Sistemas Encaixados

Tema 2: Arquitectura dos Sistemas Encaixados

Oscar García Lorenzo

Escola Politécnica Superior de Enxeñaría

Arquitectura dos Sistemas Encaixados

1. Linguaxe Ensamblador
2. Arimética e lóxica
3. Load e Store
4. Saltos e condicional
   - Comprobación de Condición
   - Saltos
   - Estruturas de control
   - Reuso de Rexistros
5. Subrutinas
6. Atmel ARV

Índice

1. Linguaxe Ensamblador
2. Arimética e lóxica
3. Load e Store
4. Saltos e condicional
   - Comprobación de Condición
   - Saltos
   - Estruturas de control
   - Reuso de Rexistros
5. Subrutinas
6. Atmel ARV

Índice

1. Linguaxe Ensamblador
2. Arimética e lóxica
3. Load e Store
4. Saltos e condicional
   - Comprobación de Condición
   - Saltos
   - Estruturas de control
   - Reuso de Rexistros
5. Subrutinas
6. Atmel ARV

Índice

1. Linguaxe Ensamblador
2. Arimética e lóxica
3. Load e Store
4. Saltos e condicional
   - Comprobación de Condición
   - Saltos
   - Estruturas de control
   - Reuso de Rexistros
5. Subrutinas
6. Atmel ARV

Índice

1. Linguaxe Ensamblador
2. Arimética e lóxica
3. Load e Store
4. Saltos e condicional
   - Comprobación de Condición
   - Saltos
   - Estruturas de control
   - Reuso de Rexistros
5. Subrutinas
6. Atmel ARV

Fluxo do programa

- Normalmente as instruccións execútanse unha tras outra
- O contador de programa (PC) incrementanse en 4 tras ler unha instrucción
- Modificar o PC implica modificar a orde de execución
- Os modos de cambiar o fluxo son:
  - Instruccións de salto
  - Execución condicional
  - Chamada a subrutina
  - Interrupcións

Comprobación de condición

- Para escoller se realizar os saltos/execucións úsanse as bandeiras **N**, **Z**, **C** e **V**
- Hai instruccións que se executan dependendo desas bandeiras
  - Execución condicional
  - En Thumb, microcontroladores sinxelos, son poucas
  - En Thumb-2, con instruccións 32 bits, son moitas
- Úsase a mesma nomenclatura (sufixo) para todas as instruccións que o permiten

Compare	Signed	Unsigned	Relationship Tested
==	EQ	EQ	Equal to
!=	NE	NE	Not equal to
>	GT	HI	Greater than
≥	GE	HS	Greater than or equal to
<	LT	LO	Less than
≤	LE	LS	Less than or equal to

Comprobación de condición

- Moitas instruccións actualizan **N**, **Z**, **C** e **V**
- O normal é usar CMP, que é como una resta pero descarta o resultado
  - **CMP RD, OP2**
- Depois, según as bandeiras, executase a seguinte instrucción

Exemplo: Calcular valor absoluto

CMP R1, #0 ;compara e actualiza N,Z,C,V
RSB**BT** R1, R1 #0 ;resta inversa, r1=0-r1 se r1<0, **LT, menor que, con signo**

Comprobación de condición

As 15 comparacións en Cortex-M

Suffix	Description	Flags tested	Logic Implementation
EQ	EQual	Z = 1	Z
NE	Not Equal	Z = 0	$\bar{Z}$
CS/HS	unsigned Higher or Same	C = 1	C
CC/LO	unsigned LOwer	C = 0	$\bar{C}$
MI	MInus (negative)	N = 1	N
PL	PLus (positive or zero)	N = 0	$\bar{N}$
VS	oVerflow Set	V = 1	V
VC	oVerflow Clear	V = 0	$\bar{V}$
HI	unsigned HIgher	C = 1 & Z = 0	$C\bar{Z}$
LS	unsigned Lower or Same	C = 0 or Z = 1	$\bar{C} + Z$
GE	signed Greater or Equal	N = V	$NV + \bar{N}\bar{V}$
LT	signed Less Than	N != V	$N\bar{V} + \bar{N}V$
GT	signed Greater Than	Z = 0 & N = V	$\bar{Z}(NV + \bar{N}\bar{V})$
LE	signed Less than or Equal	Z = 1 or N ! = V	$Z + N\bar{V} + \bar{N}V$
AL	ALways		

Comprobación de condición

- Non poñer nada equivale a AL
- Ás veces non chega con ver N ou Z
- CMP R0, R1 equivale a unha resta, así que, para sen signo, **C**:
  - Se $r0 \geq r1$, non hai préstamo, **C=1**
  - Se $r0 < r1$, hai préstamo, **C=0**
- CMP R0, R1 equivale a unha resta, así que, para con signo, **V**:
  - Hai 4 combinacións posíbeis de **V** e **N**

	N=0	N=1
V=0	$r0 \geq r1$	$r0 < r1$
V=1	$r0 < r1$	$r0 \geq r1$

  - Se **V=1** o resultado é incorrecto e **N** está ao revés
- Por iso hai comparacións que comproban dúas bandeiras

Comprobación de condición

V en detalle

- **V=0,N=0**: Non hai desborde, o resultado é correcto e non negativo, $r0 - r1 \geq r0$, $r0 \geq r1$
- **V=0,N=1**: Non hai desborde, o resultado é correcto e negativo, $r0 - r1 < 0$, $r0 < r1$
- **V=1,N=0**: Hai desborde, o resultado é incorrecto debería ser negativo, $r0 - r1 < 0$, $r0 < r1$
- **V=1,N=1**: Hai desborde, o resultado é incorrecto debería ser non negativo, $r0 - r1 \geq r0$, $r0 \geq r1$
- Polo tanto:
  - Se $N = V$, $r0$ ten signo e é maior ou igual a $r1$
  - Se $N \neq V$, $r0$ ten signo e é menor que $r1$

Saltos

- Hai unha instrucción de salto incondicional
  - O salto realizase sempre
- En ensamblador sempre usase unha etiqueta para indicar o destino
- Carga o valor da etiqueta ao PC
- **B *label***: Salto incondicional á etiqueta
- Se se combina cos sufixos anteriores consiguense saltos condicionais
  - O salto realizase se a condición se cumple
- Non todos os procesadores Cortex-M soportan todas as condicións directamente, pero pódese usar igualmente:
  - O compilador as traduce a instruccións `if-then-else (IT)`

Saltos Condicionais

	Instruction	Description	Flags tested
Unconditional Branch	B *label*	Branch to label	none
Conditional Branch	BEQ *label*	Branch if EQual	Z = 1
	BNE *label*	Branch if Not Equal	Z = 0
	BCS/BHS *label*	Branch if unsigned Higher or Same	C = 1
	BCC/BLO *label*	Branch if unsigned LOwer	C = 0
	BMI *label*	Branch if MInus (Negative)	N = 1
	BPL *label*	Branch if PLus (Positive or Zero)	N = 0
	BVS *label*	Branch if oVerflow Set	V = 1
	BVC *label*	Branch if oVerflow Clear	V = 0
	BHI *label*	Branch if unsigned HIgher	C = 1 & Z = 0
	BLS *label*	Branch if unsigned Lower or Same	C = 0 or Z = 1
	BGE *label*	Branch if signed Greater or Equal	N = V
	BLT *label*	Branch if signed Less Than	N != V
	BGT *label*	Branch if signed Greater Than	Z = 0 & N = V
	BLE *label*	Branch if signed Less than or Equal	Z = 1 or N = !V

Saltos Condicionais

Comparison	Signed	Unsigned
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
≥	BGE	BHS
<	BLT	BLO
≤	BLE	BLS

Saltos Condicionais

- Con estas instruccións (e CMP) fanse os `if-then-else`, `while`, `for` ...
- Non son iguais para enteiros con e sen signo
  - $a = 0xFFFFFFFF$, $b = 0x00000001$
  - Sen signo $a > b$
  - Con signo $b > a$
- Ao programar en ensamblador hai que ter coidado
- Ao programar en C ou outra linguaxe o compilador encárgase de escoller as correctas

Exemplo If-Then

Exemplo con enteiros sen signo

C Program	Assembly Program
`uint32_t x, y, z;`	`MOV r5, #0x00000001 ; r5 = x`
`x = 0x00000001;`	`MOV r6, #0xFFFFFFFF ; r6 = y`
`y = 0xFFFFFFFF;`	`CMP r5, r6`
	`BLS else ; branch if ≤`
`if (x > y)`	`then MOV r7, #1 ; z = 1`
`z = 1;`	`B endif ; skip next instruction`
`else`	`else MOV r7, #0 ; z = 0`
`z = 0;`	`endif`

Exemplo If-Then

Exemplo con enteiros sen signo. Ollo a trampa do nome da etiqueta, da igual

C Program	Assembly Program
`int32_t x, y, z;`	`MOVS r5, #0x00000001 ; r5 = x`
	`MOVS r6, #0xFFFFFFFF ; r6 = y`
`x = 1; // 0x00000001`	
`y = -1; // 0xFFFFFFFF`	`CMP r5, r6`
	`BLE then ; branch if signed ≤`
`if (x > y)`	`MOVS r7, #1 ; z = 1`
`z = 1;`	`B endif ; skip next instruction`
`else`	`then MOVS r7, #0 ; z = 0`
`z = 0;`	`endif`

Saltos Condicionais

- Os saltos que comparan con 0 son moi comúns
- Por iso existen instruccións que saltan se 0:
  - CBZ RN, LABEL: Salta se $rn = 0$
  - CBNZ RN, LABEL: Salta se $rn = 1$
  - Aforran unha instrucción CMP RN, #0 antes do salto, Thumb non as ten
- Hai instruccións especias de salto a subrutina BL, BLX e BX, xa as veremos

Execución Condicional

- Arm permite a execución condicional (non o simulador de prácticas)
- É algo raro na maoiría de arquitecturas
- Engadindo o sufixo de comparación podemos indicar instruccións condicionais:
  - ADDNE RD, RN, OP2: Suma se non son iguais
  - SUBMI RD, RN, OP2: Resta se negativo
- Simplifican programación (se estás afeito)
  - Incrementa $r3$ se $r0 > r1$
    CMP R1, R0
    ADDNE R3, R3, #1

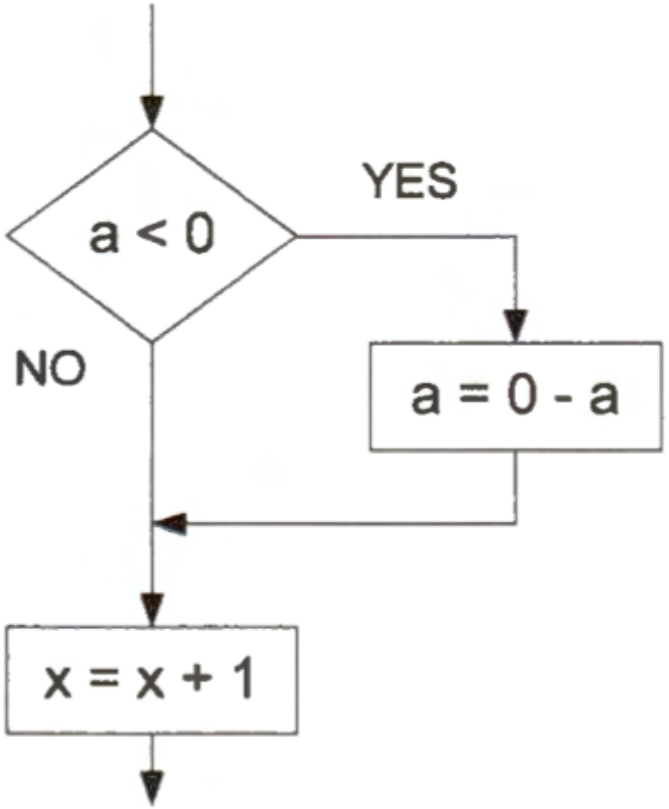
If-Then

- Unha estrutura `if-then` executa un bloque de código se se cumple unha condición booleana
- Se non se cumple a condición o bloque sáltase
- En ensamblador pódense implementar cunha comparación e un salto

If-Then

C Program

```
if (a < 0 ) {
   a = 0 - a;
}
x = x + 1;
```




```
        ; r1 = a, r2 = x
        CMP r1, #0          ; Compare a with 0
        BGE endif           ; Go to endif if a ≥ 0
then    RSB r1, r1, #0      ; a = - a
endif   ADD r2, r2, #1      ; x = x + 1
```

If-Then

- Moitas veces a condición é composta
- En C: And:&&, OR:||, NOT:!
- En ensamblador;
  - Con OR non hai moito problema, basta con ir comprobando cada condición por separado
  - Con AND é máis dificil, pódense usar as condicións de De Morgan:
    - $\neg(A\&\&B) = \neg A||\neg B$
    - Negar a condición e pasar a OR

$\neg((x > 20)\&\&(x < 25)) = \neg(x > 20)||\neg(x < 25) = (x \leq 20)|(x \geq 25)$

If-Then

C Program	Assembly Program

`if(x > 20 && x < 25){`
`  a = 1;`
`}`
	`; Assume r0 = x, r1 = a`
	`CMP  r0, #20  ; compare x with 20`
	`BLE  endif  ; go to endif if x <= 20`
	`CMP  r0, #25  ; compare x with 25`
	`BGE  endif  ; go to endif if x >= 25`
	`MOVS r1, #1  ; a = 1`
	`endif`

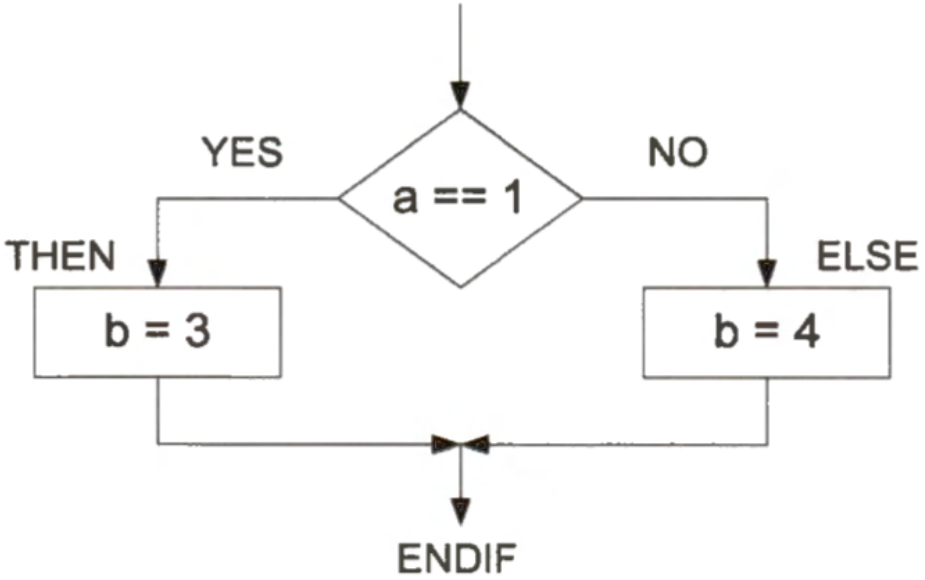
If-Then-Else

- Unha estrutura `if-then-else` executa un bloque de código se se cumple unha condición booleana e outro diferente se non
- En ensamblador pódense implementar cunha comparación e dous saltos

If-Then-Else

C Program

```
if (a == 1)
    b = 3;
else
    b = 4;
```



Assembly Program 1

```
        ; r1 = a, r2 = b
        CMP r1, #1   ; compare a and 1
        BNE else     ; go to else if a ≠ 1
then    MOV r2, #3   ; b = 3
        B   endif    ; go to endif
else    MOV r2, #4   ; b = 4
endif
```

Assembly Program 2

```
; r1 = a, r2 = b
CMP   r1, #1   ; compare a and 1
MOVEQ r2, #3   ; b = 3 if a = 1
MOVNE r2, #4   ; b = 4 if a ≠ 1
```

Bucle For

- Un bucle `for` executa un bloque de código múltiples veces mentres se satisfaga unha condición
- Consta de 3 expresións:
  - Expresión inicial: Execútase so unha vez, xeralmente para inicializar os índices
  - Expresión condicional: Compróbase ao principio de cada iteración, o corpo do bucle execútase se é verdadeira. Se é falsa ao principio da primeira iteración o corpo non se debe executar
  - Exepresión de bucle: Execútase ao final de cada iteración, normalmente para incrementar ou decrementar índices
- En ensamblador necesita unha comparación e dous saltos, máis a inicialización

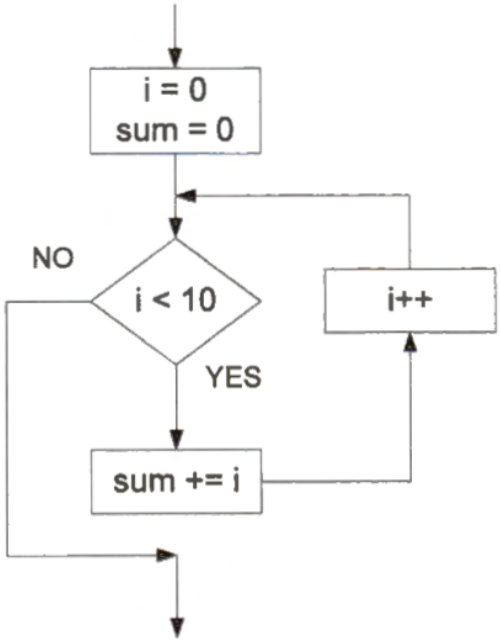
```
for (initial_expression; condition_expression; loop_experssion) {
  // loop body
  ...
}
```

Bucle For

C Program

```
int i;
int sum = 0;

for(i = 0; i < 10; i++){
  sum += i;
}
```



Assembly Program 1

```
      MOV r0, #0   ; i
      MOV r1, #0   ; sum

      B   check
loop  ADD r1, r1, r0
      ADD r0, r0, #1
check CMP r0, #10
      BLT loop
endloop
```

Assembly Program 2

```
      MOV r0, #0   ; i
      MOV r1, #0   ; sum

loop  CMP r0, #10
      BGE endloop
      ADD r1, r1, r0
      ADD r0, r0, #1
      B   loop
endloop
```

Assembly Program 3

```
      MOV r0, #0   ; i
      MOV r1, #0   ; sum

loop  CMP    r0, #10
      ADDLT  r1, r1, r0
      ADDLT  r0, r0, #1
      BLT    loop
endloop
```

Bucle While

- Un bucle `while` executa un bloque de código múltiples veces mentres se satisfaga unha condición
- A condición ompróbase ao principio de cada iteración, o corpo do bucle execútase se é verdadeira. Se é falsa ao principio da primeira iteración o corpo non se debe executar
- Pódese expresar coma un for, en ensamblador son equivalentes
- En ensamblador necesita unha comparación e dous saltos, máis a inicialización

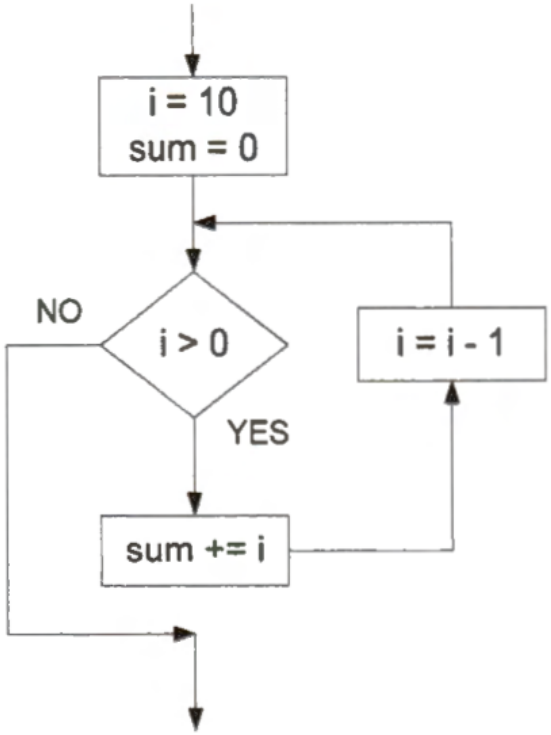
```
while (condition_expression) {
  // loop body
  ...
}
```

Bucle While

C Program

```
int i = 10;
int sum = 0;

while( i > 0 ){
  sum += i;
  i--;
}
```



Assembly Program 1	Assembly Program 2	Assembly Program 3

`MOV r0, #10 ; i`
`MOV r1, #0 ; sum`
 	
`B check`
`loop ADD r1, r1, r0`
`SUB r0, r0, #1`
`check CMP r0, #0`
`BGT loop`
`endloop`
	`MOV r0, #10 ; i`
	`MOV r1, #0 ; sum`
	
	`loop CMP r0, #0`
	`BLE endloop`
	`ADD r1, r1, r0`
	`SUB r0, r0, #1`
	`B loop`
	`endloop`
		`MOV r0, #10 ; i`
		`MOV r1, #0 ; sum`
		
		`loop CMP r0, #0`
		`ADDGT r1, r1, r0`
		`SUBGT r0, r0, #1`
		`BGT loop`
		`endloop`

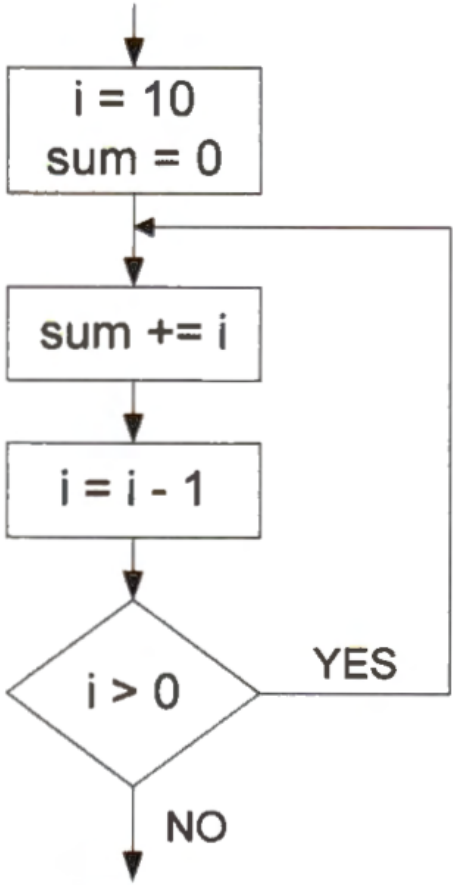
Bucle Do-While

- Un bucle `do-while` executa un bloque de código múltiples veces mentres se satisfaga unha condición
- Igual que `while`, pero a condición compróbase ao final de cada iteración. Se é falsa ao principio da primeira iteración o corpo executase polo menos unha vez

Bucle Do-While

C Program

```
int sum = 0;
int i = 10;
do{
    sum += i;
    i--;
} while( i > 0 );
```



Assembly Program 1

```
        MOV r0, #10      ; i = 10
        MOV r1, #0       ; sum = 0

loop    ADD r1, r1, r0   ; sum += i
        SUB r0, r0, #1   ; i--
        CMP r0, #0
        BGT loop
endloop
```

Assembly Program 2

```
        MOV  r0, #10     ; i = 0
        MOV  r1, #0      ; sum = 0

loop    ADD  r1, r1, r0  ; sum += i
        SUBS r0, r0, #1  ; i--
        BGT  loop
endloop
```

Continue

- Unha sentencia `continue` indica non facer nada máis e pasar á seguinte iteración dun bucle

C Program	Assembly Program

`int i;`
`int sum = 0;`
 	
`for(i = 0; i < 10; i++) {`
`  if (i == 5) // skip 5`
`  continue;`
`  sum += i;`
`}`
	`  MOVS r0, #0  ; i = 0`
	`  MOVS r1, #0  ; sum = 0`
	
	`loop  CMP r0, #10`
	`  BGE endloop`
	`  CMP r0, #5`
	`  ADDNE r1, r1, r0  ; sum += i`
	`  ADD r0, r0, #1  ; i++`
	`  B loop`
	`endloop`

Break

- Unha sentencia `break` indica non facer nada máis e sair do bucle
- Se hai varios bucles aniñados so debe sair do máis cercano (do que está dentro)

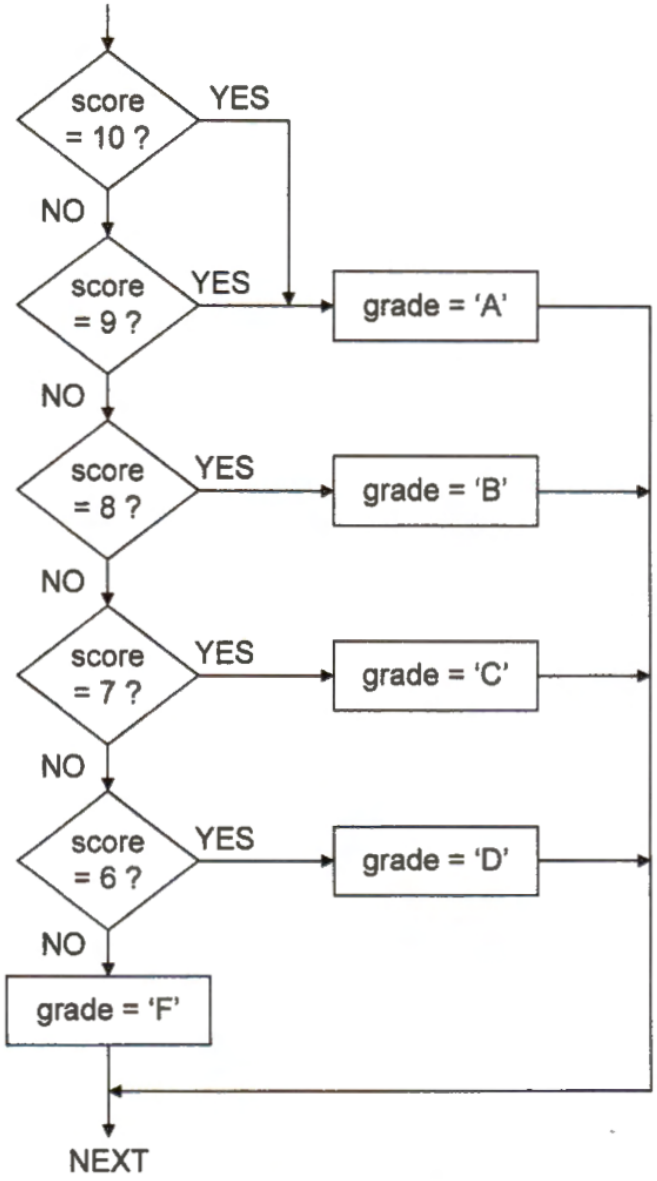
C Program	Assembly Program

`// Find string length`
`char str[] = "hello";`
`int len = 0;`
`char *p = str;`
 	
`for( ; ; ) {`
 	
`  if (*p == '\0')`
`  break;`
 	
`  p++;`
`  len++;`
`}`
	`; r0 = string memory address`
	`; r1 = string length`
	`MOV r1, #0  ; len = 0`
	
	`loop  LDRB r2, [r0]`
	
	`CBNZ r2, notZero`
	`B  endloop`
	
	`notZero  ADD r0, r0, #1  ; p++`
	`ADD r1, r1, #1  ; len++`
	
	`B  loop`
	`endloop`

Switch

- Unha sentencia `switch` permite executar distintos bloques de código dependendo do valor dunha expresión
- A expresión debe coincidir cunha serie de valores enteiros predeterminados
- Cando hai múltiples opcións fai o programa máis estruturado e fácil de ler que con `if-then-else`
- En Arm hai unha isntrucción, TBB, que permite definir unha táboa de bytes que indica o deprazamento do PC para cada condición, pero non a imos ver

Bucle Do-While



Reuso de Rexistros

- O número de rexistros é limitado
  - Moitas instrucción so poden usar do 0 ao 7
- Se se acaban os rexistros hai que gardar as varibales á memoria
  - Os accesos a memoria son caros
- Os rexistros se reusan se están fora do seu rango de vida

Rango de vida dun rexistro

O intervalo entre que se escribe nun rexistro e se le por derradeira vez antes de volverse a escribir. Un rexistro pode ter múltiples rangos de vida ao longo dun programa.

Reuso de Rexistros

Global variables

```
int A = 0;  // 0x00000000
int B = -1; // 0xFFFFFFFF
int C = -2; // 0xFFFFFFFE
int D = 2;  // 0x00000002

void main(void){
  A = B + C - D;
  return;
}
```

Data memory

Address	Data	
0x2000,000F	0x00	D = 0x0000,0002 = 2
0x2000,000E	0x00	
0x2000,000D	0x00	
0x2000,000C	0x02	
0x2000,000B	0xFF	C = 0xFFFF,FFFE = -2
0x2000,000A	0xFF	
0x2000,0009	0xFF	
0x2000,0008	0xFE	
0x2000,0007	0xFF	B = 0xFFFF,FFFF = -1
0x2000,0006	0xFF	
0x2000,0005	0xFF	
0x2000,0004	0xFF	
0x2000,0003	0x00	A = 0x0000,0000 = 0
0x2000,0002	0x00	
0x2000,0001	0x00	
0x2000,0000	0x00	

Assembly instructions

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B      ; r2 = 0x2000,0004
    LDR r3, [r2]    ; r3 = B = -1
    LDR r4, =C      ; r4 = 0x2000,0008
    LDR r5, [r4]    ; r5 = C = -2
    LDR r6, =D      ; r6 = 0x2000,000B
    LDR r7, [r6]    ; r7 = D = 2
    ADD r1, r3, r5  ; r1 = B + C
    SUB r1, r1, r7  ; r1 = B + C - D
    LDR r0, =A      ; r0 = 0x2000,0000
    STR r1, [r0]    ; Save A
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

Reuso de Rexistros

- No primeiro paso cambiamos $r4$, $r6$ e $r0$ por $r2$ e $r1$ por $r3$.
  - Pasamos de 8 a 4
- No segundo paso cambiamos $r7$ por $r2$.
  - Pasamos de 4 a 3
- Seguimos repetindo ata o necesario

Reuso de Rexistros

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B        Lifetime of r2
    LDR r3, [r2]
    LDR r4, =C        Lifetime of r4
    LDR r5, [r4]
    LDR r6, =D        Lifetime of r6
    LDR r7, [r6]
    ADD r1, r3, r5    Lifetime of r3
    SUB r1, r1, r7
    LDR r0, =A        Lifetime of r0
    STR r1, [r0]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

8 registers used

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B        Lifetime of r2
    LDR r3, [r2]
    LDR r2, =C
    LDR r5, [r2]
    LDR r2, =D
    LDR r7, [r2]
    ADD r3, r3, r5
    SUB r3, r3, r7
    LDR r2, =A        Lifetime of r2
    STR r3, [r2]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

4 registers used

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B
    LDR r3, [r2]
    LDR r2, =C
    LDR r5, [r2]
    LDR r2, =D
    LDR r2, [r2]      Reus
    ADD r3, r3, r5    r2
    SUB r3, r3, r2
    LDR r2, =A
    STR r3, [r2]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

3 registers used

Índice

1. Linguaxe Ensamblador
2. Arimética e lóxica
3. Load e Store
4. Saltos e condicional
   - Comprobación de Condición
   - Saltos
   - Estruturas de control
   - Reuso de Rexistros
5. Subrutinas
6. Atmel ARV

Subrutinas

Subrutina

Fragmento de código independente cun propósito determinado e un nome

- AKA: Función, procedemento, rutina
- Permite o reuso dun código que cumple unha función específica
- Incrementa a facilidade de comprensión dun programa
- Facilita a programación
- Usalmente:
  - Toman uns parámetros de entrada (argumentos)
  - Devolve un resultado

Subrutinas

Cando se programa en ensamblador hai que considerar:

- Preservación e recuperación do entorno do invocador:
  - Quen chama á subrutina non pode ver cambiado o contido dos seus rexistros, aínda os pode necesitar
  - A subrutina ten que gardar o contido dos rexistros que vai a usar na pila
  - Ao final da subrutina, esta debe volver a meter o contido nos rexistros desde a pila
- Debe de seguir o procedemento estándar para recibir as entradas e devolver os resultados:
  - Arm EABI

Protocolo Arm EABI para subrutinas

Protocolo Arm EABI para subrutinas

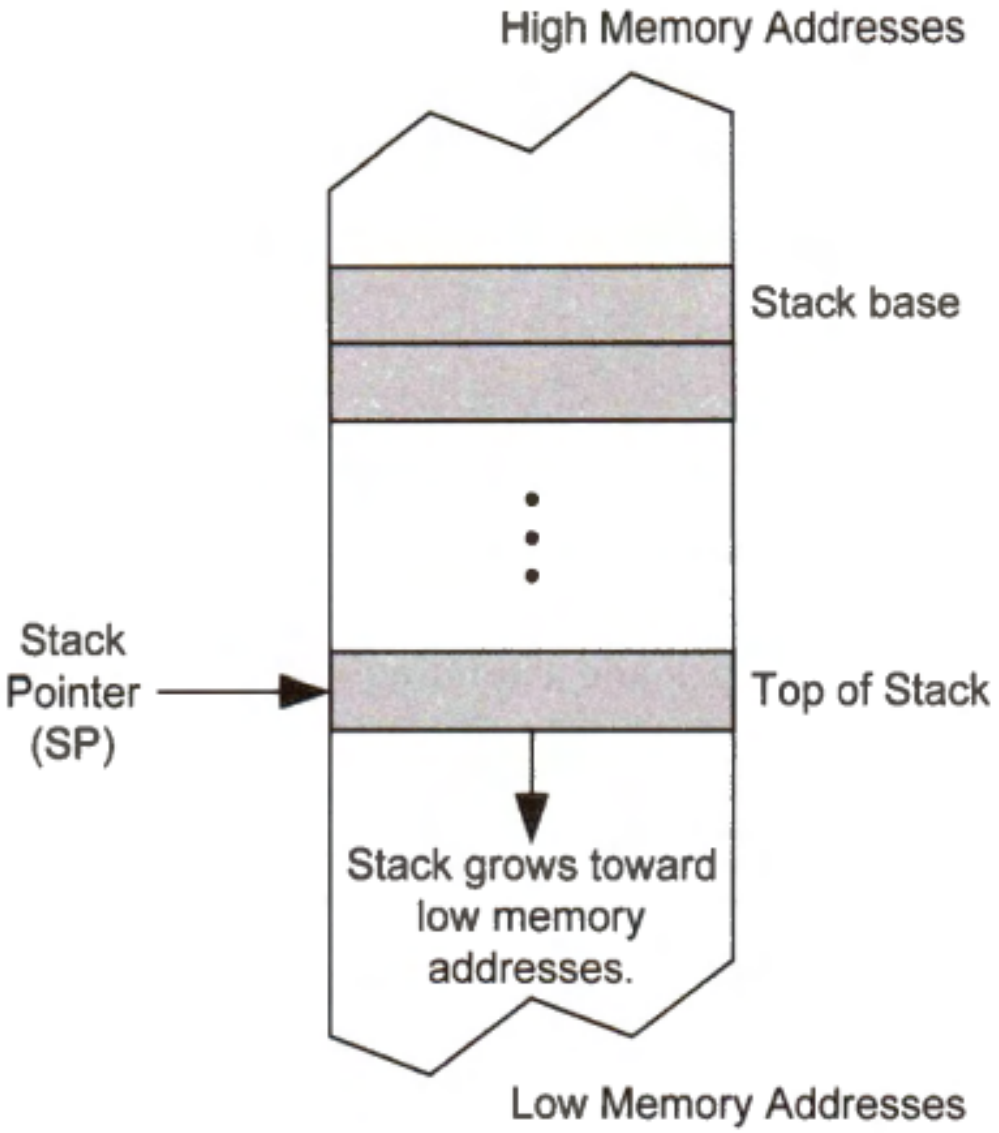
Protocolo que indica como usar subrutinas en Arm

- Entrada
  - Ata 4 argumentos de 32 bits: métense nos rexistros $r0$, $r1$, $r2$ e $r3$
  - Ata 2 argumentos de 64 bits: rexistros $r0$, $r1$, $r2$ e $r3$, cos msb en $r1$ e $r3$
  - 1 argumento de 128 bits en $r0$, $r1$, $r2$ e $r3$, msb en $r3$
  - Cando son máis de 4 argumentos de 32 bits, os primeiros en $r0$, $r1$, $r2$ e $r3$ e o resto na pila
- Saída
  - 1 valor de 32 bits: $r0$
  - 1 valor de 64 bits: $r0$, $r1$
  - 1 valor de 128 bits: $r0$, $r1$, $r2$ e $r3$

A Pila

- Cortex-M usa pilas de descenso, con pilas separadas para procesos e xeral (main and process), pero non nos imos liar
- Unha pila (ou cola LIFO, Last In First Out) é unha estrutura de datos que permite extraer datos na orde contraria na que foron introducidos
- Engadir datos é apilar (push)
- Extraer datos é desapilar (pop)
- A dirección do último elemento, o tope da pila, SP
- **PUSH R4**: Mete na pila
  - **SUB SP, SP, #4**
  - **STR R4, [SP]**
- **POP R4**: Saca da pila
  - **LDR R4, [SP]**
  - **ADD SP, SP, #4**

A Pila



PUSH {register_list}
equivalent to:
STMDB SP!, {register_list}

DB: Decrement Before

POP {register_list}
equivalent to:
LDMIA SP!, {register_list}

IA: Increment After

A Pila

A Pila úsase para:

- Almacenamento e recuperación do contido dos rexistros
- Almacenamento e recuperación das direccións de retorno para permitir aniñamento de subrutinas
- Creación e utilización de variables locáis da subrutina

Bloque de activación

Parte da pila coa información dubha subrutina

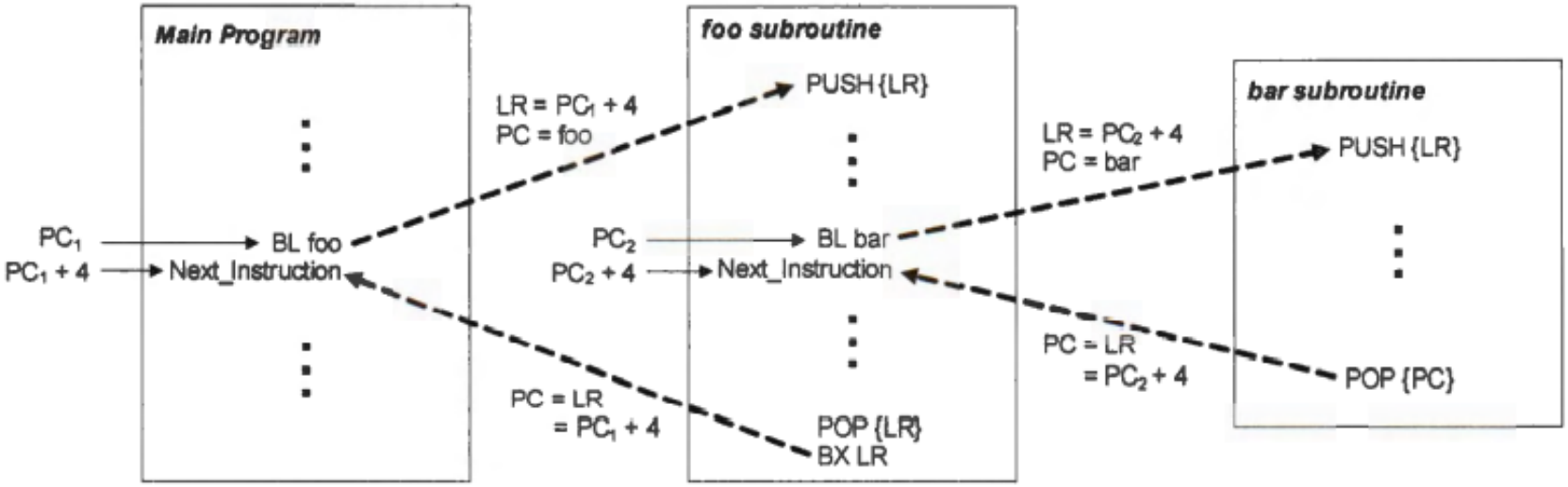
- Dirección de retorno orixinal
- Espazo para variables locais
- Almacenamento de rexistros modificados
- Valores pasados coma argumentos

Chamadas a Subrutinas

- Entrar na subrutina: **BL ETIQUETA**: Branch and Link
  - Almacena a dirección da seguinte instrucción (PC+4) no rexistro LR (Link Register)
  - Salta a subrutina (salto incondicional á etiqueta) (pon o PC ao valor da dirección da primeira instrucción da subrutina)
  - Ten 32 bits, pero funciona no simulador
  - O valor en LR é a dirección de retorno
- Sair da subrutina: **MOV PC, LR**: Mete no PC o valor da dirección de retorno (gardada en LR)
  - Podería usarse BX LR: Branch and link
  - O valor de LR podería estar na pila, nese caso POP PC

Chamadas a Subrutinas

- Nas instruccións BL sáltase ás subrutinas
- Se se aniñan as chamadas o valor de LR ten que gardarse na pila (conservar o entorno)
- Aquí `bar()` no tería porque gardar LR na pila
- `foo()` ten que gardar LR na pila porque vai facer outra chamada a unha subrutina



Perservación do entrono de execución

ARM embedded application binary interface (EABI):

- Os rexistros $r4 - r11$ hai que perservalos
  - Se se modifican o seu valor orixinal hai que metelo na pila e sacalo ao final da subrutina
- Os rexistros $r0 - r3$ pódense modificar
  - Chamanse memoria de scratch
  - Se o invocador os quere conservar ten que facelo el (metendoos antes na pila)
- O rexistro LR non é necesario conservalo, pero recoméndase (polo aniñamento)
- A orde dos rexistros nunha instrucción PUSH ou POP non importa, van sempre de maior a menor

Paso de variables

Paso por valor

- Á subrutina chega o valor da variable, o dato en si
- No invocador o valor desa variable non pode cambiar
- O normal en C
- En ensamblador:
  - Non é o mesmo o rexistro que a variable
  - A variable gárdase en memoria
  - Pásase o seu valorá subrutina nun rexistro
  - A subrutina non pode cambiar o seu valor, non sabe onde está
- Para parámetros de entrada

Paso de variables

Paso por referencia

- Á subrutina chega a dirección en memoria da variable
- No invocador o valor desa variable pode cambiar
- En C simúlase co paso de direccións (&)
- En ensamblador:
  - A variable gárdase en memoria
  - Pásase a sua dirección á subrutina nun rexistro
  - A subrutina pode cambiar o seu valor
- Para parámetros de entrada/saída ou saída

Bloque de activación

Antes de chamar á subrutina

No invocador

- Gardar nos rexistros *r*0-*r*3 os argumentos de entrada
- (opcional) Gardar na pila os valores de *r*0-*r*3 se se queren conservar
  - Exemplo: **PUSH R0-R3**
- Gardar na pila os argumentos de entrada que non entren nos rexistros
  - Exemplo: **PUSH R4,R5** //2 parámetros extra

Bloque de activación

Na entrada á subrutina

Na subrutina, xusto tras tomar o control

- Apilar na pila todos os rexistros $r4 - r11$ que se vaian a modificar, tamen o LR
  - Exemplo: **PUSH R6,R7** //Imos usar $r6$ e $r7$
- Reservar memoria na pila para as variables que se vaian a usar (facelo de 4 en 4 bytes)
  - Exemplo: **SUB SP,SP,#8** //Imos usar dúas variables (4+4)

Bloque de activación

Na saída da subrutina

Na subrutina, xusto antes de volver

- Cargar os valores a devolver en $r0 - r3$
- Liberar o espazo das variables locais
  - Exemplo: **ADD SP,SP,#8** //Movemos o punteiro da pila a antes das dúas variables (4+4)
- Restaurar o valor orixinal dos rexistros (incluido o LR, sobre PC)
  - Exemplo: **POP R6,R7** //Recuperamos $r6$ e $r7$

Bloque de activación

Ao volver ao invocador

No invocador, xusto despois de volver

- Eliminar da pila os parámetros que se apilaran
  - Exemplo: **POP R4,R5** ou **ADD SP,SP,#8** //Recuperamos os parámetros ou non nos importan
- Recoller os resultados
- (opcional) Recoller da pila os valores de $r0$-$r3$
  - Exemplo: **POP R0-R3**

Funcións Recursivas

Funcións Recursivas

Funcións que se invocan a sí mesmas

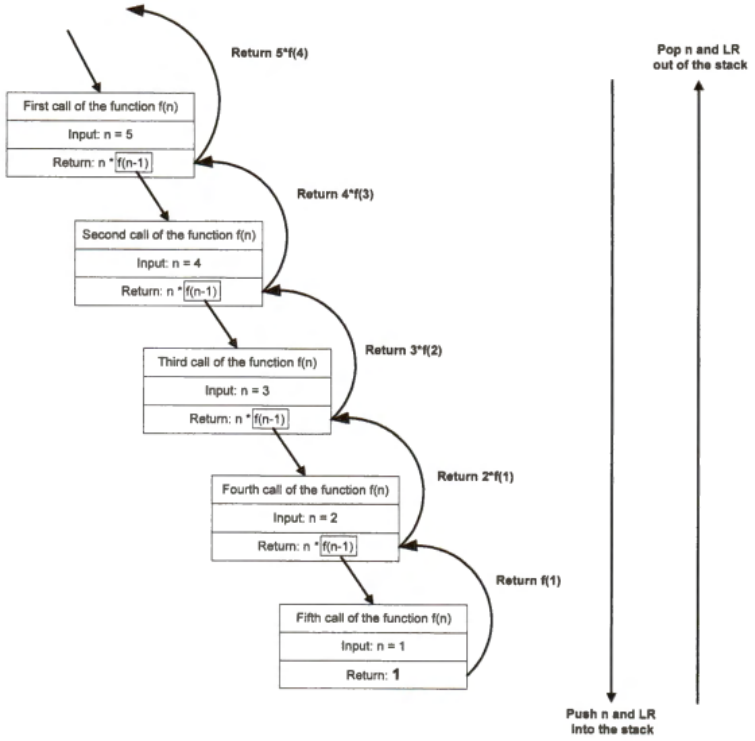
- Xeralmente van traballando con datos menores de cada vez
- É un eficaz divide e vencerás
- Todo o que se resolve cunha función recursiva pódese facer con unha iterativa (con bucles)
- Poden ser máis fáciles de programar e entender
- PERO: requiren moita máis memoria
  - Agora que coñecedes como son as chamadas a subrutina podedes entender por que

Funcións Recursivas

Recursive Function	Iterative Function

`int factorial(int n) {`
`  if(n==1)`
`  return 1;`
`  else`
`  return n * factorial(n-1);`
`}`
 	
`int main(void){`
`  int y;`
`  y = factorial(5);`
`  return 0;`
`}`
	`int factorial(int n) {`
	`  result = 1;`
	`  for (int i = 1; i < n; i++)`
	`  result *= i;`
	`  return result;`
	`}`
	
	`int main(void){`
	`  int y;`
	`  y = factorial(5);`
	`  return 0;`
	`}`

Example 8-21. Factorial function implemented by using recursive and iterative function



Índice

1. Linguaxe Ensamblador
2. Arimética e lóxica
3. Load e Store
4. Saltos e condicional
   - Comprobación de Condición
   - Saltos
   - Estruturas de control
   - Reuso de Rexistros
5. Subrutinas
6. Atmel ARV

Atmel ARV

Atmel ARV

- Arquitectura diferente a Arm-Cortex
- Tamén para microcontroladores
- Empezouse a desenvolver en 1996
- Popularizada por incluirse nos Arduino
  - ATmega8
  - ATmega168 (Arduino Nano)
  - ATmega328 (Arduino Uno)
  - ATmega1280 (Arduino Mega 1280)
  - ATmega2560 (Arduino Mega 2560)
  - SAM3X8E (Arduino Due, 32 bit)
    - Fabricado por Atmel, pero Arm Cortex-M3

Atmel ARV

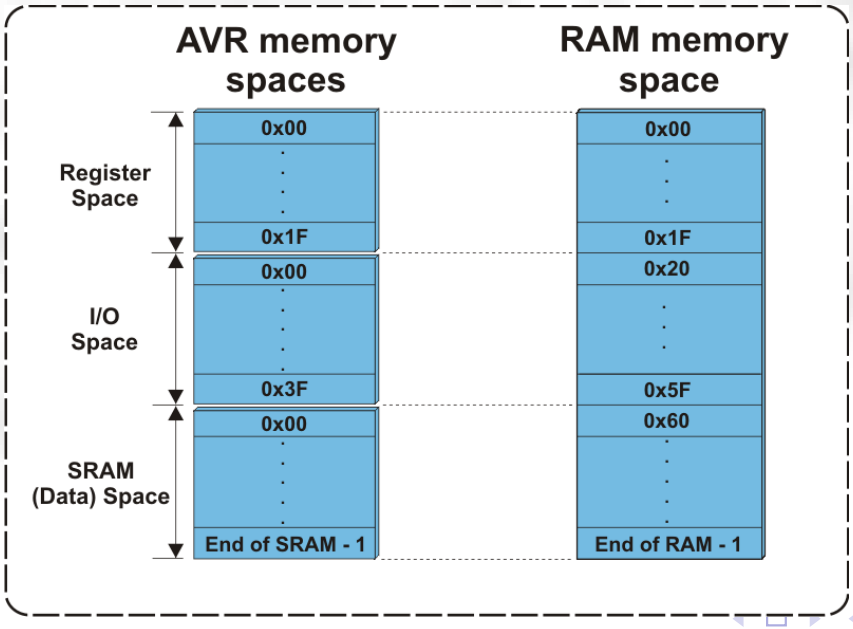
Atmel ARV Rexistros

- Principalmente de 8 bit
- 32 rexistros 8 bits:
  - Os 16 máis baixos usables por tódalas instruccións (limitacións para operandos inmediatos)
  - Os 6 últimos úsase para gardar direccións de memoria
    - X (R27:R26), Y (R29:R28), Z (R31:R30)
  - PC, SP
  - SREG: Status Register, 8 bits para flags
    - C, Z, N, V
    - S, para signo: $N \oplus V$ para comparacións
    - H, half carry, T, bit copy, I, interrupts enabled
  - RAMPX, RAMPY, RAMPZ, RAMPD e EIND
    - Úsanse para permitir chegar a direccións de memoria de ata 24 bits

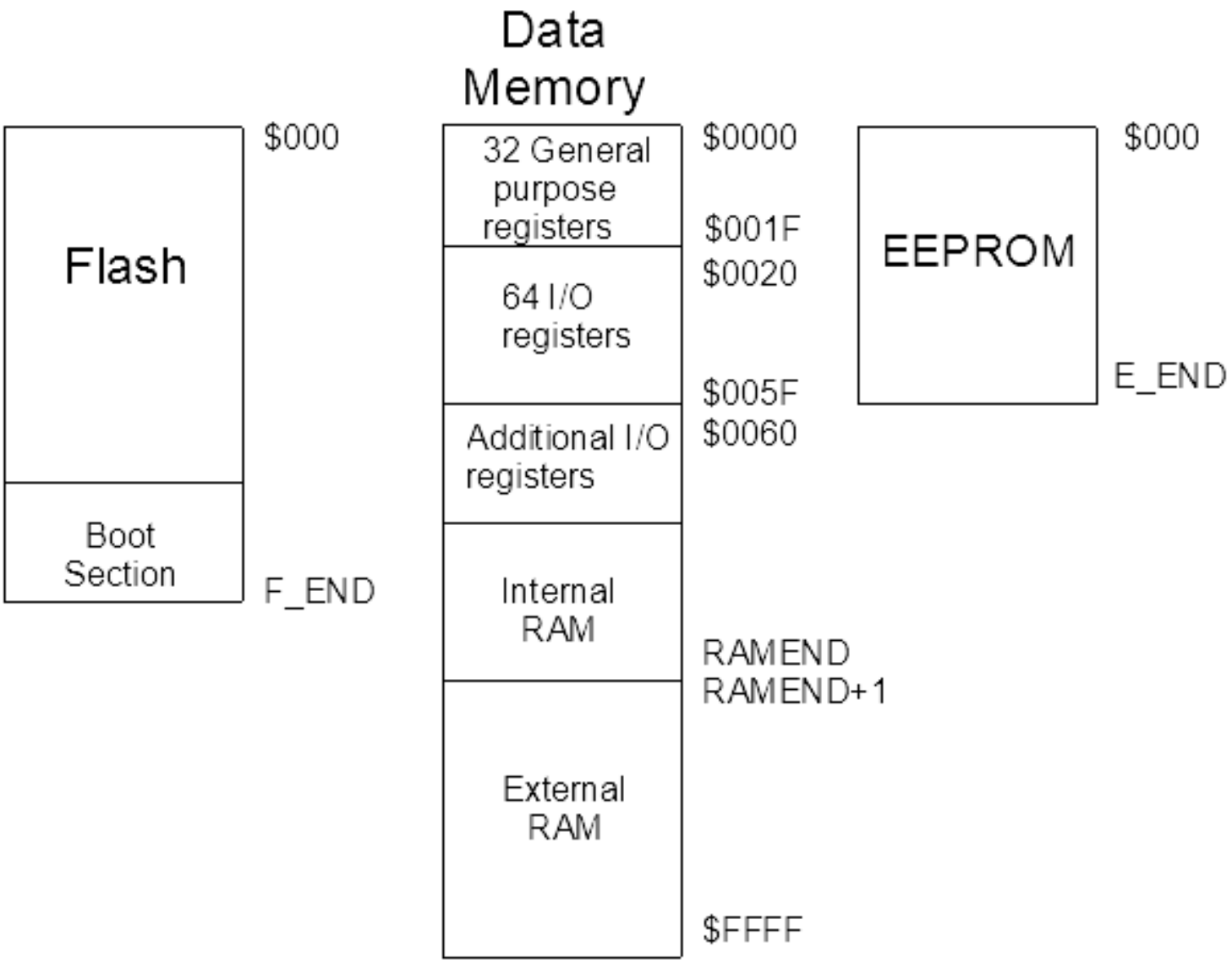
Atmel ARV

Atmel ARV Memoria

- Permite postincremento, predecremento, e inmediato
- Mapeanse os rexistros, a I/O (cunha parte direccionable bit a bit) e a RAM
- A memoria de instruccións vai aparte
- Certos rexistros tamén se poden direccionar bit a bit



Atmel ARV Memoria



Atmel ARV

Atmel ARV Instruccións

- Tardan 1 ciclo, salvo multiplicación e outras que operan en varios rexistros (2 ciclos)
- So se pode operar con rexistros
- So se pode acceder á memoria copiándoa aos rexistros (tarda 2 ciclos)
- As instrccións son de 16 bits (salvo algunha que use direccións de memoria de 16 ou 24 bits)
- Ten saltos que usan os flags da ALU
- Tamén ten ”skips” que saltan unha dirección dependendo de 1 bit nun rexistro ou I/O
- Podedes ver as instruccións en: AVR Wikipedia

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