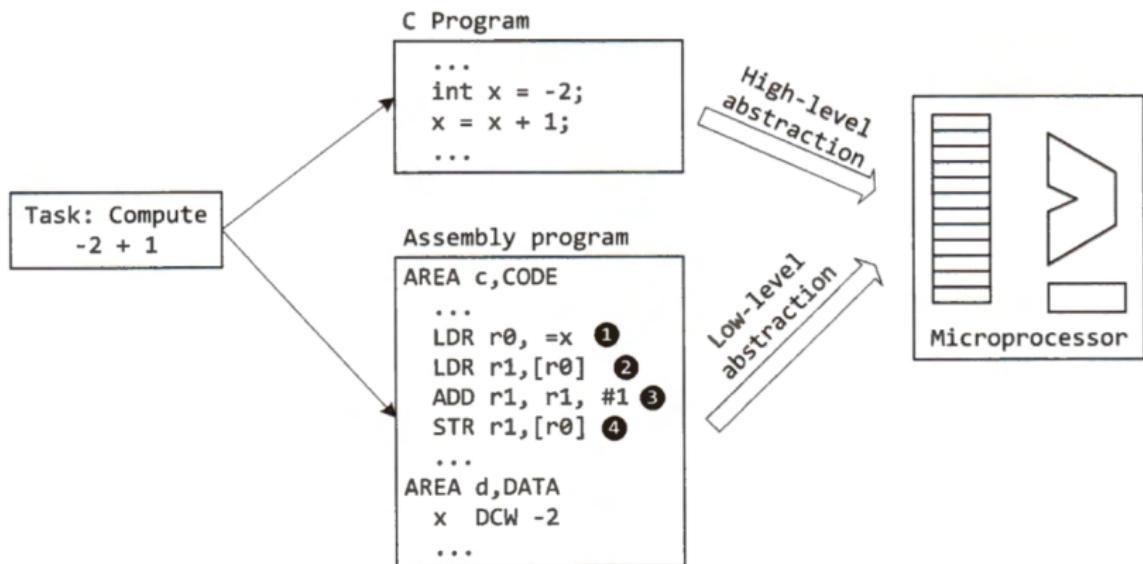
En xeral hai 3 tipos de arquitecturas para instrucións

- Baseadas en Acumulador
 - Un dos operandos da ALU e sempre o rexistro acumulador
 - O resultado sempre se garda no acumulador
 - 1950
- Baseadas na Pila (Stack)
 - Os dous operandos da ALU están arriba da pila
 - O resultado gardase enriba da pila
 - Push/pop
 - 1960
- Cargar/gardar (Load/Store)
 - Os dous operandos da ALU están en calquera rexistro
 - O resultado gardase en calquera rexistro
 - So se pode mover entre rexistros e memoria con load/store
 - Maioría actualmente

Load/Store Instruction Set

- Soporta dous operandos
 - Para algunas instruccións un deles pode ser un número constante, codificado na propia instrución
- Máis rápida que calquera das anteriores
- Os operandos hai que traelos desde memoria
 - *load-modify-store*

Programa en C e Ensamblador



Load/Store Instruction Set

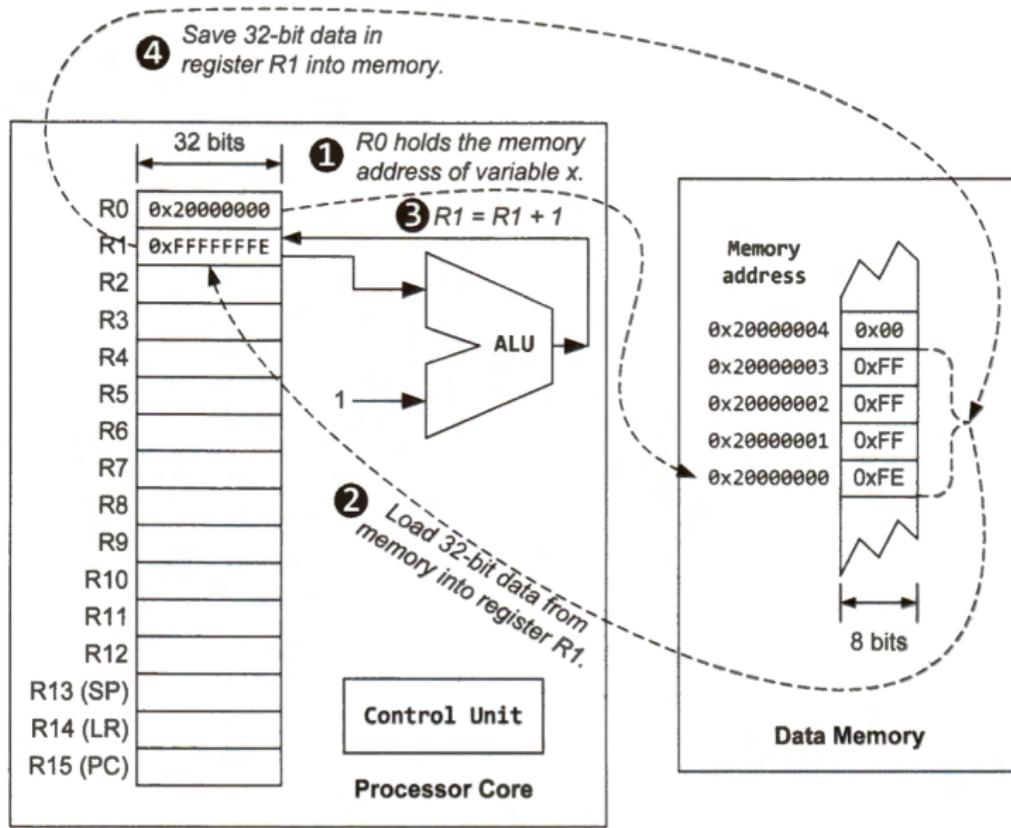
LDR r0, =x ; Paso 1: Prepara a dirección
(Carga a dirección de x en r0)

LDR r1, [r0] ; Paso 2: Load
(r0 ten a dirección de x)

ADD r1, r1, #1 ; Paso 3: Modify
(Incrementa o valor en r1 por 1)

STR r1, [r0] ; Paso 4: Store
(Garda o contido de r1 na memoria)

Load-Modify-Store



Instrucciones

Instrucción máquina

Consiste de:

- **opcode**: Código en binario da operación a realizar
- Cero ou más operandos

Instrucción ensamblador

- O **opcode** cámbiase por un mnemónico
- É diferente o linguaxe ensamblador (o que usamos para programar, Assembly) e o ensamblador (o que a transforma a código máquina, Assembler)

En xeral o formato é :

etiqueta mnemónico operando1, op2, op3 ;comentarios

Formato linguaxe ensamblador

- **etiqueta (label)**

- É unha referencia á dirección da instrución
- O ensamblador a cambia pola dirección real ou unha relativa

- **mnemónico**

- A operación a realizar

- **operандos**

- Poden ser de 0 a 3
- Sepáranse por comas
- Algunhas instrucións permiten números inmediatos
- Operando 1: rexistro destino
- Operando 2: operando fonte, xeralmente un rexistro
- Operando 3: operando fonte, rexistro, número inmediato, rexistro desprazado por uns bits, rexistro máis un offset

- **comentarios:** todo o que vai despois do punto e coma, para axudar a entender o código

Formato linguaxe ensamblador

O anterior era o formato ARM, en GNU é algo diferente. En practicas usaremos o GNU:

etiqueta : mnemónico operando1, op2, op3 /*comentarios*/

etiqueta : mnemónico operando1, op2, op3 @ comentarios

Exemplos

```
ADD r0, r2, r3 ; r0 =r2 + r3
SUB r3, r0, #3 ; r3 =re - 3
MOV r0, #'M'
ADD r1, r2, r3 ;r1=r2+r3
ADD r1, r3 ;r1=r1+r3 op1 pode omitirse se =op2
ADD r1, r2, #4 ;r1=r2+4
ADD r1 , #15 ,r1=r1+1S
```

no manual:

```
ADD {Rd, } Rn, Op2 ; Rd= Rn+ Op2
```

Barrel Shifter

```
ADD r0, r2, r1, LSL#2  
MOV r0, r2, ASR #2  
MOV r0, r0, ROR #16
```

O último operando (op2 ou op3) pode ter diferentes formatos,
aquí desprazamos (xa o veremos)

Formato programa ensamblador

- **etiqueta (label)**

- Sempre ao principio da liña, pode ser o nome dunha función

- **directiva**

- Axudan ao ensamblador a crea o código máquina
- PROC e ENDP indican principio e fin de funcións (subrutina)
- AREA indica unha rexión de datos ou código
- ...
- En GNU son diferentes e empezan por punto (.area,.word)

- **instrucións en ensamblador**

- Comandos que controlan o fluxo do programa ou manipulan datos
- Poden ser pseudo-instrucións, sen tradución directa ao linguaxe máquina

- **comentarios**

Programa Exemplo

Labels

```
AREA string_copy, CODE, READONLY
EXPORT _main
ALIGN
ENTRY
PROC
```

Directives

Program
Comments

Code Area

strcpy	LDR r1, =srcStr	; Retrieve address of the source string
	LDR r0, =dstStr	; Retrieve address of the destination string
loop	LDRB r2, [r1], #1	; Load a byte & increase src address pointer
	STRB r2, [r0], #1	; Store a byte & increase dst address pointer
	CMP r2, #0	; Check for the null terminator
stop	B loop	; Copy the next byte if string is not ended
	B stop	; Dead loop. Embedded program never exits.

Assembly Instructions

ENDP

Directives

```
AREA myData, DATA, READWRITE
ALIGN
```

Data Area

srcStr	DCB "The source string.",0	; Strings are null terminated
dstStr	DCB "The destination string.",0	; dststr has more space than srcstr

END

Data

Program
Comments



Programa Exemplo

- A AREA de datos define dúas cadeas de caracteres srcStr e dstStr
- A AREA de código inclúe a función _main que copia srcStr a dstStr



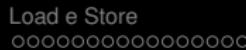
Tipos de instruccions

A maioría de instruccions dos ARM Cortex-M poden clasificarse como:

- Aritméticas, desprazamento ou lóxicas
- Movemento de datos
- Comparación e salto
- Miscelánea (debug ou cousas raras)

Os Cortex-M4 e M7 teñen ademais:

- Procesamento de sinais
- Punto flotante



Tipos de instrucciones

Shift, logic, and bit instructions	<p>Shift: LSL (logic shift left), LSR (logic shift right), ASR (arithmetic shift right), ROR (rotate right), RRX (rotate right with extend)</p> <p>Logic: AND (bitwise and), ORR (bitwise or), EOR (bitwise exclusive or), ORN (bitwise or not), MVN (move not)</p> <p>Bit set/clear: BFC (bit field clear), BFI (bit field insert), BIC (bit clear), CLZ (count leading zeros)</p> <p>Bit/byte reordering: RBIT (reverse bit order in a word), REV (reverse byte order in a word), REV16 (reverse byte order in each halfword independently), REVSH (reverse byte order in the bottom halfword, and sign extend to 32 bits)</p>
------------------------------------	--

Tipos de instruccións

Arithmetic instructions	Addition: ADD, ADC (add with carry)
	Subtraction: SUB, RSB (reverse subtract), SBC (subtract with carry)
	Multiplication: MUL (multiply), MLA (multiply with accumulate), MLS (multiply with subtract), SMULL (signed long multiply), UMULL (unsigned long multiply), SMLAL (signed long multiply, with accumulate), UMLAL (unsigned long multiply, with subtract)
	Division: SDIV (signed), UDIV (unsigned) Saturation: SSAT (signed), USAT (unsigned) Extension: SXTB (sign-extend a byte), SXTH (sign-extend a halfword), UXTB (zero-extend a byte), UXTH (zero-extend a halfword) Bit field extract: SBFX (signed extraction), UBFX (unsigned extraction)

Tipos de instrucciones

Memory access instructions	<p><i>Read data memory:</i></p> <p>LDRB (load byte), LDRH (load halfword), LDR (load word), LDRD (load double-word), LDRSB (load signed byte), LDRSH (load signed halfword), LDM, LDMDB, LDMFD (load multiple words) LDREXB, LDREXH, LDREX (load register exclusive with a byte, halfword, and word), LDRT (load in privileged modes), POP (load from stack)</p> <p><i>Write data memory:</i></p> <p>STRB (store byte), STRH (store halfword), STR (store word), STRD (store double-word), STRSB (store signed byte), STRSH (store signed halfword), STM, STMDB, STMF (store multiple words), STREXB, STREXH, STREX (store register exclusive with a byte, halfword, and word), STRT (store in privileged modes), PUSH (store into stack)</p>
Data copy instructions	<p>MOV (move), MOVT (move top), MOVW (move halfword), MRS (move from coprocessor), MSR (move to coprocessor)</p>

Tipos de instruccións

Data compare instructions	CMP (compare), CMN (compare negative), TST (test), TEQ (test equivalent), IT (if-then)
Branch instructions	B (branch), CBZ (compare and branch on zero), CBNZ (compare and branch on non-zero), TBB (table branch byte), TBH (table branch halfword)
Subroutine instructions	BL (branch with link), BLX (branch with link and exchange), BX (branch and exchange)
Miscellaneous instructions	BKPT (breakpoint), NOP (no operation), SEV (set event), WFE (wait for event), WFI (wait for interrupt), CPSID (interrupt disable), CPSIE (interrupt enable), DMB (data memory barrier), DSB (data synchronization barrier), ISB (instruction synchronization barrier)

Directivas

AREA, .section en GNU

Indica ao ensamblador o inicio dun área de códigos ou datos

- É a unidade básica e indivisible
- Deben de ter un nome e non poden ter o mesmo
- En GNU existen xa definidas .data, .text, .bss

ENTRY, etiqueta .main en GNU

Indica ao ensamblador a primeira instrución a executar

- Para aplicacóns en C o punto de entrada é a libraría estándar de C, non se soe ver

END, usar etiqueta .end en GNU

Indica o final do ficheiro de código

- É obligatoria en armasm

Directivas

PROC ENDP

Marcan o inicio e final dunha función (subrutina) (procedemento)

- Non se poden aniñar, non pode haber unha subrutina dentro de outra subrutina
- Cada programa require unha chamada `_main`
- En GNU úsanse etiquetas

DCB, DCW, DCD, DCQ, SPACE, FILL

Reservan a memoria para as variables e inicializan o seu contido En GNU úsase `.word` para palabras (DCD), `.hword` para medias palabras (DCW), `.byte` para bytes (DCB), `.space` para libre

Directivas

Directive	Description	Memory Space
DCB	Define Constant Byte	Reserve 8-bit values
DCW	Define Constant Half-word	Reserve 16-bit values
DCD	Define Constant Word	Reserve 32-bit values
DCQ	Define Constant Doubleword	Reserve 64-bit values
SPACE	Defined Zeroed Bytes	Reserve some zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

```

AREA myData, DATA, READWRITE
hello DCB "Hello World!",0 ; Allocate a string that is null-terminated
dollar DCB 2,10,0,200 ; Allocate integers ranging from -128 to 255
scores DCD 2,3.5,-0.8,4.0 ; Allocate 4 words containing decimal values
miles DCW 100,200,50,0 ; Allocate integers between -32768 and 65535
p SPACE 255 ; Allocate 255 bytes of zeroed memory space
f FILL 20,0xFF,1 ; Allocate 20 bytes and set each byte to 0xFF
binary DCB 2_01010101 ; Allocate a byte in binary
octal DCB 8_73 ; Allocate a byte in octal
char DCB 'A' ; Allocate a byte initialized to ASCII of 'A'

```

Directivas

EQU, RN, .equ, .req

Fan o programa máis fácil de entender

- EQU asocia un nome simbólico a unha constante numérica (similar a `#define` en C)
- En GNU: `.equ <symbol name>, <value>`
- RN asocia un nome simbólico a un rexistro
- En GNU usa nomes para os rexistros `<register_name>`
`.req <register_name>`

```
; Interrupt Number Definition (IRQn)
BusFault_IRQn    EQU    -11      ; Cortex-M Bus Fault Interrupt
SVCall_IRQn      EQU    -5       ; Cortex-M Supervisor Call (SVC) Interrupt
PendSV_IRQn      EQU    -2       ; Cortex-M Pend SVC Interrupt
SysTick_IRQn      EQU    -1       ; Cortex-M System Tick Interrupt
```

Dividend	RN	6	<i>; Defines dividend for register 6</i>
Divisor	RN	5	<i>; Defines divisor for register 5</i>

Directivas

ALIGN, .balign

Permite aliñar os datos e instrucións á memoria

- Accedese á memoria en bytes
- Pero os datos poden ser de palabra ou dobles palabras, etc
- Podes indicar a primeira dirección, se divisible por 8, por 4 ou 2
- Os datos desaliñados provocan unha perda de rendemento
- En algúns procesadores poder dar fallo directamente
- En GNU: `.balign <power_of_2>, <fill_value>, <max_padding>`

Directivas

EXPORT, IMPORT, .global

Permite localizar símbolos en diferentes ficheiros fonte

- EXPORT fai un símbolo visible
- IMPORT indica que un símbolo está noutro ficheiro
- En GNU se exporta xa o busca o linker

INCLUDE, GET, .include

Permite incluír outros ficheiros fonte ao ficheiro fonte actual

- Útil para ter un ficheiro de definición de constantes con EQU e incluílo en diferentes códigos

Índice

1 Linguaxe Ensamblador

- Arm
- Compoñentes dun microcontrolador
- C e Ensamblador
- Formato de instrucións
- Tipos de instruccións
- directivas

2 Arimética e lóxica

- Desprazamento e Rotación
- Aritméticas
- Lóxicas
- Outros

3 Load e Store

- Números Inmediatos
- Acceso a memoria

4 Saltos e condicional

5 Programación estructurada e subrutinas



Bandeiras/Flags

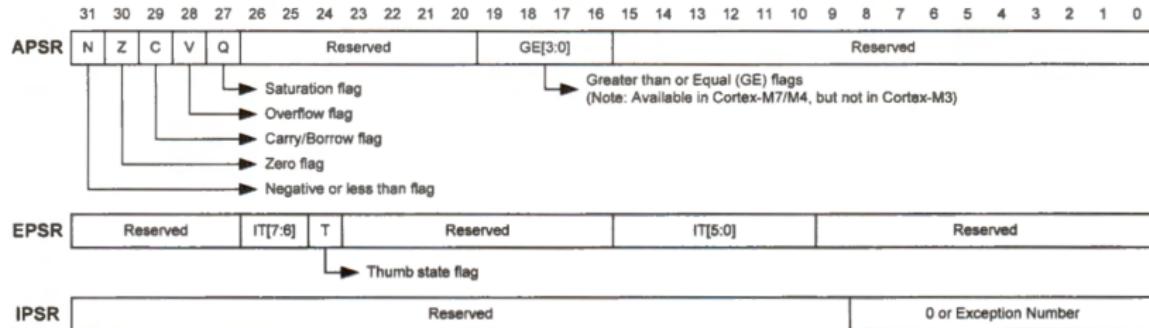
- **N**: 1 se resultado da ALU negativo (bit 32 a 1), senón a cero
- **Z**: 1 se resultado da ALU cero, senón a cero
- **C**: 1 se hai un acarreo ao final dunha suma da ALU cero, a cero se hai un préstamo nunha resta
- **V**: 1 se hai desborde unha suma ou resta con signo
- **Q**: 1 se hai saturación tras unha instrución SSAT ou USAT

Rexistro de estado do programa

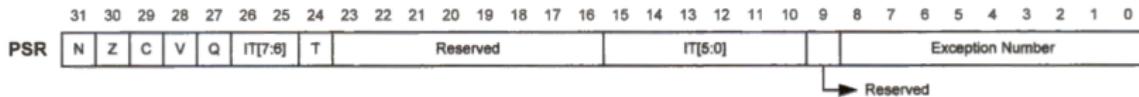
SPR

- **APSR:** Application:
 - Flags
- **EPSR:** Execution
 - Modo Thumb/ARM32 (a 1 sempre en Cortex-M)
 - **IT:** Indica cousas sobre bloques IF-THEN
- **IPSR:** Interrupt
 - Está en modo fío
 - O número de interrupción ou excepción
- So se poden acceder con dúas operacións especiais:
 - MSR: Move do rexistro especial a un xeral
 - MRS R0, APSR
 - MRS: Move do xeral ao especial
 - MSR APSR_NZCVQ, R0

Program Status Register



$$\text{PSR} = \text{APSR} + \text{EPSR} + \text{IPSR}$$





Bandeiras/Flags

- As instrucións acabadas en S modifican as bandeiras de estado.
 - ADDS, MOVS, SUBS ...
- As instrucións sen S non modifican as bandeiras de estado.
 - ADD, MOV, SUB ...
 - No simulador QtARMSim todas se cambian á versión S
- As instrucións de comparación tamén cambian as bandeiras de estado.
 - CMP, TEQ ...

Desprazamento e Rotación

- **LSL Rd, Rn, OP2:** Logical Shift Left
 - Move os bits n direccións á esquerda, reenche con ceros
 - Equivale a multiplicar por 2^n (« en C)
- **LSR Rd, Rn, OP2:** Logical Shift Right
 - Move os bits n direccións á dereita, rechea con ceros
 - Equivale a dividir por 2^n (» en C para sen signo)
- **ASR Rd, Rn, OP2:** Arithmetic Shift Right
 - Move os bits n direccións á dereita e copia o bit máis á esquerda (o signo)
 - Equivale a dividir por 2^n (» en C para signo)
- **ROR Rd, Rn, OP2:** Rotate Right
 - Move os bits n direccións á dereita e copia os desbordados pola esquerda
 - O bit desbordado pola dereita copia á bandeira **C**
- **RRX Rd, Rn, OP2:** Rotate Right with extend
 - Move os bits n direccións á dereita e copia os desbordados pola esquerda, o bit de **C** entra na rotación

Desprazamento e Rotación

- Son operacións moi comúns e útiles, ao principio pouco intuitivas
- C non ten ROR ou RRX depende do compilador
- As operacións cara á esquerda pódense facer con operación á dereita con $32 - n$
- O rexistro destino é opcional, pode ser igual á RN
- En Thumb, con so 16 bits por instrución, pode ser obligatorio RD=RN
- O **Op2** pode ser un rexistro ou un valor inmediato, pero so o **Op2**

Operacións Aritméticas, Sumas e Restas

- **ADD Rd, Rn, OP2:** Sumar
 - $Rd = Rn + Op2$
- **ADC Rd, Rn, OP2:** Sumar + Acarreo
 - $Rd = Rn + Op2 + C$
- **SUB Rd, Rn, OP2:** Restar
 - $Rd = Rn - Op2$
- **SBC Rd, Rn, OP2:** Restar + Acarreo -1
 - $Rd = Rn - Op2 + C - 1$
- **RSB Rd, Rn, OP2:** Restar ao revés
 - $Rd = Op2 - Rd$

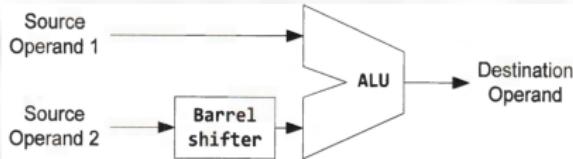


Operacións Aritméticas, Sumas e Restas

- O rexistro destino é opcional, pode ser igual á RN
- En Thumb, con so 16 bits por instrución, pode ser obligatorio RD=RN
- O **Op2** pode ser un rexistro ou un valor inmediato, pero so o **Op2**
 - SUB R1, #5
- As operacións con acarreo serven para operar con números de más de 32 bits, operando de 32 en 32 bits e acumulando o acarreo

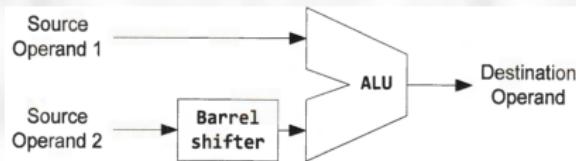
Aritméticas con desprazamento

- Algúns procesadores Cortex-M teñen o **Barrel Shifter** nunha das entradas da ALU, o que permite facer desprazamentos á vez que sumas/restas:
 - ADD R0, R2, R1, LSL #1
 $r0 = r2 + r1 << 1 = r2 + 2 \times r1$
 - ADD R1, R0, R0, LSR #3
 $r1 = r0 + r0 >> 3 = r0 + r0/8$
- O QtArmSim non o ten



Aritméticas con desprazamento

- O Barrel Shifter pode desprazar n bits por ciclo de reloxo
- Moi rápido
- Se se coloca ao principio da ALU pódese aforran en operacións custosas
- ADD R1, R0, R0, LSL #3
 - MOV R2, #9 ;r2 = 9
 - MUL R1,R0,R2 ;r1=r0*9
- É un exemplo de como o código ensamblador pode ser más eficiente





Operacións Aritméticas, Multiplicación

- **MUL Rd, Rn, Rm:** Multiplicar
 - $Rd = Rn \times Rm[31 : 0]$
 - UMUL Rd, RN, RM é igual
- **MLA Rd, Rn, Rm, Ra:** Multiplicar con acumulación
 - $Rd = Ra + (Rn \times Rm)[31 : 0]$
- **MLS Rd, Rn, Rm, Ra:** Multiplicar con substracción
 - $Rd = Ra - (Rn \times Rm)[31 : 0]$
- So gardan os 32 bits menos significativos, truncan



Operacións Aritméticas, División

- SDIV Rd, Rn, RM: Dividir con signo
 - $Rd = Rn/Rm$
- UDIV Rd, Rn, RM: Dividir sen signo
 - $Rd = Rn/Rm$
- Moitos procesadores non teñen isto
- Incluso se a linguaxe ensamblador as acepta convértense nunha chamada a procedemento

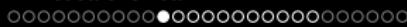


Operacións Aritméticas, Multiplicación Longa

- UMULL RdLo,RdHi,Rn,Rm
- SMULL RdLo,RdHi,Rn,Rm
- UMLAL RdLo,RdHi,Rn,Rm
- SMLAL RdLo,RdHi,Rn,Rm
- Gardan o resultado en dous rexistros
- U sen signo, S con signo
- LAL con acumulación

Operacións Aritméticas, Saturación

- SSAT, USAT
- Limitan o resultado a un rango de valores
- Se é maior ou menor ponse ao valor máximo ou mínimo do rango
- U sen signo, S con signo
- O operando sempre é inmediato, é a n de 2^n
- SSAT R2, #11, R1
 - $-2^{10} \leq r1 \leq 2^{10}$
- USAT R2, #11, R1
 - $0 \leq r1 \leq 2^{11}$



Operacións Lóxicas Bit a Bit

a	b	a & b	a b	a \oplus b	\neg a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Operacións Lóxicas Bit a Bit

- **AND RD, RN, OP2:** AND lóxico
 - $Rd = Rn \& Op2$
- **ORR RD, RN, OP2:** OR lóxico
 - $Rd = Rn | Op2$
- **EOR RD, RN, OP2:** OR exclusivo lóxico
 - $Rd = Rn \oplus Op2$
- **ORN RD, RN, OP2:** NOT OR lóxico
 - $Rd = Rn | (\neg Op2)$
- **BIC RD, RN, OP2:** NOT AND lóxico (limpiar bit)
 - $Rd = Rn \& (\neg Op2)$
- **MVN RD, OP2:** Negar todos os bits
 - $Rd = 0xFFFFFFFF \oplus Op2$
- **BFC RD, #LSB, #WIDTH:** limpar bits de lsb a width
- **BFI RD, RN, #LSB, #WIDTH:** meter bits de lsb a width desde RN

Operacións Lóxicas Bit a Bit

Máscara/Bitmask

- Para un número N a máscara ten o mesmo número de bits
- O bit $mascara(i)$ da máscara está a 1 se hai que operar co bit $N(i)$
- Se $N(i)$ é 1 dicimos que está enmascarado/masked
- A máscara divide os bits do número N en dous, unha parte que se selecciona e outra que se ignora
- Os bits que están a 1 na máscara comprobanse en N
- Operación moi útil e sinxela
- Esencial para aforrar memoria, acelerar, básica en SE

Operacións Lóxicas Bit a Bit

- $N = 0xA2 = 0b10100010$
- $Mask = 0x34 = 0b00110100$

Bitwise Operators	Symbol	Example
AND	&	$C = N \& Mask; // C = 0b00100000 = 0x20$
OR		$C = N Mask; // C = 0b10110110 = 0xB6$
EXCLUSIVE-OR (EOR)	^	$C = N ^ Mask; // C = 0b10010111 = 0x97$
NOT	~	$C = \sim N; // C = 0b01011101 = 0x5D$
SHIFT RIGHT	>>	$C = N >> 2; // C = 0b00101000 = 0x28$
SHIFT LEFT	<<	$C = N << 2; // C = 0b10001000 = 0x88$

Operacións Lóxicas Bit a Bit

Comprobar se un bit a 1

- Usar &
- Poñemos a 1 os bits a comprobar na máscara
- O resultado non pode ser 0

C Program	Assembly Program 1	Assembly Program 2
<pre>char a = 0x34; char mask = 1<<5; char b; // Check bit 5 b = a & mask;</pre>	<pre>LDR r0,#0x34 ; r0 = a LDR r1,#(1<<5) ; r1 = mask ANDS r2,r0,r1 ; r2 = b</pre>	<pre>LDR r0,#0x34 ; r0 = a ANDS r2,r0,#(1<<5)</pre>

Operacións Lóxicas Bit a Bit

Poñer un bit a 1 (is set)

- Usar |
- Os bits que deixamos a 0 na máscara non cambian
- Os bit que poñemos a 1 na máscara acabarán sempre a 1

C Program	Assembly Program 1	Assembly Program 2
char a = 0x34; char mask = 1<<5; // Set bit 5 a = mask;	LDR r0,#0x34 ; r0 = a LDR r1,#(1<<5) ; r1 = mask ORR r0,r0,r1	LDR r0,#0x34 ; r0 = a ORR r0,r0,#(1<<5)

Operacións Lóxicas Bit a Bit

Limpar un bit (clear)

- Usar &
- Os bits que deixamos a 0 na máscara pasa a 0
- Os bits que poñemos a 1 na máscara quedan igual
 - Todo a 1 menos os que queremos limpar (poner a 0)

C Program	Assembly Program 1	Assembly Program 2
<pre>char a = 0x34; char mask = 1<<5; // Reset bit 5 a &= ~mask;</pre>	<pre>LDR r0,#0x34 ; r0 = a LDR r1,#(1<<5) ; r1 = mask MVN r1,r1 ; NOT EOR r0,r0,r1</pre>	<pre>LDR r0,#0x34 ; r0 = a BIC r0,#(1<<5)</pre>

Operacións Lóxicas Bit a Bit

Cambiar un bit (toggle)

- Usar \oplus
- Os bits que deixamos a 0 na máscara quedan igual
- Os bits que poñemos a 1 na máscara cambian, pasan a negado (0 a 1, 1 a 0)
 - Todo a 0 menos os que queremos cambiar

C Program	Assembly Program 1	Assembly Program 2
<pre>char a = 0x34; char mask = 1<<5; // Toggle bit 5 a ^= mask;</pre>	<pre>LDR r0,#0x34 ; r0 = a LDR r1,#(1<<5) ; r1 = mask EOR r0,r0,r1</pre>	<pre>LDR r0,#0x34 ; r0 = a EOR r0,r0,#(1<<5)</pre>



Operacións Lóxicas Bit a Bit

- Non confundir coas operacións booleanas en C
 - $A \&\& B$, $A || B$, $!B$
- As booleanas operan en palabras
- Son as que usamos para as comparacións
- $0x01 \& 0x10 = 0x00$
- $0x01 \&\& 0x10 = 0x01$ (Verdadeiro)
- $\sim 0x01 = 0xFFFFFFFF$
- $!0x01 = 0x00$ (Falso)

Operacións Lóxicas Bit a Bit

O normal e usar EQU como directiva para definir as máscaras que imos usar

```
RCC_AHB2ENR_GPIOAEN EQU (0x00000001) ; GPIO port A clock enable
RCC_AHB2ENR_GPIOBEN EQU (0x00000002) ; GPIO port B clock enable
RCC_AHB2ENR_GPIOCEN EQU (0x00000004) ; GPIO port C clock enable

LDR r7, =RCC_BASE           ; Address of reset and clock control (RCC)
LDR r1, [r7, #RCC_AHB2ENR]   ; Load AHB2ENR from memory into r1
ORR r1, r1, #RCC_AHB2ENR_GPIOAEN ; Enable clock of GPIO port A
ORR r1, r1, #RCC_AHB2ENR_GPIOBEN ; Enable clock of GPIO port B
ORR r1, r1, #RCC_AHB2ENR_GPIOCEN ; Enable clock of GPIO port C
STR r1, [r7, #RCC_AHB2ENR]   ; Save to RCC->AHB2ENR
```



Operacións Lóxicas Bit a Bit

As operacións lóxicas modifican **N**, **Z** e **C** (se usan o barrel-shifter). **V** non se modifica

Cambiar a orde

- **RBIT Rd, Rn:** Cambiar a orde dos bits
 - $(i = 0; i < 32; i++) \quad Rd[i] = Rn[31 - i]$
- **REV Rd, Rn:** Cambiar a orde dos bytes
 - $Rd[31:24] = Rn[7:0], Rd[23:16] = Rn[15:8]$
 - $Rd[15:8] = Rn[23:16], Rd[7:0] = Rn[31:24]$
- **REV16 Rd, Rn:** Cambiar a orde dos bytes en cada media palabra
 - $Rd[15:8] = Rn[7:0], Rd[7:0] = Rn[15:8]$
 - $Rd[31:24] = Rn[23:16], Rd[23:16] = Rn[31:24]$
- **REVSH Rd, Rn:** Cambiar a orde dos bytes na media palabra baixa e extender o signo
 - $Rd[15:8] = Rn[7:0], Rd[7:0] = Rn[15:8]$
 - $Rd[31:16] = Rn[7] \& 0xFFFF$
- Útiles para cambiar entre formatos

Extensión de signo

- **SXTB Rd, Rm ,ROR #N:** Estender o signo dun byte
 - Copia o signo (0 ou 1) en $Rd[31:8]$, o número en $Rd[7:0]$
- **SXTH Rd, Rm ,ROR #N:** Estender o signo de media palabra
 - Copia o signo (0 ou 1) en $Rd[31:16]$, o número en $Rd[15:0]$
- **UXTB Rd, Rm ,ROR #N:** Reencher con ceros un byte
 - Copia o número en $Rd[7:0]$, enche de ceros $Rd[31:8]$
- **UXTH Rd, Rm ,ROR #N:** Reencher con ceros unha media palabra
 - Copia o número en $Rd[15:0]$, enche de ceros $Rd[31:16]$
- Con n podemos para de estender antes dos 32 bits

Extensión de signo

Os enteiros poden non ser todos de 32 bits. En C pódese pasar de un a outro. Este programa debería dar as operacións anteriores ao compilalo.

```
int_8    a = -1;    // a signed 8-bit integer, a = 0xFF
int_16   b = -2;    // a signed 16-bit integer, b = 0xFFFF
int_32   c;        // a signed 32-bit integer
c = a;          // sign extension, c = 0xFFFFFFFF
c = b;          // sign extension, c = 0xFFFFFFFF

uint_8   d = 1;    // an unsigned 8-bit integer, d = 0x01
uint_32  e;        // an unsigned 32-bit integer
e = d;          // zero extension, e = 0x00000001
```

Comparación

- **CMP RD, OP2:** Comparar
 - Realiza a resta $Rd - Op2$, marca os flags **NZCV**, descarta o resultado
- **CMN RD, OP2:** Comparar negativamente
 - Realiza a resta $Rd + Op2$, marca os flags **NZCV**, descarta o resultado
- **TST RD, OP2:** Comprobar (test)
 - Realiza un AND $Rd \& Op2$, marca os flags **NZ, C** (se barrel shifter), descarta o resultado.
 - Comproba se os bits indicados por $Op2$ (a 1) son 0 en Rd
- **TEQ RD, OP2:** comprobar equivalencia (test)
 - Realiza un OR $Rd | Op2$, marca os flags **NZ, C** (se barrel shifter), descarta o resultado.
 - Como CMP pero non toca **CV**, comproba se son iguais

A forma de responder a estas instrucións (ler **NZCV**) a veremos na sección de saltos e condicional

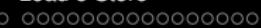
Mover entre rexistros

- **MOV RD, OP2:** Copiar dun rexistro a outro (ou inmediato)
 - $Rd = Op2$
- **MVN RD, OP2:** Copiar dun rexistro o negado a outro
 - $Rd = \neg Op2$
- **MRS RD, SPEC_REG:** Copiar dun rexistro especial a un xeral
- **MRS RD, SPEC_REG:** Copiar dun rexistro xeral a un especial
- Rexistros especiais: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, e CONTROL

AVISO: no simulador QtARMSim non ahi rexistros especiais, así que MRS, MSR non funcionan, pero compilan porque son Thumb

Extraer bits

- SBFX RD, RN, #LSB, #WIDTH: Copiar dun rexistro a outro so so bits entre LSB e WIDTH (con signo)
 - Extende o signo de Rn , rechea con ceros o resto
- UBFX RD, RN, #LSB, #WIDTH: Copiar dun rexistro a outro so so bits entre LSB e WIDTH (sen signo)
 - Rechea con 0
- Onde estas instrucións non estean dispoñibles, usar máscaras e operacións lóxicas, extender signo.



Índice

1 Linguaxe Ensamblador

- Arm
- Compoñentes dun microcontrolador
- C e Ensamblador
- Formato de instrucións
- Tipos de instruccións
- directivas

2 Arimética e lóxica

- Desprazamento e Rotación
- Aritméticas
- Lóxicas
- Outros

3 Load e Store

- Números Inmediatos
- Acceso a memoria

4 Saltos e condicional

5 Programación estructurada e subrutinas

Número inmediato a memoria

- **MOV RD, #<IMMEDIATE_8>**: Copiar a un rexistro un número inmediato
 - O número ten que entrar nos bits da instrución, temos un límite de 8 bits (255)
- **MVN RD, #<IMMEDIATE_8>**: Copiar a un rexistro o inverso dun número inmediato
- **MOVT RD, #<IMMEDIATE_16>**: Copiar a un a parte alta dun rexistro (bits 32-16) o número inmediato
- **MOVB RD, #<IMMEDIATE_16>**: Copiar a un a parte baixa dun rexistro (bits 15-0) o número inmediato
- As instrucións MOVT e MOVB son de 32 bits, por iso poden ter números más grandes
- Os números inmediatos poden ser en hexadecimal ou binarios:
 - #0xFF
 - #0b10011100
- Recordade: MOV e MVN tamén moven entre rexistros

Cargar/Load

LDR Pode ser unha instrución ou unha pseudo-instrución

- Unha pseudo-instrución non é unha instrución real, existe so na linguaxe para simplificar a programación
- **LDR Rt, =#<IMMED_8>**: Pseudo-instrución, igual a MVN Rd, #<IMMED_8>, poderíase cambiar (sen símbolo # no simulador)
- **LDR Rt, =#<IMMED_32>**: Pseudo-instrución, o número inmediato non entra nunha instrución, cambiase por varias instrucións (sen símbolo # no simulador)
- **LDR Rt, =<LABEL>**: Pseudo-instrución, a label indica unha posición de memoria
 - Neste casos gardaría o número ou a posición de memoria no código, preto da onde se carga
 - Usaría LDR cunha dirección relativa ao PC para cargalo (vese a continuación)

Cargar/Load

- ADR RT, #<IMMED_8>: Como LDR peso sen signo igual
 - Carga a un rexistro unha dirección de memoria
 - Tamén pseudo-instrucción
 - Tradúcese por un ADD ou SUB cun dos operandos como PC

Indexación con números inmediatos

Nos Cortex-M hai tres maneiras de indicar unha dirección de memoria:

- **Pre-indexación:**

- A dirección é indicado polo contido dun rexistro máis un número inmediato (que pode ser cero)

- Pre-indexación con actualización:

- A dirección é indicado polo contido dun rexistro máis un número inmediato
- Despois de acceder á memoria incrementase o rexistro usado co número

- Post-indexación:

- A dirección é indicado polo contido dun rexistro
- Despois de acceder á memoria incrementase o rexistro usado co número inmediato

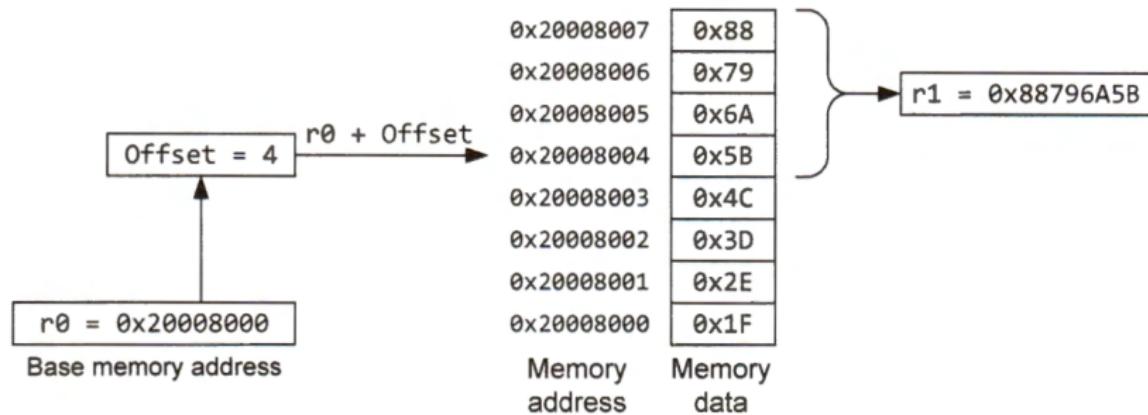
Indexación con números inmediatos

Nos Cortex-M hai tres maneiras de indicar unha dirección de memoria:

Memory Address Mode	Example	Equivalent
Pre-index	LDR r1, [r0, #4]	$r1 \leftarrow \text{memory}[r0 + 4]$, r0 remains unchanged.
Pre-index with update	LDR r1, [r0, #4]!	$r1 \leftarrow \text{memory}[r0 + 4]$ $r0 \leftarrow r0 + 4$
Post-index	LDR r1, [r0], #4	$r1 \leftarrow \text{memory}[r0]$ $r0 \leftarrow r0 + 4$

Indexación

Pre-Indexación:



Notación

- Rx é un rexistro, ou o seu valor contido actual
- $[Rx]$ indica que hai que usar o valor do rexistro para ir a unha dirección de memoria (iso chámase facer unha indirección)
 - O que se faga nesa dirección de memoria depende da instrución, xeralmente levar o dato a un rexistro (LDR) ou gardar o dato dun rexistro na dirección (STR)

Cargar/Load

- **LDR Rt, [Rn, #OFFSET]**: Copiar a un rexistro o contido da dirección de memoria indicada en $[Rn + offset]$ (+ as 3 seguintes)
 - $offset$ pode ser cero
- **LDRB Rt, [Rn, #OFFSET]**: Copiar un byte a un rexistro o contido da dirección de memoria indicada en $[Rn + offset]$ (So 1 byte)
- **LDRH Rt, [Rn, #OFFSET]**: Copiar media palabra a un rexistro o contido da dirección de memoria indicada en $[Rn + offset]$ (+ 1 seguinte)
- **LDM Rn, REGISTER_LIST**: Copiar múltiples palabras a múltiples rexistros

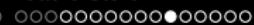
Cargar/Load

- LDRSB RT, [RN, #OFFSET]: Copiar byte con signo (estende o signo para completar)
- LDRSH RT, [RN, #OFFSET]: Copiar media palabra con signo (estende o signo para completar)
 - LDRSB e LDRSH existen porque se cargamos menos de 32 bits hai que estender o signo, ou o contido non sería válido



Gardar/Store

- **STR RT, [RN, #OFFSET]**: Garda dun rexistro á dirección de memoria indicada en $[Rn + offset]$ (+ as 3 seguintes)
 - $offset$ pode ser cero
- **STRB RT, [RN, #OFFSET]**: Garda un byte dun rexistro á dirección de memoria indicada en $[Rn + offset]$ (So o byte máis baixo)
- **STRH RT, [RN, #OFFSET]**: Garda media palabra dun rexistro á dirección de memoria indicada en $[Rn + offset]$ (+ 1 seguinte)
- **STM RN, REGISTER_LIST**: Garda múltiples palabras dos rexistros a partir da dirección de memoria $[Rn + offset]$



Load/Store con rexistros

- LDR, STR e o resto poden tamén usarse con dous rexistros (cambiar o offset cun rexistro)
- Tecnicamente son instrucións diferentes pero funcionan igual
- O valor do *offset* obtense de $[Rm]$
- **LDR Rt, [Rn, Rm]**
- **STR Rt, [Rn, Rm]**
- etc

Indexación relativa ao contador de programa

- Indicar direccións relativas ao contador de programa, PC, é moi común
- Aínda que non se use no programa o ensamblador converterá moitas instrucións a este modo
- Desta maneira a posición de memoria é independente de onde estea situado o programa na memoria real
- Pasa cando usamos a pseudo-instrucción LDR para cargar unha etiqueta ou inmediato:
 - O ensamblador escribe o valor da etiqueta (unha dirección) ou do inmediato no código
 - Converte a LDR a unha instrución de carga desa dirección do código, usando a indexación relativa ao PC
- O PC incrementase en 4 despois de ler a instrución
- A dirección de memoria a que apunta é $direccion_actual + 4 + inmediato$

Indexación relativa ao contador de programa

- LDR R1, #0xF1234567
- 0xF1234567 non entra como inmediato, moi grande
- O seu valor métese no código, 24 posicións más adiante
- $0x0800012C - 0x08000144 = 0x18 = 24$
- Como o PC incrementase en 4 xusto despois de ler a instrución hai que telo en conta
- Da igual onde na memoria se cargue o código

0x0800012C	LDR r1, [pc, #20] ; @0x08000144
...	...
0x08000144	DCW 0x4567 ; Lower halfword
0x08000146	DCW 0xF123 ; upper halfword

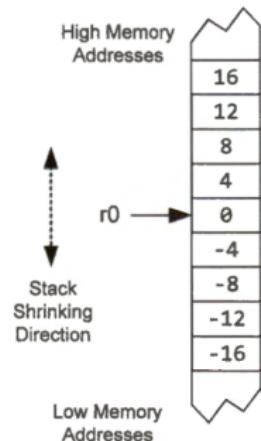
Cargar/Gardar múltiples rexistros*

- Coas instrucións **LDM RN, REGISTER_LIST, STM RN, REGISTER_LIST** pódense cargar/gardar múltiples rexistros
- En realidade son LDMXX, STMXX onde XX indica como se indexan esos rexistros.
- So veremos a normal, incrementar despois, o equivalente ao pre-indexado, IA
- Así empezase na dirección $[Rn]$ e se incrementa en 4 despois de cargar/gardar un rexistro
- A orde dos rexistros na instrución da igual, cárganse/escríbense por orde de nome (de 0 a 15)
- Os rexistros poden estar limitados aos baixos (0-7) nos procesadores más sinxelos
- $[Rn]$ pode incrementarse se se indica con !. En procesadores sinxelos (e no simulador) é obligatorio

Cargar/Gardar múltiples rexistros*

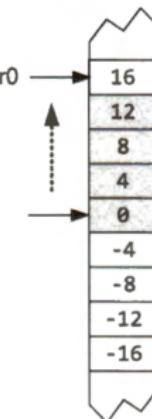
Pre-Indexación:

`LDMxx r0!, {r3,r1,r7,r2}`



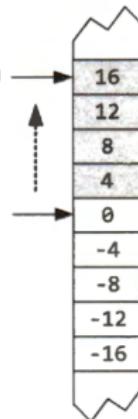
`LDMIA`

Increment After



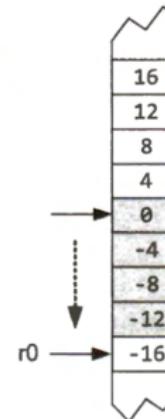
`LDMIB`

Increment Before



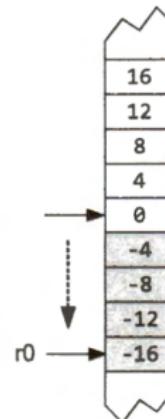
`LDMDA`

Decrement After



`LDMDB`

Decrement Before





Stalls/Paradas*

- Como podedes supoñer a anterior instrución parece que non se pode executar nun so ciclo
- Parece excesivo deseñar o hardware para poder ler múltiples rexistros á vez
- É unha instrución que claramente tarda máis
- As instrución non tardan todas o mesmo, algunas tardan máis



Índice

1 Linguaxe Ensamblador

- Arm
- Compoñentes dun microcontrolador
- C e Ensamblador
- Formato de instrucións
- Tipos de instruccións
- directivas

2 Arimética e lóxica

- Desprazamento e Rotación
- Aritméticas
- Lóxicas
- Outros

3 Load e Store

- Números Inmediatos
- Acceso a memoria

4 Saltos e condicional

5 Programación estructurada e subrutinas

Índice

1 Linguaxe Ensamblador

- Arm
- Compoñentes dun microcontrolador
- C e Ensamblador
- Formato de instrucións
- Tipos de instruccións
- directivas

2 Arimética e lóxica

- Desprazamento e Rotación
- Aritméticas
- Lóxicas
- Outros

3 Load e Store

- Números Inmediatos
- Acceso a memoria

4 Saltos e condicional

5 Programación estructurada e subrutinas

Referencias

Imaxes desde:

Y. Zhu. "Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C". Third Edition. E-Man Press LLC. 2017. ISBN-13: 978-0982692660